Program Compilation for Large-Scale Quantum Computers

Thèse n. 8756
présentée le 14 décembre 2020
à la Faculté des sciences et techniques de l'ingénieur
programme doctoral en Génie électrique
École polytechnique fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Giulia Meuli

acceptée sur proposition du jury :

Prof. Andreas Peter Burg, président du jury
Prof. Giovanni De Micheli, directeur de thèse
Dr. Mathias Soeken, co-directeur de thèse
Prof. Paolo Ienne, rapporteur
Prof. Michele Mosca, rapporteur
Dr. Nikolaj Bjørner, rapporteur

Lausanne, EPFL, 2020

To my beloved family

Acknowledgements

First of all, I would like to express my gratitude to my thesis director, Prof. Giovanni De Micheli, for believing in me from the very first moment and for the continuous guidance and support. I would also like to thank my co-director Dr. Mathias Soeken, for guiding me during these years and for passing on the love for scientific research. I am thankful for all the fun moments we had during our many, sometimes little crazy trips.

My gratitude goes to the referees of my thesis, Prof. Michele Mosca, Dr. Nikolaj Bjørner, and Prof. Paolo Ienne, for their time and their precious feedback. I hope to have the opportunity to further discuss our research.

I am grateful to Dr. Martin Roetteler and Dr. Thomas Häner for the opportunity of working on such exciting summer internship projects.

I would like to thank all my special colleagues, that created the right environment for an amazing experience. Thank you, Ivan, for always making me smile and for being there for me in all kinds of situations. I am glad to have a friend like you.

Thanks, Eleonora, for our friendship, which now has deep roots in all the things we have in common and all the experiences we have had together.

Thanks to the best botanist and green thumb office mate, Simone. Thank you for the company, for your availability and the delicious presents from your hometown. It has been a pleasure sharing the office with you.

Thanks to Francesca C., for all the long talks and the fun activities. Your energy is contagious, thanks for sharing some with me.

Thank you, Giovanni, for all the excellent dishes you have cooked. You have dampened the feeling of homesickness with your excellent cooking skills. Thank you for all the amazing trips and events that you organized.

Thanks to Francesca S., Enrico T., and to the little Leo, for the sweetness and hospitality. Your family has always been a pillar of our community in times of need but also in times of celebration.

Although part of LSI only for a short time, I would like to thank Sofia N., Sara, Kate, Nadya, Lucia, Sofia M., Samuel and Abduwali for making the PhD experience a lot more fun.

Thank you Bruno for the discussions and collaborations on quantum computing and for all the fun moments we had together outside of the lab.

A special thank goes to all my present and past logic synthesis colleagues for the fruitful discussions: Heinz, Zhufei, Winston, Sonia, and Ferehste.

For always assisting me with bureaucratic problems (and non-), I would like to thank Cristina, Carol, and Chantal.

I am grateful to all the people that I have met during this amazing journey. First of all, I am grateful to Chiara for being among my closest friends. You have been an amazing flatmate, you are generous beyond imagination, and you always manage to cheer me up. Also, I would like to thank Giacomo, Jacopo, Nicoló, Carlotta, Matteo, Luca, Nadia, Mariazel, and Emanuele, for being my family far from home.

I would also like to thank my fellow summer interns, even if we have spent a little time together, you made me feel at home. A special thanks to Fernando, Harsh, Jessica L., Jessica S., and Ilaria. I hope that our paths will bring us together once more.

I am thankful to the people that supported me from my hometown: my friends, and my dear family. Lorenzo, I could not be luckier than having you as my long-time friend. Thank you for giving me strength and for inspiring me to become a better version of myself. Chiara, thank you for always cheering for me and my success. Thanks for your unique and true friendship. Alessandro, thank you for loving me, for your patience, for being by my side every day even when apart, for making me happy, and for taking care of me.

Finally, I would like to thank my family. You have supported me with love, attention, and dedication in my path of personal and professional growth. I have always felt close to you over the years and all my achievements, are also your achievements.

Lausanne, December 14, 2020

Abstract

Practical realizations of quantum computers are poised to deliver outstanding computational capabilities far beyond the reach of any classical supercomputer. While classical systems are reliably implemented using CMOS technology, the fabrication of quantum hardware has not yet reached a commercially viable level of maturity. Nowadays, many different technologies are being developed as competing candidates to build the first error-corrected quantum computer. Among the software resources required to operate such a system, quantum compilers translate a high-level description of a quantum algorithm into low-level, technology-dependent quantum instructions.

Many quantum algorithms, including Shor's and Grover's algorithms, require to compute some classical logic functions on the quantum computer. The first part of this thesis deals with the problem of compiling quantum circuits that perform such classical Boolean functions, called *oracles*, into a fault-tolerant set of instructions. Oracle circuits often demand many resources in terms of the number of qubits and gates. The focus is on compiling quantum circuits starting from multi-level logic networks representing large Boolean functions while minimizing the resource footprint of the obtained quantum circuits. At the same time, the trade-off between memory resources and the number of operations typical of such a compilation process is explored. As part of the effort in reducing the resources required to perform a given quantum program, I also present some quantum circuit optimization techniques.

The researched algorithms leverage data structures and techniques borrowed and adapted from classical logic synthesis, e.g., SAT solvers, LUT mapping, and multi-level logic networks. I implemented all the compilation algorithms presented in this thesis as part of open-source projects. In particular, I develop and maintain *caterpillar*—a C++ header-only library dedicated to the quantum compilation of oracles and quantum memory management.

The second part of this thesis describes how to equip quantum programming language compilers with automatic accuracy management. Despite the availability of many of such languages, resource estimation of quantum algorithms does not yet support taking approximation errors into account. This general methodology is applicable to any programming language and I demonstrate its integration into the Q# compiler. The technique consists of providing language support to ease the accuracy-aware programming task. The user can define accuracy parameters that will be automatically optimized according to a constraint and a cost function directly generated from the source code. In the presented practical evaluations, the constraint function is the overall allowed accuracy, while the cost function is application-dependent and related to the number of operations. During the optimization process, such functions are evaluated hundreds of times. For this reason, they are extracted as (near-)symbolic expressions, whose evaluation time does not depend on the quantum algorithm size.

The algorithms and the methodologies presented in this thesis are part of a widespread effort of the research community to build a complete and efficient software stack to program and control the first practical universal quantum computer.

Keywords: logic synthesis, quantum computing, quantum compilation, quantum programming, resource estimation

Estratto

I futuri computer quantistici promettono capacità di calcolo eccezionali, ben al di là della portata di qualsiasi supercomputer classico. Mentre i sistemi classici possono essere implementati in modo affidabile utilizzando la tecnologia CMOS, la fabbricazione di hardware quantistico non ha ancora raggiunto un livello di maturità commercialmente sostenibile. Al giorno d'oggi, diverse tecnologie sono in fase di sviluppo e sono in competizione tra loro per costruire il primo computer quantistico con correzione degli errori. Tra le risorse software necessarie per far funzionare un tale sistema, i compilatori quantistici traducono una descrizione ad alto livello di un algoritmo quantistico in istruzioni quantistiche a basso livello, dipendenti dalla tecnologia.

Molti algoritmi quantistici, compresi gli algoritmi di Shor e Grover, richiedono che il computer quantistico calcoli alcune funzioni logiche. La prima parte di questa tesi affronta il problema della compilazione di circuiti quantistici che eseguono tali classiche funzioni Booleane, denominati *oracle*, in un insieme di istruzioni supportato dai codici di correzione degli errori. I circuiti *oracle* spesso richiedono molte risorse in termini di numero di qubit e di operazioni. Per questo, la mia ricerca si focalizza su minimizzare tali risorse attraverso un'efficiente compilazione di circuiti quantistici a partire da reti logiche multilivello. Allo stesso tempo, la tesi esplora il *trade-off* tra la memoria e il numero di operazioni, tipico di un tale processo di compilazione. Nel quadro dello sforzo volto a ridurre le risorse necessarie per eseguire un dato programma quantistico, presento inoltre alcune tecniche di ottimizzazione dei circuiti quantistici.

Gli algoritmi oggetto di questa ricerca sfruttano strutture di dati e tecniche prese in prestito e adattate dalla sintesi logica, ad esempio: SAT *solvers*, LUT *mapping* e reti logiche multilivello. Ho implementato tutti gli algoritmi presentati in questa tesi come parte di progetti *open-source*. In particolare, sviluppo e mantengo una libreria C++ dedicata alla compilazione quantistica degli *oracle* e alla gestione della memoria quantistica.

La seconda parte di questa tesi descrive come dotare i compilatori dei linguaggi di programmazione quantistica di una gestione automatica della loro accuratezza. Nonostante la disponibilità di molti di questi linguaggi, la stima delle risorse degli algoritmi quantistici non prende ancora in considerazione gli errori di approssimazione. Questa metodologia è generalmente applicabile a qualsiasi linguaggio di programmazione e ne dimostro l'integrazione nel compilatore del linguaggio Q#. La tecnica consiste nel fornire, attraverso il linguaggio, un supporto per facilitare il compito di tenere in considerazione l'accuratezza. L'utente può definire parametri di accuratezza che saranno automaticamente ottimizzati secondo una

funzione di vincolo e una funzione di costo generati direttamente dal codice sorgente. Nelle valutazioni pratiche presentate, la funzione di vincolo è l'errore complessivo consentito, mentre la funzione di costo dipende dall'applicazione e si riferisce al numero di operazioni. Durante il processo di ottimizzazione, tali funzioni vengono valutate centinaia di volte. Per questo motivo, vengono estratte come espressioni (quasi) simboliche, il cui tempo di valutazione non dipende dalla dimensione dell'algoritmo quantistico.
Gli algoritmi e le metodologie presentate in questa tesi fanno parte di uno sforzo diffuso della comunità scientifica per costruire uno *stack* software completo ed efficiente per programmare e controllare il primo computer quantistico universale.

Parole chiave: sintesi logica, calcolo quantistico, compilazione quantistica, programmazione quantistica, stima delle risorse

Contents

Acknowledgements **i**

Abstract (English/Italian) **iii**

List of Figures **x**

List of Tables **xiii**

List of Acronyms **xv**

1 Introduction **1**

1.1 History of quantum computing 1

1.1.1 The rise of quantum mechanics 1

1.1.2 Computing with quantum physics 2

1.2 Quantum computing systems 4

1.2.1 Trapped ions 5

1.2.2 Superconducting circuits 6

1.2.3 Nitrogen vacancies and quantum dots 7

1.2.4 Anyonic particles 7

1.3 Software requirements 7

1.3.1 Model of computation 7

1.3.2 Control software 9

1.3.3 Debugging and testing 9

1.4 Quantum compilers 10

1.5 Thesis contribution 11

1.5.1 Compilation of combinational logic 11

1.5.2 Accuracy management 14

1.6 Thesis organization 15

I Background **19**

2 Logic synthesis **21**

2.1 Logic representations 21

2.1.1 Boolean functions 21

2.1.2 Truth tables 22
2.1.3 ESOP representation of Boolean function 22
2.1.4 Multi-level logic networks 23
2.2 Classification of Boolean functions 24
2.3 Look-Up Table (LUT) mapping 27
2.4 Boolean satisfiability (SAT) problem 28
2.5 Summary 29

3 Quantum computing 31
3.1 Quantum computing principles 31
3.1.1 Qubits 31
3.1.2 Gates 32
3.1.3 Universal quantum libraries 32
3.1.4 Measurements 34
3.1.5 Quantum circuits 34
3.2 Phase polynomial representation 35
3.3 Summary 36

4 Reversible circuits 39
4.1 Reversible gates 39
4.2 Synthesis of reversible circuits 40
4.2.1 ESOP-based reversible logic synthesis 41
4.2.2 Hierarchical reversible synthesis 41
4.3 Decomposing reversible gates into quantum circuits 42
4.3.1 Gray synthesis 42
4.3.2 Quantum circuits for the Toffoli gate 42
4.4 Summary 44

II Quantum circuit synthesis and optimization 45

5 Hierarchical reversible synthesis 47
5.1 Method overview 47
5.2 Quantum-aware k-LUT mapping 50
5.2.1 Cut enumeration and costing 50
5.2.2 XOR-block matching 51
5.3 Quantum memory management 52
5.3.1 Reversible pebbling game 54
5.3.2 SAT-encoding 55
5.3.3 Show-cases 57
5.4 Results 59
5.5 Summary 62

6 Compiling xor-and-inverter graphs 63

6.1 Methods overview . . . 63
6.2 Xor-and-inverter graphs . . . 65
6.3 Methods . . . 67
6.3.1 Algorithm 1: minimizing the T-count . . . 67
6.3.2 Algorithm 2: minimizing the T-depth . . . 69
6.3.3 Algorithm 3: minimizing the number of qubits . . . 70
6.4 Results . . . 72
6.4.1 Improving the T-count versus T-depth . . . 73
6.4.2 Qubits/T-count trade-off . . . 73
6.4.3 Discussion . . . 74
6.5 Summary . . . 77

7 Single-target gate decomposition **79**
7.1 Precomputed quantum circuits . . . 79
7.1.1 Spectral equivalence in quantum compilation . . . 80
7.1.2 Algorithms for the synthesis of the database . . . 82
7.1.3 Database of optimized quantum circuits . . . 84
7.2 ESOP-based decomposition . . . 86
7.2.1 Optimal ESOP for quantum compilation . . . 86
7.2.2 Constraint-based ESOP synthesis . . . 87
7.2.3 Results . . . 91
7.3 LUT-based decomposition . . . 96
7.4 Experimental results . . . 98
7.5 Summary . . . 98

8 Quantum circuit optimization **101**
8.1 T-count optimization using graph matching . . . 101
8.1.1 Optimization properties . . . 101
8.1.2 Graph matching problem . . . 103
8.2 SAT based {CNOT, T} optimization . . . 104
8.2.1 Linear reversible functions . . . 104
8.2.2 SAT-based algorithm for the synthesis of {CNOT, T} circuits . . . 105
8.2.3 SAT-based rewriting algorithm . . . 109
8.2.4 Results . . . 109
8.3 Summary . . . 110

III Compiler-integrated accuracy management **113**

9 Automatic accuracy management **115**
9.1 Errors in quantum compilation . . . 117
9.1.1 Synthesis errors . . . 117
9.1.2 Phase estimation errors . . . 117
9.1.3 Algorithmic errors . . . 117

9.2 Approximation errors in quantum circuits 118
9.3 Sample quantum programs 120
9.3.1 Quantum Fourier transform 120
9.3.2 Simulating time-evolution of operators 121
9.3.3 Quantum phase estimation 122
9.4 Adding language support for accuracy management 123
9.5 Automating accuracy management 125
9.5.1 Cost/constraint functions: extraction 126
9.5.2 Cost/constraint functions: optimization 127
9.6 Compiler and language requirements 129
9.6.1 LLVM prototype 130
9.6.2 Q# integration 133
9.7 Qualitative evaluation 135
9.8 Quantitative evaluation 136
9.9 Summary 138

IV Open-source development 141

10 Caterpillar 143
10.1 Compiling with caterpillar 144
10.1.1 Mapping strategies 145
10.1.2 Pebbling strategies 147
10.1.3 Mapping strategies for XAGs 148
10.2 Show-case: resource estimation for large quantum circuits 148
10.3 Show-case: compilation with a fixed number of helper qubits 149
10.4 Show-case: looking for the minimum pebbling number 150
10.5 Summary 151

11 Conclusions and future directions 153

A Database integrated into RevKit 159

Bibliography 161

Curriculum Vitae 177

List of Figures

1.1 NISQ systems per year of release. Labels: <*name*> Q<*number of qubits*> QV<*quantum volume*>. 4

1.2 Sequence diagram illustrating how the QPU interacts with a host CPU, image from [1]. 8

1.3 Different stages of the offline software stack. Image from [2]. 10

2.1 Electronic Design Automation (EDA) flow. 22

2.2 An XAG for the majority of 3-inputs. 24

2.3 (a) An AIG graph performing the function $f = (\overline{x}_1 + \overline{x}_2)x_3\overline{x}_4$ with two possible 3-feasible cuts; (b) 3-LUT network generated by cut_1; (c) 3-LUT network generated by cut_2. 27

2.4 LUT mappings for the function prime$_6$. 28

3.1 The Bloch sphere. 32

3.2 (a) A quantum circuit computing an entangled state and (b) the corresponding expected measurement outcomes. 35

3.3 A {CNOT, T} circuit. 36

4.1 Examples of reversible gates. 40

4.2 Notations for the multiple-controlled Toffoli: the left gate is $T_{1\wedge 2}(\{1,2\},3)$ in the complete notation and $T(\{1,2\},3)$ in the special notation; for the second gate $T_{\overline{1}\wedge 2}(\{1,2\},3)$ and $T(\{\overline{1},2\},3)$. 40

4.3 Example of mapping a single-target gate into Toffoli gates using ESOP decomposition. 41

5.1 (a) Circuit computing $y = g(f(x_1, x_2), x_3)$ with two single-target gates generating an unknown intermediate result. (b) Garbage-free circuit where the intermediate result has been uncomputed by applying f twice. 48

5.2 k-LUT-based hierarchical quantum compilation flow illustrating the example of a three-input multi-level logic network performing the Boolean function f (a), mapped into a 2-LUT network for f (b), translated into the reversible circuit for the 2-LUT network using the Bennett clean-up strategy (c) then decomposed into a quantum circuit (d). 49

5.3 Mapping into qubits LUTs performing the XOR function. 51

5.4 (a) Example of a DAG. (b)(c)(d) Three different uncomputing strategies: (a) Bennett strategy; (b) space-optimized by reordering; (c) space-optimized by increasing the number of gates. 53
5.5 Two different pebbling strategies for a given DAG. 55
5.6 Illustration of how *pebbling* maps the given computation into a fixed number of helper qubits: respectively 24 (Add:28, Sub:20, Sqrt:15, Mult:11), 20 (Add:36, Sub:32, Sqrt:21, Mult:9), 16 (Add:28, Sub:24, Sqrt:17, Mult:13) , 12 (Add:24, Sub:34, Sqrt:19, Mult:13) and 10 (Add:34, Sub:38, Sqrt:25, Mult:13). 58
5.7 (a) LHRS flow; (b) ROS flow. 60
5.8 Qualitative description of ROS's capability. 60

6.1 Illustration of the compilation of an AND step of an XAG into a quantum circuit computing its linear transitive fan-in cones in-place using CNOT gates. 66
6.2 An XAG graph representing the majority-of-three Boolean function. 66
6.3 Example in which one transitive fan-in is included in the other and the computed intermediate value can be reused. 69
6.4 Illustration of how algorithm 6.2 compiles one level with two AND nodes, x_i and x_s, into a quantum circuit with one single T-stage. 69
6.5 Illustration of a pebbling strategy for the input DAG (a) using 3 pebbles and 6 moves (b)-(g), and the corresponding compiled reversible circuit of Toffoli gates (h). 71
6.6 Illustration of how sections of the XAG are compressed in a box node of the abstract network. 71
6.7 Optimized and non-optimized pebbling results using different number of pebbles for selected logic networks. 75

7.1 Reversible circuits implementing the five spectral invariant operations. 80
7.2 Synthesized reversible circuit for the function f, obtained performing all the spectral operations to retrieve f from the optimal implementation available in the database f_d ($^{\#}$0x0888). 81
7.3 Bar plots showing the cost functions characterizing the three different implementations for the database functions (Q, TC, TD). The y-axes report: (a) average ratio between number of qubits and best number of qubits in the database, (b) average ratio between T-count and best T-count in the database, (c) average ratio between T-depth and best T-depth in the database. Numbers on the bars show average values of the respective cost function for the three implementations. 84
7.4 Synthesis results for two different ESOPs representing the same function f. . . 87
7.5 Three possible ESOP covering for the function $f = x_1 \vee x_2$. 89
7.6 Histogram showing the improvement of exact methods over *PKRM* with respect to two different cost functions: number of terms (*EXACT(unit)*) and number of literals (*EXACT(lit)*). 91
7.7 Two different state-of-the-art compilation flows for Boolean functions that use ESOP-based reversible synthesis. 93

7.8 Example of applying a LUT-based mapping method to single-target gates obtained from the hierarchical reversible synthesis flow. 97

8.1 Rule 2 example. Equivalence rules from [3]: (a) D1, (b) D7, (c) D1. 102

8.2 Rule 3 example. Equivalence rules from [3]: (a) D2, (b) D3. 102

8.3 Example of a linear reversible CNOT circuit. 105

8.4 Illustration of SAT encoding for sample circuit in Example 8.2.3. 106

9.1 Efficient circuit computing the quantum Fourier transform. 120

9.2 Quantum circuit performing quantum-phase estimation on an n-qubit system with an accuracy of $k+1$ bits. 122

9.3 Flow diagram explaining how the code of the QPE algorithm is transformed into a code evaluating the overall approximation error ε. 125

9.4 C++ code for the approximate QFT as consumed by the LLVM prototype. 131

9.5 Q# implementation of the approximate quantum Fourier transform with epsilon declarations. 133

9.6 A comparison between the runtimes of the non-symbolic approach and the symbolic approach developed in this thesis is shown. Each data point marks the runtime required for a single evaluation of the T function plus a single evaluation of the E function for the QPE algorithm with approximate QFT and for Shor's algorithm. Optimized accuracy parameters are used as input in order to assert that the total approximation error is at most $5 \cdot 10^{-3}$. I provide polynomial extrapolations from the collected runtimes for the non-symbolic approach as the runtime tops out for bit sizes above 512 for Shor and 32 for QPE. 138

10.1 *Caterpillar*'s logo 143

A.1 Representative functions, starting point Toffoli networks, and obtained T-count for single-target gates with 4-input control functions. 159

A.2 Representative functions, starting point Toffoli networks, and obtained T-count for single-target gates with 5-input control functions. 160

List of Tables

5.1 Comparison between the Bennett and the pebbling strategy. 59

5.2 Comparison between *ROS* and *LHRS*. 61

6.1 Compilation results for several arithmetic, floating point and cryptographic designs. 74

7.1 Specifications of the quantum circuits in the open-source database. 85

7.2 Comparison of different ESOP synthesis methods. 92

7.3 Comparison between exact method and heuristic for small reversible functions 94

7.4 Comparison between PKRM and the pseudo-exact optimal ESOP synthesis integrated into LHRS to synthesize the EPFL arithmetic benchmark 95

7.5 Experimental evaluation of different single-target gate decomposition methods in LHRS on the EPFL arithmetic benchmarks. 100

8.1 Results of the SAT-based rewriting algorithm. 111

9.1 Compiler transformations used to optimize the cost and constraint functions. . 127

9.2 Symbolic expressions for the T-count (T) and total approximation error (E) as extracted by the LLVM prototype from the source code of various example programs. 137

List of Acronyms

AIG And-Inverter Graph. 21, 25, 26, 45, 92, 140

ASIC Application Specific Integrated Circuit. 10

CNF Conjunctive Normal Form. 26

CNOT controlled-NOT. 101

CPU Central Processing Unit. 7, 8

DAG Directed Acyclic Graph. 48–51, 54, 141, 143, 146

DBS Decomposition-Based Synthesis. 13, 88

EDA Electronic Design Automation. 19, 27

ELU Elementary Logic Unit. 5

EPFL École Polytechnique Fédérale de Lausanne. 139

ESOP Exclusive Sum-Of-Products. xiii, xvi, xvii, xix, 13, 14, 27, 75, 78, 82–89, 91–94, 96, 140, 141, 153, 154

FPGA Field Programmable Gate Array. 25

HLS High Level Synthesis. 19

HPC High-Performance Computing. 7

IC Integrated Circuit. 19

ISA Instruction Set Architecture. 8

LHRS LUT-based Hierarchical Reversible Synthesis. xvi, 45, 56

LLVM Low-Level Virtual Machine. 112, 116, 134, 135

LSI Integrated Systems Laboratory. 139

LUT Look-Up Table. xiii, xvii, 15, 21, 27, 46, 58, 59, 72, 75, 76, 89, 92–94, 96, 100, 140, 141, 151, 152, 154

MIG Majority-Inverter Graph. 21, 22, 140

MWY Many Worlds Interpretation. 2

NISQ Noisy Intermediate-Scale Quantum. 3, 151

NPN Negation-Permutation-Negation. 22

NV Nitrogen Vacancy. 7

PKRM Pseudo-Kronecker Reed Muller. 141

QCA Quantum Cellular Automata. 22

QCU Quantum Control Unit. 8

QEC Quantum Error Correction. 9

QPU Quantum Processing Unit. 7, 8

QRAM Quantum Random Access Machine. 8, 9, 32

QV Quantum Volume. xv, 5, 6

ROS Resource-constrained Oracle Synthesis. xvi, 55, 56

RSA Rivest–Shamir–Adleman. 3

RTL Register Transfer Level. 19

STG Single-target gate. xiii, 48, 49, 75–94, 96

TBS Transformation-Based Synthesis. 13

XAG Xor-And-inverter Graph. xiv, 15, 21, 22, 59–61, 72, 78, 140, 141, 143–147, 152, 153

XMG Xor-Majority-inverter Graph. 21, 140

1 Introduction

1.1 History of quantum computing

1.1.1 The rise of quantum mechanics

At the beginning of the twentieth century, what we now call *classical* physics was no more capable of explaining observed phenomena in nature. For this reason, it was common to formulate an ad-hoc corrective hypothesis to justify *experimentally observed* phenomena in radiation and atoms theory, and such a hypothesis needed continuous updates.

Theories in physics were predicting absurdities such as the *ultraviolet catastrophe*, a phenomenon for which ideal black bodies should emit an arbitrarily high energy with decreasing emitted wavelength. Consider Rutherford's atomic model [4], theorized in 1911 and based on the well-known gold foil experiment performed in 1909. According to Rutherford, the atomic mass is concentrated at the center of the atom, in the so-called *nucleus* (Rutherford did not use this term in his paper). The nucleus is characterized by a positive charge. Electrons, negatively charged, orbit around the nucleus. According to Maxwell's theory of electromagnetism [5], orbiting negative charges should emit radiation and lose energy. As a consequence, if atoms were subject to the laws of classical physics only, electrons should quickly and inexorably collapse into the nucleus following a spiral trajectory. The amount of time it would take an electron to crash into the nucleus of its atom is about ten picoseconds [6].

It is from similar observations that pioneer scientists developed the wave-particle duality theory. Plank and Einstein showed how light waves could exhibit particle-like properties. In 1900, Plank deduced that electromagnetic radiation could only be emitted and absorbed in discrete packets, i.e., *quanta*, of energy, hence deriving the right intensity spectral distribution for the black body [7]. In 1905, Einstein postulated that Plank's quanta were actual localized particles, later called photons [8]. This theory also explained the photoelectric effect: a phenomenon for which a material hit by electromagnetic radiation emits electrons, only if the radiation's frequency exceeds a given threshold, and independently from its intensity. Besides, De Broglie hypothesized in his doctoral thesis that particles also could have wave-like properties [9]. In 1926, Schrödinger proposed that electrons in the atoms behave like waves and follow Schrödinger's equation [10], which is solved by a series of wave functions associated with a specific energy. This theory led to the quantum mechanical model of the atom, which is still state-of-the-art.

This duality is reflected in the well-known Copenhagen interpretation of quantum mechanics. This interpretation states that a quantum system is modeled by wave functions and evolves according to the Schrödinger equation. Certain observable quantities, e.g., the position of electrons in an atom, can be measured and the result obtained will depend on the square of the amplitude of the wave function. Measurements cause the wave function to collapse on the obtained result. This interpretation is the one currently accepted and taught to physics students, even if many questioned the distinction between the microscopic quantum world and the macroscopic classical world that the Copenhagen interpretation implies. This distinction suggests that one should not even try to understand quantum behavior [11]. Using Bohr's words: "We must be clear that when it comes to atoms, language can be used only as in poetry. The poet, too, is not nearly so concerned with describing facts as with creating images and establishing mental connections.".

A new interpretation is gaining increasing consent: the *Many Worlds Interpretation (MWY)*, proposed in 1957 by Hugh Everett [12] and named by Bryce DeWitt [13]. This interpretation, which exists in many variants, does not make a distinction between the microscopic quantum world, led by the rules of quantum mechanics and the Schrödinger equation, and the macroscopic world that we experience. On the opposite, it states that all the possible outcomes of a quantum experiment exist at the same time in different worlds.

1.1.2 Computing with quantum physics

In the 1980s researchers started to look at the opportunity of using quantum mechanics to boost computing. Feynman postulated that quantum systems could be simulated by quantum computers [15]. In 1985 Deutsch [16] theorized an "universal quantum computer", with properties that are not reproducible by any Turing machine. Using his words during an interview at Wired Magazine in 2007: "The most important application of quantum computing in the future is likely to be a computer simulation of quantum systems, because that is an application where we know for sure that quantum systems in general cannot be efficiently simulated on a classical computer. This is an application where the quantum computer is ideally suited". Later, together with Jozsa, he formulated the *Deutsch-Jozsa algorithm* [17]. He was a supporter of the MWY, which he calls the *multiverse*.

In 1994 Peter Shor proposed an algorithm that solves prime factorization exponentially faster than classical computers [18]. The advent of Shor's algorithm suggested that problems classically considered hard to solve and consequently used in encryption algorithms, e.g., the Rivest–Shamir–Adleman (RSA) algorithm [19], may be solved on a quantum computing system within hours, but would take billions of years on a classical system. In 2001 IBM demonstrated

(a) Left: Artist's rendition of Google's Sycamore processor mounted in the cryostat. (Forest Stearns, Google AI Quantum Artist in Residence) Right: Photograph of the Sycamore processor. (Erik Lucero, Research Scientist and Lead Production Quantum Hardware)

(b) System One quantum computer (Image Credit IBM [14]

Shor's algorithm on a 7 qubits system [20]. In 1996 Grover introduced a quantum search algorithm with a quadratic speed-up with respect to classical solutions. Unfortunately, a quantum system capable of performing such an algorithm for sizes that cannot be classically simulated is not available at the time of writing this manuscript.

In 2000 David DiVincenzo laid out the requirements that a quantum system must comply with to be able to perform computation [21]. These criteria are known as *DiVincenzo's criteria*. The first five are necessary for quantum computation: (i) an extensible register of two-level systems, usable as qubits; (ii) the ability to initialize the state of the qubits to a simple fiducial state; (iii) long relevant decoherence times; (iv) a "universal" set of quantum gates; (v) a qubit-specific measurement capability. The remaining two are necessary for quantum communication: (i) the ability to interconvert stationary and flying qubits; (ii) the ability to faithfully transmit flying qubits between specified locations. According to these criteria, new hardware technologies

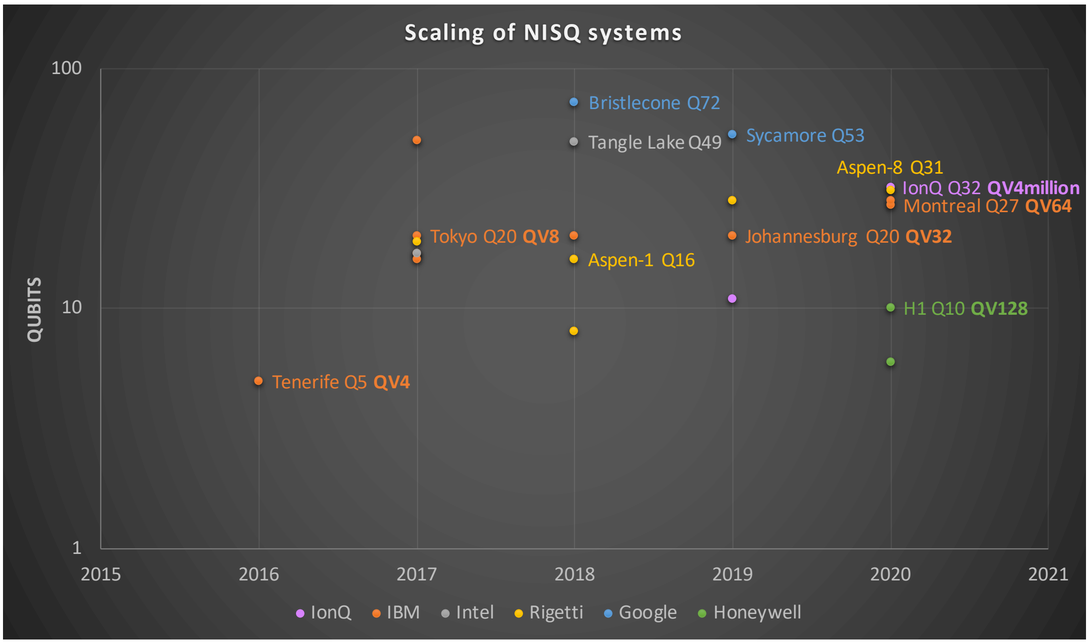


Figure 1.1 – NISQ systems per year of release. Labels: <*name*> Q<*number of qubits*> QV<*quantum volume*>.

are developed as candidates to build scalable quantum computers.

1.2 Quantum computing systems

In the last few years, different computing systems have been developed, known as Noisy Intermediate-Scale Quantum (NISQ) computers. They owe their name to some shared features: few noisy qubits and low-fidelity operations.

Even if current quantum systems are decades of development away from becoming error-corrected universal quantum computers, research is focusing on proving that they are capable of performing tasks beyond the reach of ordinary digital computers. This milestone, called *quantum supremacy*, has been proposed for the first time by Preskill in 2012 [22]. Quantum complexity theory shows that it is not necessary to develop a universal quantum computer to demonstrate quantum supremacy [23]. Besides quantum supremacy, the term *quantum advantage* is often used. Nevertheless, according to Preskill [24]: “In a race, a horse has an advantage if it wins by a nose. In contrast, the speed of a quantum computer vastly exceeds that of classical computers, for certain tasks. At least, that’s true in principle.".

IBM has recently proposed the Quantum Volume (QV) [25] as a metric to evaluate the ability of a quantum computer to perform complex computation. The complexity of an algorithm is relative to the circuit *width*, i.e., the number of qubits and to the circuit *depth*, i.e., the number of operations performed. The latter depends on many factors such as decoherence time, qubit fidelity, interconnections, etc.. Randomized model circuits are used to evaluate the system's capabilities: the metric represents the largest squared-shaped model circuit that the system can perform successfully. The QV is now used to compare different quantum computers and enables researchers to quantify the impact of new solutions and design choices.

In the following sections, I will describe four different generations of quantum computers [2], characterized by technologies with promising scalability. The ultimate goal is to fabricate an universal quantum computer, characterized by a large number of qubits, supporting error correction, and long coherence time. Such computer would be capable of performing quantum algorithms promising exponential speed-ups. Fig. 1.1 reports the currently available systems by year of release, showing their number of qubits (Q) and quantum volume (QV), when known.

1.2.1 Trapped ions

The first generation is led by the trapped-ion technology [26]. In an ultra-high vacuum, individual atoms are ionized and trapped using controlled electrical fields. Ions are then used to perform quantum computation by means of an array of individual laser beams (one per ion) and a global beam: the interference between the beams gives the required energy to switch a qubit's state.

The idea of computing with trapped ions and the demonstration of fundamental quantum logic gates date back to 1995 [27, 28]. One approach to build a large computing system is the monolithic design proposed in [29], in which segmented ion traps are fabricated and connected together to build the overall system. Segmented ion traps [30, 31] are architectural models that use DC electrodes to move the ion along trapping pathways and enabling interaction between qubits, i.e., multi-qubit gates. Opposite to the monolithic approach is the fabrication of many Elementary Logic Unit (ELU), optically interconnected [32].

Trapped ions exhibit very long qubits' coherence times, up to 600 seconds [33]. Another advantage of this technology is that both single and two-qubit gates can be implemented with very high fidelity [34]. The achieved state preparation and readout fidelities are better than the one demonstrated by any other technology [35]. Finally, they do not require long calibration steps, as, e.g., superconducting qubits. Even if trapped ions have the highest coherence time over gate time ratio, performing each operation is an order of magnitude slower when compared to the superconducting technology, i.e., typically between 1 to 100 μs against tens of nanoseconds. Another drawback is size, as the scaling is slowed down by the fabrication of the required optical control. Finally, this technology is characterized by a high fabrication cost.

Honeywell has presented a 10-qubit system, called H1, with a quantum volume of 128 [36]. The company IonQ develops trapped ions quantum computers with configurable architectures of fully connected qubits. In particular, it has tested single-qubit gates on 79 qubits [37]. IonQ has recently released its 32-qubit quantum computer achieving the quantum volume of 4 millions [38], which is the highest ever tested.

1.2.2 Superconducting circuits

One of the leading technology to build quantum computing systems is based on superconducting qubits. The first superconducting qubit has been demonstrated in 1999 [39].

Google, Intel, and IBM are developing superconducting quantum technology. IBM is one of the leading companies in quantum computing hardware and since the realization of the first computer in 2016, with 5 qubits, makes the system available on a cloud service, i.e., IBM's quantum experience[1]. In 2017, IBM released on its platform other two quantum computers with 16 and 20 qubits. In January 2018, Intel presented its 49-qubit superconducting quantum processor, Tangle Lake. In March 2018, Google Quantum AI Lab announced its new processor Bristlecone with 72 qubits [40]. In January 2019, IBM announced the first commercial circuit-based integrated 20-qubit quantum computer "IBM Q System One" and the "IBM Q Network" platform [41]. In October 2019, IBM made its 53-qubit system available [42]. At the same time, Google announced the achievement of quantum supremacy, intended as the ability to outperform classical computers on a specific task [43]. Such results were demonstrated on a 53-qubit superconducting computer called Sycamore. Sycamore completed in few seconds a task that a state-of-the-art supercomputer would require approximately 10,000 years to perform, according to Google. Later, IBM expressed some criticism on Google's achievement [44], pointing out that the task can be performed faster on certain classical computing processors, i.e., in only 2.5 days. IBM also pointed out that according to the first formulation of quantum supremacy, as by Preskill [22], this is only achieved if the selected task is unfeasible on classical computers.

IBM has computed the QV of several of its quantum computers, namely the 5-qubit Tenerife (2017), the 20-qubit Tokyo (2018), the 20-qubit Johannesburg (2019), and the 27-qubit Montreal (2020), showing an exponential growth resembling the well known Moore's law. The Montreal quantum computer achieves a quantum volume of 64 [45].

In addition to IBM, Intel, and Google, the start-up company Rigetti focuses on building superconducting quantum computers. The company has released in 2019 a quantum system called Aspen-4, for which a quantum volume of 8 has been demonstrated [46]. Rigetti's latest quantum computer, called Aspen-7 features 28 qubits and average gate times of 80ns-340ns, depending on if it is a single- or a two-qubit gate.

Supercomputing qubits have a smaller footprint and faster intrinsic gate times with respect to ion traps, reaching physical speeds of MHz. Nevertheless, fabrication and placement of the necessary control wiring for large qubits array is an unsolved engineering problem, that may compromise the scalability of these systems. Finally, this technology requires many calibration

[1] https://web.archive.org/web/20160927160441/http://www.research.ibm.com/quantum/

steps, due to the device mismatch which characterizes the fabrication process.

1.2.3 Nitrogen vacancies and quantum dots

The third generation of quantum computers counts extremely promising but less developed technologies with respect to the previous generations. The technologies in this generation promise high speed, in the order of the GHz and small sizes (nm-um). Impurities in diamond have long been of interest as a potential technology for both large-scale quantum computing and communications, in particular the one based on the *Nitrogen Vacancy (NV)* [47]. Diamond-based quantum technology do not require high infrastructure cost, a vacuum is not needed, and it works at a temperature of *only* 4K, rather than the 4mK needed by the superconducting qubits.

Another promising technology is the one based on quantum dots, which traps electrons at the boundary between different superconductors. The single qubit has been demonstrated in [48], and a two-qubit gate is realized in [49]. Quantum dots have the potential speed and integration density, thanks to the advanced silicon fabrication technology. This technology has demonstrated sufficient thermal robustness to enable computing a universal set of quantum gates, hence including two-qubit gates, at temperatures greater than 1K [50]. Intel is currently researching this technology, which promises small physical sizes, stability, and duration.

1.2.4 Anyonic particles

The fourth and last generation consists of the anyonic or topological quantum computers [51]. Such systems aim at suppressing errors using the fundamental physics of the system, with states naturally protected from decoherence. Even if this technology promises easy-to-scale systems with low error rates, the existence of anyonic (or Majorana) particles in engineered systems has not been yet demonstrated.

1.3 Software requirements

Previous sections have discussed the requirements of quantum computing hardware according to DiVincenzo's criteria and described state-of-the-art computing systems. This section focuses on the software resources necessary to operate a quantum computer. It starts with a description of the model of computation using a Quantum Processing Unit (QPU) and gives details on the required control software.

1.3.1 Model of computation

A quantum computer is not to be considered as an independent computing system but as a co-processor to classical computation [1], integrated into a future High-Performance Computing

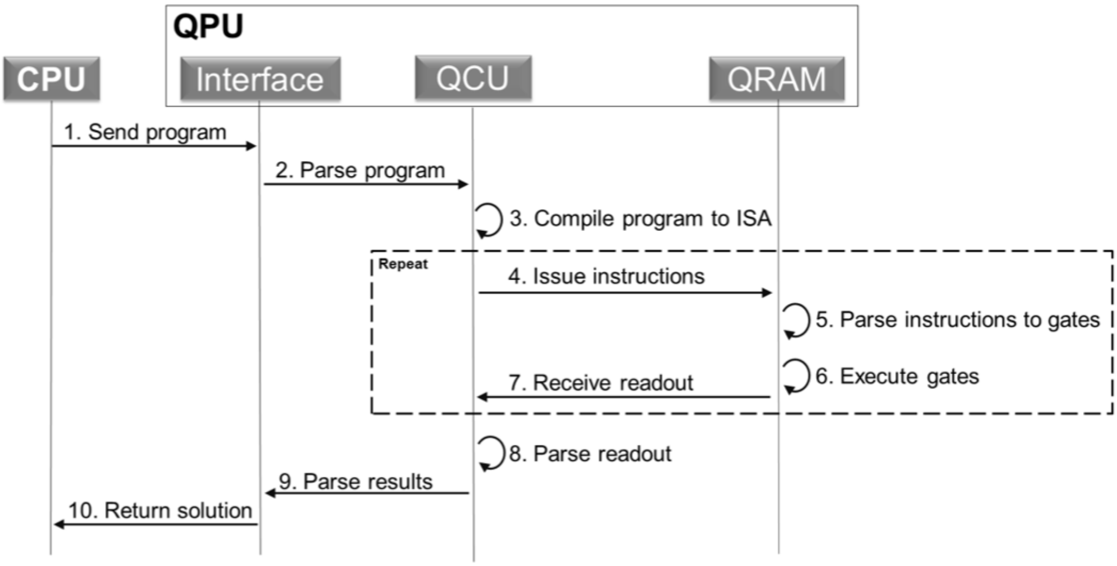


Figure 1.2 – Sequence diagram illustrating how the QPU interacts with a host CPU, image from [1].

(HPC) architecture. Fig. 1.2 (from [1]) shows a sequence diagram that models the interaction between the Central Processing Unit (CPU) and the Quantum Processing Unit (QPU).

The computing core of the QPU is the Quantum Random Access Machine (QRAM), a machine with the ability to perform a set of operations on quantum registers: quantum state preparation, unitary operations, and measurements [52]. Computation on a quantum register consists of operations modifying the state of the qubits. Such states are in a superposition of the classical states $|0\rangle$ and $|1\rangle$ and are represented by two complex coefficients which encode the probability to obtain the classical 0 or 1 upon measurement in the computational basis. Any gate operating on a qubit state has to guarantee that the probabilities of measurement outcomes sum up to one, and as such, they are represented by norm-preserving unitary matrices. Hence, any quantum operation is reversible. The QRAM receives instructions from a predefined set of unitary operations. The set of unitaries is a *universal quantum library*, as it must allow the QRAM to perform any quantum algorithm with arbitrary precision.

In addition to the QRAM, the QPU includes: an interface layer with the CPU, which parses the high-level description of the quantum program sent by the CPU; and the Quantum Control Unit (QCU), which compiles the program to an Instruction Set Architecture (ISA), describing the technology-independent instruction set available to perform on the quantum computer. The QRAM receives this program and executes it on the quantum registers. At the end of the instruction sequence, the QRAM performs measurements whose classical results are returned to the QCU and may be used to determine the quantum program to perform next. This is repeated until the final results of the program are available to the QCU, which sends them to the CPU to conclude the process.

1.3.2 Control software

Fig. 1.3 (from [2]) represents the software requirements to programming a large-scale quantum computer. The identified four elements of the software design stack are responsible for the quantum compilation of the algorithm, the optimization of the resulting quantum circuit and the mapping into the hardware. They are part of the resources needed for the so-called *offline control* of quantum computation. When operating the quantum computer, software libraries are required to enable working in tandem with the classical controller. Such libraries define the *online control*, which includes, e.g., dynamically decoding and correcting errors.

Initially, a high-level quantum programming language is used to describe the algorithm. To date, several of such languages have been developed [53, 54]. Relevant examples are: Q# (Microsoft) [55], Qiskit (IBM) [56], PyQuil/Quil (Rigetti) [57], Circ (Microsoft) [58], Quipper [59], Scaffold/ScaffCC [60], and ProjectQ [61]. Some of these frameworks provide domain-specific languages with the necessary abstractions, libraries, simulators, and cloud access to small-scale quantum computers. Some of them are embedded in classical host languages.

The first element of the stack, called *algorithmic design*, is responsible of expressing the high-level specification of the algorithm as a quantum/reversible circuit, optimized for space (number of qubits/inputs) and time (number of quantum/reversible gates). Please note that all quantum gates are reversible, as they represent unitary matrices, and for this reason, most software stacks use reversible networks as intermediate representations. In the second element of the stack, the circuit is converted into quantum gates. Large-scale quantum computation requires error-correcting codes. For this reason, the gate sequences that the QRAM receives must be compatible with fault-tolerant error correction. Then, the sequence is minimized with respect to typical cost functions for fault-tolerant circuits. In Chapter 3, further details on such cost functions are provided. Once the quantum circuit is specified using a fault-tolerant library, the third stack element performs topological optimization. This consists of converting the circuit into geometric structures for topological QEC [62]. The last stack element is responsible for the final mapping into the hardware, including selecting native operations on each specific qubit set and measurements.

1.3.3 Debugging and testing

Another fundamental challenge of quantum software is debugging and testing. In general, the characteristic superposition capabilities of quantum systems make them ill-suited to be simulated by classical machines. Nevertheless, there are some solutions that are being currently explored. The most straightforward is to run test problems that are small enough to be classically simulated. For example, Microsoft found errors in state-of-the-art circuits for quantum chemistry by automatically generating small circuits [63]. There are some quantum

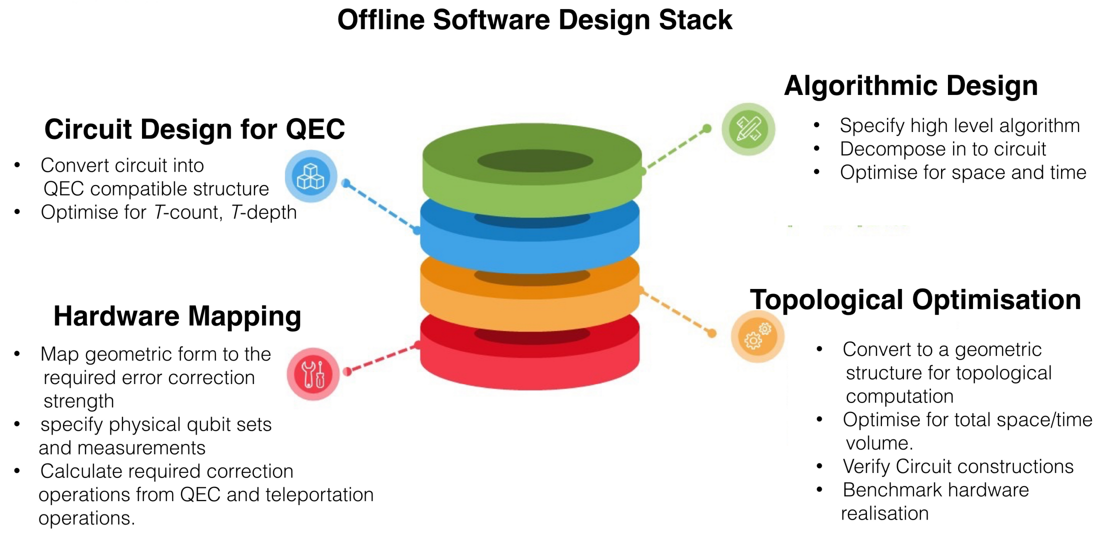

Offline Software Design Stack
Algorithmic Design
• Specify high level algorithm
• Decompose in to circuit
• Optimise for space and time
Circuit Design for QEC
• Convert circuit into QEC compatible structure
• Optimise for T-count, T-depth
Topological Optimisation
• Convert to a geometric structure for topological computation
• Optimise for total space/time volume.
• Verify Circuit constructions
• Benchmark hardware realisation
Hardware Mapping
• Map geometric form to the required error correction strength
• specify physical qubit sets and measurements
• Calculate required correction operations from QEC and teleportation operations.

Figure 1.3 – Different stages of the offline software stack. Image from [2].

computations that can be efficiently simulated even with large problem sizes. For example, the time to simulate the quantum circuit grows quadratically with the size of the circuit, if some (difficult to simulate) gates are not present in the quantum circuit.

For example, IBM has a standard state vector simulator that “only” goes up to 32 qubits. To be able to simulate larger systems, it recently developed new simulation methods based on the stabilizer formalism [64] and demonstrated them by simulating quantum algorithms with 40-50 qubits without resorting to high-performance computers [65].

Alternative approaches to check the correctness of a quantum algorithm and to verify some specific properties is to use the compiler to annotate known properties of the quantum state at different levels of abstraction and checking such annotations using formal verification, e.g., Satisfiability Modulo Theory (SMT) solvers [66].

1.4 Quantum compilers

Quantum compilers have the crucial role of bridging advances in quantum algorithm theory with the ones achieved in fabrication technology. They are responsible of analyzing and optimizing the code while mapping it into native hardware operations. The input of the process is the quantum circuit high-level description and the output is a set of mapped quantum gates. It is important to note that quantum gates are a way to describe quantum operations performed on the qubits. They indeed describe a quantum program, i.e., a sequence of instructions, and not an actual physical implementation as, e.g., an Application Specific Integrated Circuit (ASIC) design.

It is possible to distinguish between static compilation and dynamic compilation, depending on if it is part of the offline or the online software stack. Dynamic compilation deals with information measured at runtime. This includes rotation operations with angles depending on measured data [67] and control measurements to verify hardware physical conditions at runtime. Static quantum compilation, instead, is performed before execution. It has several advantages with respect to classical compilation [68]. The first advantage is that more detailed information offered by quantum computing programming models allows us to perform advanced optimization. In addition, quantum systems enable much more parallelism than classical hardware. Another difference with respect to classical compilation is that the quantum compiler requires to know the problem size prior to compilation. The compiler optimization of quantum programs is crucial as quantum systems suffer from severe resource constraints. Indeed, given the high optimization level required, it is customary to repeat the compilation for different input sizes. On the other hand, such aggressive optimization requires substantial computation resources. It follows that one crucial challenge for quantum compilers is to being capable of tackling large-scale algorithms.

As introduced in the previous section, there are several programming languages that have been developed to program quantum computers. Among all, popular ones are IBM's Qiskit [56] and Microsoft's Q# [55]. Both languages are equipped with quantum compilers targeting real and simulated quantum computers. Obviously, the Qiskit compiler targets IBM's superconducting systems through the IBM Q Experience, while the Q# compiler target hardware from partner companies such as Honeywell, IonQ, and Quantum Circuits, Inc., through Azure Quantum. Other companies are developing software solutions to bridge advances in NISQ technology, e.g., Zapata Computing and Cambridge Quantum Computing (CQC).

1.5 Thesis contribution

In this thesis, I develop algorithms and methodologies for the static compilation of quantum algorithms. In particular, my research focuses on two main problems in quantum compilation: (1) automatic compilation of quantum circuits implementing Boolean functions (commonly called *oracles*), including quantum memory management, and (2) automatic accuracy management in quantum programs.

1.5.1 Compilation of combinational logic

Several quantum algorithms have been proposed, and are currently being researched to exploit the computational capabilities of the future, large-scale quantum computing systems. Applications span, among others: cryptography [69, 70], quantum chemistry [71, 72, 73], material science [74], machine learning [75], satisfiability solving [76], and algorithms for quantum linear systems [77, 78] and prime factorization [18].

Many quantum algorithms require the computation of some *combinational logic functions*, e.g., arithmetic functions, which usually need large amounts of resources to be computed.

Methods that are capable of generating quantum circuits for such logic designs are needed to run these algorithms on a quantum computer. For example, *Harrow-Hassidim-Lloyd (HHL)* requires the reciprocal operation, which causes a significant overhead in the number of qubits with respect to the other components of the algorithm. In some cases, the resources required to perform logic operations may dominate the overall resources and exceed the available computing power.

Quantum circuits performing combinational logic are also required in post-quantum cryptography. It has been shown how Grover's algorithm can be used to break symmetric encryption schemes such as the Advanced Encryption Standard (AES), if the quantum circuit for the encryption function is known [79, 80]. The number of resources required to break a newly proposed post-quantum encryption scheme depends on the resources required to build the required quantum oracle. Consider for example the categories for public-key schemes proposed by the National Institute of Standards and Technology (NIST) in their proposal to standardize post-quantum cryptography [81]. Shor's algorithm also requires combinational logic and can be used to construct quantum algorithms for integer factorization, finite field discrete logarithms, and elliptic curve discrete logarithms. As a consequence, cryptosystems based on these problems cannot be considered secure in a post-quantum environment.

Even if the technology is nowadays still far from achieving the system sizes and performances that these applications require, estimating the resources needed to perform combinational functions has a relevant impact on the design and applicability of advanced quantum algorithms. The resource footprint of these operations, e.g., a large number of quantum operations and qubits, can exceed the actual resources available, hence preventing some algorithms to be computed on constrained quantum hardware. Consequently, there is a large interest in compilation methods that minimize the impact of combinational logic on the cost of quantum algorithms.

Several research works focus on improving (often manually) quantum implementations of cryptographic functions. As Shor's algorithm can be used to break elliptic curve cryptography, authors of [82] have optimized the required quantum circuit that computes the costly elliptic curve scalar multiplication. In [83], authors present resource estimations of quantum preimage attacks on SHA-2 and SHA-3 based on Grover's algorithm. They present quantum oracles for SHA-256 and SHA3-256. They improve the reversible implementations derived in [84] and evaluate the cost of running the attack on a surface code based fault-tolerant quantum computer. In [85] authors focus on improving the implementation of the S-box of AES to simplify Grover based key search. Similarly, authors in [86] provide implementations for SHA-256 and AES-128, results successively improved by [80].

Compiling logic functions into quantum circuits is related to the classical problem of synthesizing electronic circuits from a description of their behavior given in a hardware description language. In classical logic synthesis [87], the high-level description of the Boolean function is parsed into a scalable representation, which is optimized for the application-specific cost functions (typically delay and area) and then the logic is mapped into a library of gates. Compared to this process, compiling quantum oracles imply the additional step of embedding the possibly non-reversible Boolean function into a reversible intermediate representation, which can be implemented using quantum gates. In addition, typical cost functions considered in quantum compilation are related to the quantum circuit width (number of qubits) and depth (number of quantum gates), the latter being constrained by the quantum state's coherence time. Another difference is that, while an electrical circuit is a fabricated object, a quantum circuit is a representation of a set of instructions performed on the quantum registers.

Data structures and optimization methods from classical logic synthesis can be borrowed and adapted to the synthesis of quantum oracles. This is generally performed in two steps. The first one consists of generating a reversible network, to comply with the native reversibility of quantum operations. In this step, the function is transformed into a garbage-free reversible circuit figuring universal reversible gates. The most common examples are the Toffoli and the Fredkin gates [88]. Using one of these two gates is enough to represent any reversible Boolean function. The second step consists of mapping each reversible gate using the native quantum operations of the targeted hardware. Often this step has to start with transformations allowing an easier mapping. Typical examples are the transformations presented in [89, 90] to decompose large reversible gates.

Many automatic methods to synthesize reversible networks exist in literature. Some methods take reversible functions as input, so they require the preliminary step of embedding a possibly non-reversible function into a permutation. The Boolean function representation required by each method also impacts its scalability. Indeed, some representations do not scale efficiently with respect to the number of inputs, making the methods based on them ill-suited to synthesize large networks. For example, the method in [91], referred to as *Transformation-Based Synthesis (TBS)*, requires to represent Boolean functions using *truth tables*—a representation that grows exponentially in the number of inputs. Other methods are based on the Reed-Muller spectra [92], Boolean satisfiability [93] and decision diagrams [94]. Decomposition-Based Synthesis (DBS) [95] uses Young-subgroup based reversible synthesis [96] to compile quantum state permutations into quantum circuit. Finally, the method proposed in [97] is based on *Exclusive Sum-Of-Products (ESOP)* decomposition.

There are also methods for the synthesis of programmable networks that can be used to synthesize oracles without passing through the intermediate reversible network, e.g., [98, 99, 100]. They end up only being applicable to relatively small logic designs, as the generated number of gates grows exponentially with the number of inputs. More scalable methods are based on symbolic representation [101, 102], and hierarchical methods [103, 104, 105, 106], which proved to be applicable to large designs being based on multi-level logic representations.

Once quantum circuits are expressed with respect to a native set of quantum operations, optimization methods can be applied to reduce the used resources. At the quantum circuit level, optimization techniques may rely on the specific properties of the selected gate set.

In [107] authors propose heuristic methods to minimize the number of qubits and gates in large-scale designs. Relevant works on optimization have focused on reducing the number of expensive quantum gates [108, 109]

Specific contributions

This thesis concerns hierarchical methods for the compilation of quantum circuits implementing Boolean functions, with a focus on leveraging the capabilities of less scalable compilation techniques using specifically-designed decomposition methods. The presented algorithms are inspired by methods currently applied in classical logic synthesis—a 30 years old research field focused on optimization and mapping of combinational designs [87]. The algorithms focus on exploring the trade-off between width and depth in quantum circuit compilation. On this topic, I show how the problem of managing the quantum memory can be cast as a satisfiability problem and, as such, can be solved using conventional SAT solvers. The thesis also deals with the problem of decomposing functionally-controlled reversible gates, also called single-target gates, into quantum circuits. A study of the applicability of exact and heuristic advanced ESOP synthesis techniques is presented. In addition, the thesis presents a database of optimal circuits that could be used in tandem with the described hierarchical methods.

1.5.2 Accuracy management

Various modern quantum programming languages provide some support for resource estimation. That is, given a completely specified quantum program, they estimate the number of quantum operations and qubits necessary to run it on a quantum computer. Examples are methods embedded in Q#, ProjectQ, and Quipper [110], which are based on circuit-description languages, and QuRE, which is capable of evaluating different technologies and error-correcting codes. Nevertheless, the provided support does not suffice to reduce the (still significant) amount of manual work involved in resource estimation [111, 112]—one reason being the lack of built-in support for handling approximation errors.

Approximation errors occur in most of the decomposition steps characterizing the compilation of a quantum program. One relevant example is the approximated decomposition of arbitrary rotation gates into a fault-tolerant library. Indeed, because of the discrete nature of any fault-tolerant set [64], such approximation cannot be avoided. When a rotation gate is mapped into this native fault-tolerant set, it is possible to reduce the approximation by increasing the number of obtained operations. This is a trade-off that the designer has to keep into account when building a quantum algorithm with a desired upper-bound on the overall approximation. With the current frameworks, the designer will have to test different settings of decomposition parameters, and for each one synthesize the quantum circuit, estimate the available resources and eventually find the best parameter setting. As an alternative, it is possible to manually derive the mathematical expressions of how the total error and the resources are affected by the algorithm parameters.

In the related work by [113], a theoretical framework is presented to reason about the robustness of quantum programs when executed on noisy quantum hardware. Specifically, the authors develop a logic that enables them to characterize the distance between an ideal quantum program and its noisy counterpart, given a noise model of the target hardware. Several case studies consisting of small quantum circuits (between 1 and 6 qubits) are presented in [113]. Computing the (Q,λ)-diamond norm, which is used to measure the distance between the ideal and the erroneous program, involves solving a semidefinite programming problem (as is the case for the regular diamond norm [114]). This becomes computationally intractable for large systems (and their corresponding noise models) due to the exponential scaling of the dimension with the number of qubits.

Specific contributions

In this thesis, I propose a methodology to manage approximation errors automatically during the compilation of the quantum program. It considers approximation errors that occur at the algorithmic level, which may be decreased by using more quantum resources. The methodology provides language support for expressing accuracy parameters and automatically solves (during compilation) the optimization problem of finding values for such parameters that minimize the total error while not exceeding a given bound on the resources or vice versa. This approach requires fast resource estimation methods to achieve scalability. As a consequence, the thesis proposes to achieve symbolic resource estimation by exploiting compiler optimization passes. A prototype implementation in *Low-Level Virtual Machine (LLVM)* is presented together with the description of how the methodology can be integrated into the compiler of any existing quantum programming language.

1.6 Thesis organization

The thesis is organized as follows:

Chaper 2 introduces the field of classical logic synthesis. It defines Boolean functions and their representations. Then, it discusses how it is possible to classify Boolean functions into equivalent classes. This chapter also presents a method that is used in classical logic synthesis to perform multi-level logic optimization and technology mapping, i.e., *Look-Up Table (LUT)* mapping. Finally, satisfiability problems are defined.

Chapter 3 provides all the background notions on quantum computing that are necessary to understand the remainder of the thesis. After a digression on basic quantum computing concepts as qubits, gates, and measurement, this chapter contains a description of the *phase polynomial representation* of quantum circuits.

Chapter 4 introduces reversible circuits and the existing methods to decompose them into quantum gates. This chapter concludes the background part of this thesis.

Chapter 5 introduces and analyses the k-LUT-based hierarchical synthesis of reversible networks. The chapter details the way this thesis contributes to improving the state-of-the-art hierarchical method. In particular, in this thesis I present quantum memory management and a novel quantum-aware LUT decomposition method. The result section shows improved results with respect to state-of-the-art.

Chapter 6 introduces novel synthesis algorithms that perform hierarchical synthesis of Xor-And-inverter Graph (XAG), a specific type of k-LUT networks. The result section reports the statistics of compiled quantum circuits for many publicly available benchmarks with application in cryptography and fault-tolerant quantum computing.

Chapter 7 elaborates on the problem of decomposing reversible gates into quantum circuits. For small Boolean functions, it is possible to store pre-computed optimal results in a database compressed using spectral classification. This chapter also presents a study on the application of advanced ESOP synthesis methods to this decomposition problem. In particular, it presents a SAT-based synthesis method that minimizes to define arbitrary cost functions. Finally, it explores methods that can be used to perform such decomposition with the support of a fixed number of helper qubits.

Chapter 8 discusses two different methods for the optimization of quantum circuits. The first one aims to reduce the number of T gates of quantum circuits derived from the decomposition of reversible networks. The method includes solving instances of the maximum weight graph matching problem. The second technique aims to reduce the number of two-qubit gates by rewriting parts of the original network exploiting an exact SAT-based synthesis method.

Chapter 9 describes the first framework with the ability to automatically manage approximation errors and generating (near-)symbolic resource estimates. The methodology integrates language support for approximation errors to facilitate the task of programming a quantum algorithm in an approximation-aware fashion. Besides, it automates the process of optimizing accuracy parameters at the compiler level. The chapter lists the features that a quantum programming language and its compiler must support in order to enable the proposed methodology and demonstrates integration into the Q# compiler. Finally, it demonstrates that a symbolic resource estimation is the only applicable strategy to solve the optimization problem for large-scale examples.

Chapter 10 presents an open-source C++ library dedicated to the quantum compilation of logic networks and to quantum memory management. By using this library, called *caterpillar* it is possible to experiment with many of the compilation algorithms presented in this manuscript.

Background Part I

2 Logic synthesis

Modern processors are designed, tested, and fabricated with the support of *Electronic Design Automation (EDA)* tools. Hardware components are initially specified using a high-level description language. Then, they are automatically compiled into the *Register Transfer Level (RTL)* abstraction by a process called *High Level Synthesis (HLS)*. An RTL description is typically given in a hardware description language such as Verilog or VHDL. These languages can describe the behavior of a component in terms of sequential and combinational logic. The RTL specification is the starting point of the *logic synthesis* process, which generates a logic gate netlist and optimizes it in terms of the required cost functions. Commonly used cost functions are area, delay, and power consumption. The final step of the entire process, schematized in Fig. 2.1, is the physical design. This last process outputs the *Integrated Circuit (IC)* layout, providing the geometrical instructions necessary to manufacture a chip.

Logic synthesis is the process of generating and optimizing a gate netlist, starting from an RTL description. In this chapter, I focus on the techniques that can be used to represent and synthesize combinational logic (implementing Boolean functions).

2.1 Logic representations

2.1.1 Boolean functions

This section defines Boolean functions and their properties.

Definition 2.1.1 *A function $f : \mathbb{B}^n \to \mathbb{B}$ is called a n-input Boolean function, where $\mathbb{B} = \{0, 1\}$ is the two-element finite field with addition defined by the logical exclusive-OR (XOR) operation and multiplication defined by the logical AND operation.*

Definition 2.1.2 *A Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ is* reversible *if and only if f is a bijection, i.e., $n = m$ and f performs a permutation of the set of input patterns.*

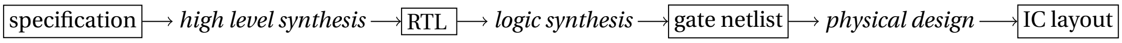


Figure 2.1 – Electronic Design Automation (EDA) flow.

Definition 2.1.3 *A Boolean function* $f : \mathbb{B}^n \to \mathbb{B}$ *is* linear *if and only if:*

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

Any linear Boolean function can be written as

$$f(x_1, \dots, x_n) = a_1 x_1 \oplus a_2 x_2 \oplus \cdots \oplus a_n x_n \tag{2.1}$$

for constants $a_1, \dots, a_n \in \mathbb{B}$. Given this notation, one can write any linear function f as a row vector of the n constant Boolean coefficients: $(a_1 \dots a_n)$.

Example 2.1.1 *Given the linear Boolean function* $f(x_1, x_2, x_3) = x_2 \oplus x_3$ *it corresponds to the row vector:* $(0\ 1\ 1)$.

Definition 2.1.4 *A multi-output Boolean function* $f : \mathbb{B}^n \to \mathbb{B}^m$ *is* linear *if and only if each component function* f_i *is linear, for* $i = 1, \dots, m$.

A multi-output linear Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ can be represented using a $m \times n$ matrix, in which each row is the row vector representing a component linear function f_i. If the multi-output function is linear and reversible the representative matrix is a non-singular matrix $n \times n$.

2.1.2 Truth tables

A Boolean function can be represented by its truth table, which is a table that shows the output values corresponding to each input assignment. Such ordered output values correspond to a bitstring $b_{2^n-1} b_{2^n-2} \dots b_0$ of size 2^n where $b_x = f(x_1, \dots, x_n)$ when $x = (x_1 x_2 \dots x_n)_2$. For large functions, it is convenient to use a hexadecimal encoding of the bitstring.

Example 2.1.2 *The truth table of the majority-of-three function* $\langle x_1 x_2 x_3 \rangle = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$, *where* $\vee$ *represents the OR operation and* $\wedge$ *represents the AND operation, is 1110 1000 or*$^{\#}$ *e8 in hexadecimal encoding.*

2.1.3 ESOP representation of Boolean function

Every Boolean function can be represented in terms of an Exclusive Sum-Of-Products (ESOP) expression.

Definition 2.1.5 *An ESOP over n Boolean variables, $x_1, \dots, x_n \in \mathbb{B}$, is an expression of form $t_1 \oplus \cdots \oplus t_k$, where each $t_i = l_{i,1} \cdots l_{i,l_i}$ is a product term (or cube) of literals $l_{i,j} \in \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ for $1 \le i \le k$ and $1 \le j \le l_i$. The symbol $\oplus$ denotes the XOR operation, and $\bar{x}_i$ denotes the negated Boolean variable x_i for $1 \le i \le n$.*

Example 2.1.3 *An ESOP for the majority-of-three function $\langle x_1 x_2 x_3 \rangle$ is $x_1 x_2 \oplus x_1 x_3 \oplus x_2 x_3$.*

An ESOP expression can be interpreted as a two-level logic circuit realizing the Boolean function $f(x_1, \dots, x_n) = t_1 \oplus \cdots \oplus t_k$. This representation is not unique, hence many heuristic and exact minimization methods have been proposed [115, 116, 117, 118].

2.1.4 Multi-level logic networks

Multi-level logic networks are scalable representations of Boolean functions. A logic network is represented by a graph in which each node performs a Boolean operation and edges define data dependencies. The inputs of the function are the primary inputs of the graph. Networks are characterized by their size, i.e., the number of nodes, and by their depth, i.e., the number of levels in the graph. Two nodes are in the same level if they have the same maximum distance from the primary inputs. For internal nodes, the indegree and outdegree are referred to as *fan-in* and *fan-out*, respectively. The *transitive fan-in cone* of a node n is the set of all the nodes for which there is a path between the primary inputs and n.

According to the characteristics of the network, it is possible to define different graph representations. For example, the *And-Inverter Graph (AIG)* is a network widely used in logic synthesis and verification, by both academic and industrial tools (see e.g., [119, 120]). In this thesis, I present algorithms that apply to different logic networks:

- *And-Inverter Graph (AIG)*, with nodes implementing the 2-input AND and inversion;
- *Majority-Inverter Graph (MIG)*[121], with nodes implementing the 3-input majority function (MAJ) and invertion;
- *Xor-And-inverter Graph (XAG)*, with nodes implementing the 2-input XOR, the 2-input AND and inversion;
- *Xor-Majority-inverter Graph (XMG)*, with nodes implementing the 2-input XOR, the 3-input MAJ and inversion;
- *Look-Up Table (LUT)*, with nodes implementing arbitrary Boolean functions.

Please note that the choice of the best network representation depends on the specific algorithm. Synthesis algorithms for post-silicon technologies exploit networks that feature the technology's native logic operation [122]. For example, *Quantum Cellular Automata (QCA)*

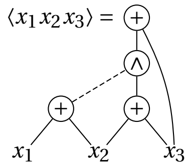


Figure 2.2 – An XAG for the majority of 3-inputs.

or superconducting quantum-flux circuits naturally implement the majority operation, as a consequence, the MIG network is usually preferred. Programs to be run on *Resistive Random Access Machine (RRAM)* should also be expressed using the majority operation [123].

When developing algorithms for the synthesis of reversible and quantum circuits, networks based on the reversible XOR operation are preferred, such as the XAG. Fig. 2.2 shows the XAG network for the majority-of-three function $^{\#}$e8, where dashed edges represent inversion.

2.2 Classification of Boolean functions

Classification of Boolean functions is the process of partitioning all functions with a specified number of inputs into equivalence classes. Every function inside a given class can be transformed into any other function in the same class by a specific set of transformations, called *invariant operations*. Different transformation sets define different classification methods. Example applications of Boolean classification in logic synthesis are network rewriting [124] and hierarchical reversible synthesis [125, 126].

A popular classification is the so-called Negation-Permutation-Negation (NPN) classification. The allowed invariant operations are: swapping two variables, complementing a variable and complementing the function [127, 128, 129]. There are 2^{2^n} n-input Boolean functions. By means of these transformation the Boolean functions with 1, 2, 3, 4 and 5 inputs are grouped into 2, 4, 14, 222 and 616 126 NPN equivalent classes.

Another very popular classification method is the spectral classification, described in detail later in this section. The operations that are used to partition Boolean functions into spectral equivalence classes are defined. Please note that the NPN invariant operations are a subset of the spectral invariant operations.

Definition 2.2.1 (Spectral invariant operations [130])

1. Swapping two variables. *One obtains* $g = f(x_1,\dots,x_j,\dots,x_i,\dots,x_n)$ *from* $f(x_1,\dots,x_i,\dots,x_j,\dots,x_n)$ *by swapping variables* x_i *and* x_j*. We denote this operation as* $f \xrightarrow{x_i \leftrightarrow x_j} g$.
2. Complementing a variable. *One obtains* $g = f(x_1,\dots,\bar{x}_i,\dots,x_n)$ *from* $f(x_1,\dots,x_i,\dots,x_n)$ *by complementing variable* x_i*. We denote this operation as* $f \xrightarrow{\bar{x}_i} g$.
3. Complementing the function. *One obtains* $g = \bar{f}$ *from* f *by complementing the whole*

function. We denote this operation as $f \xrightarrow{\neg} g$.

4. Translational operation. *One obtains* $g = f(x_1,\dots,x_i \oplus x_j,\dots,x_n)$ *from* $f(x_1,\dots,x_i,\dots,x_n)$ *by replacing* x_i *with* $x_i \oplus x_j$. *We denote this operation as* $f \xrightarrow{x_i \oplus x_j} g$.

5. Disjoint translational operation. *One obtains* $g = x_i \oplus f$ *from* f *by XOR-ing it with input* x_i. *We denote this operation as* $f \xrightarrow{\oplus x_i} g$.

These operations partition all n-input Boolean functions into equivalence classes by means of the following equivalence relation.

Definition 2.2.2 (Spectral equivalence [131]) *We say that two* n*-input Boolean functions* f *and* g *are* spectral-equivalent *if there exist operations* $o_1,\dots,o_k$ *from Definition 2.2.1 such that*

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} g.$$

One can readily verify that spectral equivalence is an equivalence relation. In the remainder, I write $f \doteq g$, if f is spectrally equivalent to g. Further, I refer to the equivalence class of f as $[f] = \{g \mid f \doteq g\}$.

Example 2.2.1 *It is possible to show that* $\langle x_1x_2x_3\rangle \doteq x_1 \wedge x_2$, *where* $x_1 \wedge x_2$ *is a 3-input Boolean function in which the third variable* x_3 *is a don't care input.*

$$\begin{aligned} & x_1 \wedge x_2 \xrightarrow{\bar{x}_2} x_1 \wedge \bar{x}_2 \xrightarrow{x_2 \oplus x_3} x_1 \wedge (\bar{x}_2 \oplus x_3) \xrightarrow{x_1 \oplus x_2} \\ & \qquad (x_1 \oplus x_2) \wedge (\bar{x}_2 \oplus x_3) = x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 \xrightarrow{\oplus x_1} \\ & \qquad\qquad x_1 \oplus x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 = x_1x_2 \oplus x_1x_3 \oplus x_2x_3 = \langle x_1x_2x_3\rangle \end{aligned}$$

Using this equivalence relation the set of all n-input Boolean functions for $n = 1,2,3,4,5,6$ collapses into just $1,2,3,8,48,150\,357$ equivalence classes [132, 133].

Spectral techniques can be used to determine to which equivalence class a function belongs. For this purpose, the Rademacher-Walsh spectrum and the auto-correlation spectrum are defined.

Definition 2.2.3 *The* Rademacher-Walsh spectrum *of an* n*-input Boolean function* f *is a mapping* $S_f : \mathbb{B}^n \mapsto \mathbb{Z}$ *defined as*

$$S_f(x) = 2^n - \sum_{b \in \mathbb{B}^n} \big(\langle b, x\rangle \oplus f(x)\big) \tag{2.2}$$

where $\langle b, x\rangle$ *is the inner product of* b *and* x, *with* $x \in \mathbb{B}^n$ *(see, e.g., [134]).*

The Rademacher-Walsh spectrum S_f of the function f expressed as a truth table in the $\{1,-1\}$ encoding F can be computed applying the following formula:

$$S_f = H_n F$$

Each coefficient of the spectrum represents the correlation with a parity function of a subset of the inputs. H_n is the Hadamard transform matrix over n variables and is defined as:

$$H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix}, \quad H_0 = 1$$

Each row of the Hadamard transform matrix is equal to the truth table of the parity function between a subset of the n variables. For example, the last row of an n-variable Hadamard matrix will be the truth table of the parity function $x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

Example 2.2.2 *Given the 3-input majority Boolean function $f(x_1, x_2, x_3) = \langle x_1 x_2 x_3 \rangle$, its truth table in the $\{1,-1\}$ encoding is:*

$$F = \begin{pmatrix} 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \end{pmatrix}$$

The Rademacher-Walsh spectrum of f is:

$$S = H_3 F = \begin{pmatrix} 0 & 4 & 4 & 0 & 4 & 0 & 0 & -4 \end{pmatrix}$$

Definition 2.2.4 *The* auto-correlation spectrum *of an n-input Boolean function f is a mapping $B_f : \mathbb{B}^n \mapsto \mathbb{Z}$ and defined as*

$$B_f(x) = 2^n - \sum_{b \in \mathbb{B}^n} \big(f(x \oplus b) \oplus f(x)\big) \tag{2.3}$$

for $x \in \mathbb{B}^n$.

Edwards [130] has shown how to derive a canonical representative $\hat{f} \in [f]$ for some Boolean function f, by applying the operations in Definition 2.2.1 directly to the Rademacher-Walsh spectrum of f (see also [135]). The operations perform sign changes and rearrangements to the spectral coefficients (see [130, 135, 136, 137] for details). The procedure can be efficiently implemented and returns a sequence of operations $o_1, \ldots, o_k$ such that $f \xrightarrow{o_1} \cdots \xrightarrow{o_k} \hat{f}$. The same algorithm can be used to derive a sequence of operations for two functions f_1, f_2 such that $f_2 \in [f_1]$. Note that both functions have the same representative $\hat{f}$ and two sequences of operations are obtained such that $f_1 \xrightarrow{o_1} \cdots \xrightarrow{o_k} \hat{f}$ and $f_2 \xrightarrow{o'_1} \cdots \xrightarrow{o'_l} \hat{f}$. Therefore $f_1 \xrightarrow{o_1} \cdots \xrightarrow{o_k} \hat{f} \xrightarrow{o'_l} \cdots \xrightarrow{o'_1} f_2$, since all operations are self-inverse.

To check whether two functions f and g with up to 4 variables are spectrally equivalent without computing their representatives and order sequences is even simpler. It has been shown that

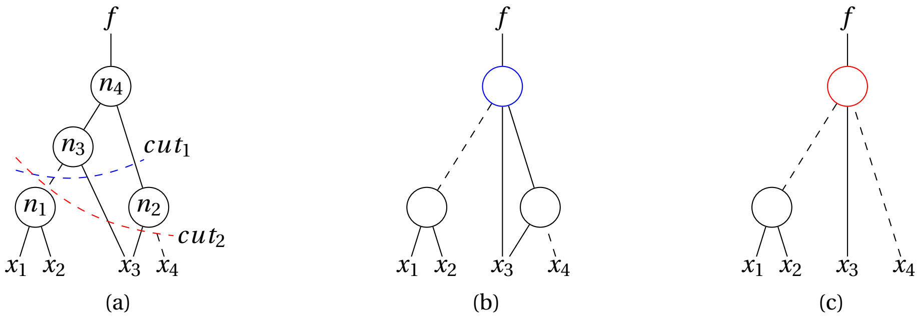


Figure 2.3 – (a) An AIG graph performing the function $f = (\overline{x}_1 + \overline{x}_2)x_3\overline{x}_4$ with two possible 3-feasible cuts; (b) 3-LUT network generated by cut_1; (c) 3-LUT network generated by cut_2.

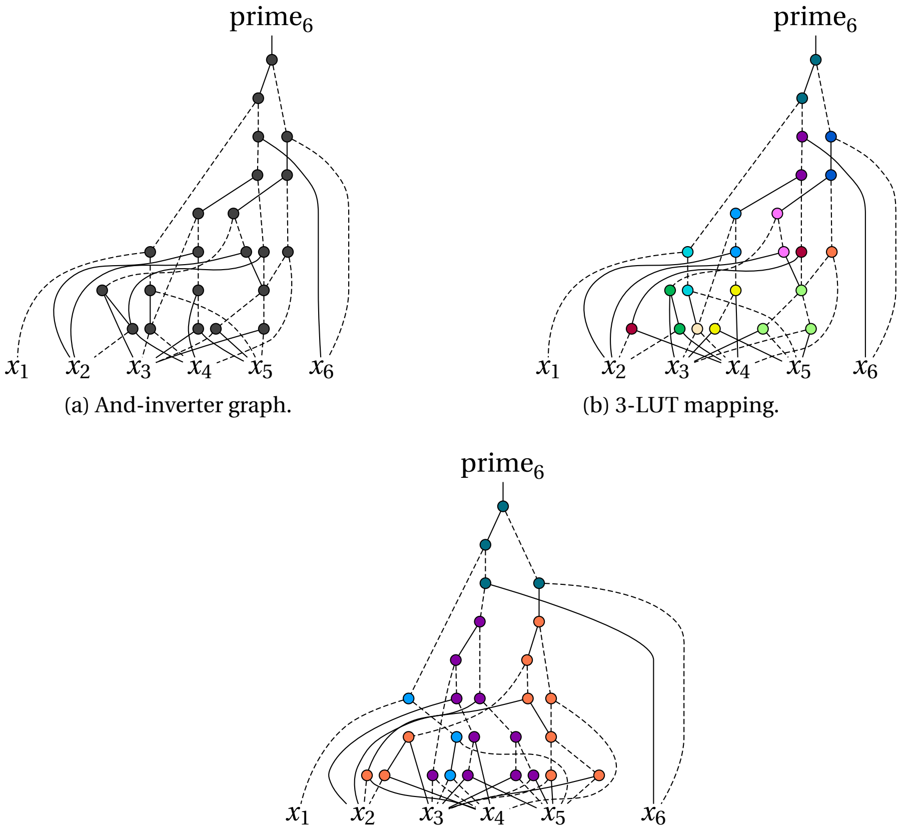
it is sufficient to check whether the distributions of the coefficients in $|S_f|$ and $|S_g|$ are equal, where $|S_f|$ and $|S_g|$ take the element-wise absolute value on S_f and S_g. For functions with up to 5 variables one needs to check whether the distributions of the coefficients in $|B_f|$ and $|B_g|$ are equal as well [134]. For further information on spectral classification the reader is referred to the literature (see, e.g., [138, 139, 140, 137]).

2.3 Look-Up Table (LUT) mapping

Look-up table (LUT) mapping is a method to decompose a network into many single-output sub-networks with an upper bound on the number of inputs of each sub-network. It has originally been used to map logic designs into the components of Field Programmable Gate Array (FPGA) that are capable of computing any Boolean function up to a given number of inputs, i.e., look-up tables. Later, LUT mappers found a successful application in logic synthesis and circuits optimization [124], thanks to their ability to decompose large functionality into smaller components. Several efficient state-of-the-art mappers are available and they are traditionally designed to minimize delay and area of the resulting circuit, e.g., [141, 142, 143].

The input of the LUT mapping is a multi-level logic network representing a Boolean function. The k-LUT mapper decomposes the multi-level network using k-feasible cuts. A cut for a node n is a set of leaves $l_1, \ldots, l_k$ such that each path from n to a primary input includes one of the leaves. A leaf is a node or a primary input of the network. A k-feasible cut is a cut that has at most k leaves.

An AIG representing the function $f = (\overline{x}_1 + \overline{x}_2)x_3\overline{x}_4$ is shown in Fig. 2.3(a). All nodes of this network perform the 2-input AND operation between the node's inputs, dashed edges represent Boolean inversion, and x_1, x_2, x_3 and x_4 are the primary inputs. Fig. 2.3(a) shows two 3-feasible cuts for the node n_4. The first cut has leaves n_1, x_3 and n_2, while the second cut has leaves n_1, x_3 and x_4. Fig. 2.3(b) and (c) show the 3-LUT networks generated by the first and the second cut, respectively. In the first graph, the node highlighted in blue is a LUT with 3 inputs that performs the combined operations of nodes n_4 and n_3, while in the second graph,



(a) And-inverter graph.

(b) 3-LUT mapping.

(c) 4-LUT mapping.

Figure 2.4 – LUT mappings for the function prime_6.

the LUT highlighted in red performs the combined operations of three nodes. Comparing the two networks, generated by two different cuts, it is clear how the choices made during the mapping process affect the number of nodes of the resulting LUT network and the complexity of the function performed by each LUT.

A larger-scale example, with k-LUT mappings generated by several cuts, is shown in Fig. 2.4. The AIG in Fig. 2.4 (a) represents the function $\text{prime}_6(x_1,\ldots,x_6) = [(x_6\ldots x_1)_2 \text{ is prime}]$. Fig. 2.4 (b) and Fig. 2.4 (c) show a 3-LUT and 4-LUT mapping of the AIG, respectively. Note that nodes with the same color belong to the same LUT. There are cases in which nodes of the initial AIG are copied so that they can belong to two different LUTs. The 3-LUT and 4-LUT mappings contain 12 and 4 LUTs, respectively.

2.4 Boolean satisfiability (SAT) problem

Definition 2.4.1 (SAT problem) *Given a Boolean function $f(x_1,...,x_n)$, the Boolean satisfiability problem consists of determining an assignment $\hat{x}$ to the variables $x_1,...,x_n$ such that $f(\hat{x}) = 1$. If such an assignment exists, it is called a satisfying assignment, otherwise the problem* *is unsatisfiable.*

The function f is specified as a Conjunctive Normal Form (CNF): a conjunction of clauses where each clause is a disjunction of literals. A literal is an instance of a variable or its complement. SAT can be summarized as follows: given a list of clauses, determine if there exists a variable assignment that satisfies all of them.

The SAT problem is well known to be an intractable NP-complete problem [144, 145]. Indeed, any other problem in this category can be reduced into a SAT problem, maintaining the number of solutions unchanged. Nevertheless, modern SAT solvers [146, 147, 148] are software programs capable of solving large instances of the problems, characterized by tens of thousands of variables and millions of constraints [149].

In order to use SAT solvers for practical applications, the decision problem to be solved must first be expressed in terms of a SAT formula in CNF. Such an encoding is crucial and can have a significant impact on the overall runtime of the SAT solver. In this thesis, problems arising from various steps of the quantum compilation flow are solved using state-of-the-art solvers, once an encoding of the original domain problem into a SAT formula is defined [150].

2.5 Summary

This chapter introduced classical logic synthesis—a process embedded into Electronic Design Automation (EDA) tools and flows, which deals with logic optimization and technology mapping. In particular, I described Boolean functions and their properties, while introducing various representations, i.e., truth tables, Exclusive Sum-Of-Products (ESOP) expressions, and multi-level logic networks (see Section 2.1). In the remainder of this thesis, Boolean functions will be classified using spectral equivalence classes, as described in Section 2.2. Besides, many of the algorithms presented in this thesis use the described k-LUT mapping—a technique to decompose a graph used in logic optimization and technology mapping. Finally, some of the problems characterizing quantum compilation can be cast as Boolean satisfiability problems, described in Section 2.4, and solved using SAT solvers.

3 Quantum computing

3.1 Quantum computing principles

3.1.1 Qubits

Qubits are the computational units of quantum computers. Thanks to some of their properties, such as *superposition* and *entanglement*, quantum systems are capable of extraordinary computational performances.

Qubits do not only exist in the two classical states $|0\rangle$ and $|1\rangle$, but can be described by a linear combination of these states: $|\phi\rangle = a_0|0\rangle + a_1|1\rangle$, where $a_0, a_1 \in \mathbb{C}$. Each qubit is in a *superposition* of the classical states, which enables parallelism in quantum computation. Measurements destroy this superposition forcing the state to collapse according to the relative probabilities $|a_0|^2$ and $|a_1|^2$. *Entanglement* is a global property of two or more states that cannot be accounted for classically [151]. Thanks to this property, two qubits in a superposition can be correlated with one another: the state of one depends on the state of the other even when separated.

Fig. 3.1 shows a commonly-used 3-dimensional representation of a qubit state, namely the Bloch sphere. Considering the real and the imaginary parts of each complex coefficient a_0 and a_1, the condition $|a_0|^2 + |a_1|^2 = 1$ only leaves three degrees of freedom. The two poles of the sphere represent the two classical states, while all the points on its surface represent superposed states. On the equator of the Bloch sphere there are all the superposed states with $|a_0|^2 = |a_1|^2 = 1/2$ characterized by different angles with respect to the Z-axis.

A 2-qubit system can be defined as:

$$|\phi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle$$

where $a_{00}, a_{01}, a_{10}, a_{11} \in \mathbb{C}$ and $|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$. As a consequence, 4 complex coefficients are needed to represent a two-qubit state, while 8 complex coefficients are necessary to describe a 3-qubit system. In general, to represent the state of n qubits and to

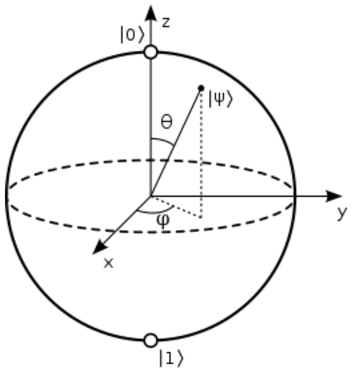


Figure 3.1 – The Bloch sphere.

simulate the quantum system behavior on a classical computer, 2^n complex coefficients are required. The state of a system is often represented using a vector notation, where each entry is the amplitude of the corresponding state $|\phi\rangle = (a_0, a_1, \ldots, a_{2^n})^T$.

3.1.2 Gates

The state of a qubit can be modified by applying quantum gates. Following the vector representation of quantum states, quantum gates can be described as matrices. For example, applying the quantum gate performing inversion X to a qubit in the state $|\phi\rangle = a_0|0\rangle + a_1|1\rangle$ corresponds to the following matrix multiplication:

$$X\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_0 \end{pmatrix}$$

A gate acting on a single qubit can be described by 2×2 matrices, in general, a gate acting on a n-qubit system is described by a $2^n \times 2^n$ matrix. The transformed state must also comply to the normalization condition, which requires $|a_0|^2 + |a_1|^2 = 1$ for a single qubit. It follows that only unitary matrices, i.e., norm-preserving linear transformations, can describe a valid quantum state transformation.

Definition 3.1.1 *A matrix U is unitary if $U^\dagger U = I$, where $U^\dagger$ is the adjoint (complex conjugate) of U, obtained by transposing and then complex conjugating U, and I is the identity matrix.*

3.1.3 Universal quantum libraries

The universal library typically-considered for fault-tolerant quantum computing is the Clifford+T gate library, which consists of the reversible CNOT gate, the Hadamard gate, abbreviated H, as well as the T gate, and its inverse $T^\dagger$.

- *H gate.* The Hadamard gate is often used to create superposition. It transforms a $|0\rangle$ into a $(|0\rangle + |1\rangle)/\sqrt{2}$ and a $|1\rangle$ into a $(|0\rangle - |1\rangle)/\sqrt{2}$. The resulting state is halfway between $|0\rangle$ and $|1\rangle$ and collapses into one of these classical states with 50% probability. For example, given n qubits initialized to $|0\rangle$, if a H gate is applied to each of them, the following state is obtained:

$$\frac{1}{\sqrt[n]{2}}(|00\ldots0\rangle + |00\ldots1\rangle + \cdots + |11\ldots0\rangle + |11\ldots1\rangle)$$

This means that the system is in all the possible input combinations at the same time, with the same probability $1/\sqrt[n]{2}$.

- *controlled-NOT (CNOT).* This operation is the only 2-qubit operation in the library. It complements the state of one qubit called *target* accordingly to the state of the other qubit called *control.* It performs the mappings: $|00\rangle \mapsto |00\rangle, |01\rangle \mapsto |01\rangle, |10\rangle \mapsto |11\rangle, |11\rangle \mapsto |10\rangle$.

- *T gate.* This gate leaves the basis state $|0\rangle$ unchanged and maps $|1\rangle$ to $e^{i\pi/4}|1\rangle$. It does not belong to the Clifford library and it is necessary to achieve universality. This means that adding this gate to the Clifford ones makes it possible to approximate any unitary matrix with arbitrary precision.

The unitary matrices of the Clifford+T gates are:

$$\text{CNOT} = \begin{pmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&0&1\\ 0&0&1&0 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1&1\\ 1&-1 \end{pmatrix}, \quad T = \begin{pmatrix} 1&0\\ 0&e^{\mathrm{i}\pi/4} \end{pmatrix}.$$

In fault-tolerant quantum computing, each logical qubit is encoded into several physical qubits using error-correcting codes. Operations must be performed on the encoded state, which may require several operations on the physical qubits. Clifford gates can be implemented using a reasonable amount of resources, for example the popular surface code comes with a set of efficient generators for the Clifford group [152]. Nevertheless, non-Clifford gates require both *teleportation*, which includes measurements, and *state distillation* [153], which requires many operations on many dedicated qubits. It follows that the number of T gate (T-count) is evaluated to determine the implementation cost of a quantum algorithm implemented with the Clifford+T library. In addition, fault-tolerant computations can be performed in time proportional to one round of measurement per layer of T gates (or T-stage), as demonstrated in [154]. As a consequence, the number of T-stages (called T-depth) in a Clifford+T circuit is another relevant cost function.

IBM's quantum computers, which are examples of Noisy Intermediate-Scale Quantum (NISQ) systems, natively support the U gate, $U(\theta,\psi,\lambda) = R_x(\psi)R_y(\theta)R_z(\lambda)$, which is parameterized over 3 continuous variables, and the CNOT gate. Usually, 2-qubit gates are more error-prone than single qubit gates. For this reason, a good measure for the cost of a quantum circuit synthesized for NISQ machines is the number of CNOT gates. The same is also valid for the Controlled-Z gate that is used, e.g., in Rigetti's NISQ systems.

For more details on quantum gates the reader is referred to [64].

3.1.4 Measurements

Quantum measurements are described by a collection $\{M_m\}$ of measurement operators [64]. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\phi\rangle$, the probability that the measurement has outcome m is:

$$p(m) = \langle\phi|M_m^\dagger M_m|\phi\rangle$$

After measurement the state of the system is:

$$\frac{M_m|\phi\rangle}{\sqrt{\langle\phi|M_m^\dagger M_m|\phi\rangle}}$$

The measurement operators satisfy the completeness equation, which reflects the fact that the sum of probabilities is one:

$$\sum_m M_m^\dagger M_m = I$$

Example 3.1.1 *Consider a measurement of a qubit in the computational basis. The two measurement operators are :* $M_0 = |0\rangle\langle 0|$, $M_1 = |1\rangle\langle 1|$. *The probability of obtaining measurement outcome* $|0\rangle$ *for the initial state* $|\phi\rangle = a_0|0\rangle + a_1|1\rangle$ *is*

$$p(0) = \langle\phi|M_0^\dagger M_0|\phi\rangle = \langle\phi|M_0|\phi\rangle = |a_0|^2$$

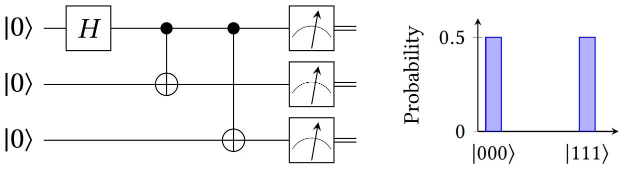
3.1.5 Quantum circuits

The Quantum Random Access Machine (QRAM), described by the computational model in Section 1.3.1, is capable of executing quantum instructions. A sequence of quantum instructions can be visualized using circuit diagrams, so-called *quantum circuits*.

An example quantum circuit is shown in Fig. 3.2. This circuit generates a Greenberger–Horne–Zeilinger state on three qubits, i.e., the state

$$\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle).$$

In a quantum circuit, each qubit is represented by a horizontal line. Operations are denoted by boxes or other symbols on the qubit(s) they are being applied to. Time advances from left to right. The first operation is a Hadamard gate (H) applied to the first qubit, which maps $|000\rangle$ to $\frac{1}{\sqrt{2}}(|000\rangle + |100\rangle)$. The next gate is a controlled-NOT or CNOT, which entangles the first and the second qubit by flipping the latter if the first qubit is $|1\rangle$. After the two CNOT gates, the



(a)

(b)

Figure 3.2 – (a) A quantum circuit computing an entangled state and (b) the corresponding expected measurement outcomes.

three qubits are in the state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$. Finally, all three qubits are measured. There is a 50% probability of measuring all 0s and a 50% probability of measuring all 1s.

A particular type of quantum circuits are the so-called *quantum oracles,* which for example are included into Grover's algorithms. Grover's oracles perform single-output Boolean functions. Here, the term is extended to quantum circuits implementing multi-output Boolean functions.

Definition 3.1.2 *A quantum oracle is defined as a "black box" unitary operation performing a multi-output Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$. The effect of an oracle O_f performing the function f over two registers (one of n qubits to store the inputs, $|x\rangle$, and one of m qubits to store the outputs, $|y\rangle$) can be described as follows:*

$$O_f : |x\rangle \otimes |y\rangle \otimes |0\rangle^l \mapsto |x\rangle \otimes |y \oplus f(x)\rangle \otimes |0\rangle^l$$

were $|0\rangle^l$ is a register of l qubits, which are internally used by the oracle and restored to their initial state. These qubits are called helper qubits. *Helper qubits initialized to $|0\rangle$ are referred to as* clean.

The implementation cost of a quantum circuit depends on the number of qubits required for the computation, and on the number of gates. Automatic tools can be used to synthesize quantum circuit with a reduced implementation cost to take into account technology constraints.

3.2 Phase polynomial representation

This section introduces a useful representation of quantum circuits that only consists of CNOT and T gates. A {CNOT, T} n-qubit quantum circuit implements a unitary matrix U that can be expressed using a linear reversible function g and a polynomial $p(x_1, \ldots, x_n)$ defining a diagonal phase transformation. This circuit description is called *phase polynomial representation* [108] and more than one {CNOT, T} circuit can share the same representation.

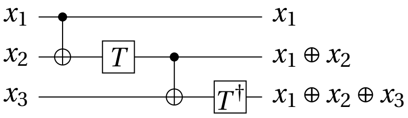


Figure 3.3 – A {CNOT, T} circuit.

Lemma 3.2.1 *The action of a {CNOT, T} circuit on the initial states $|x_1,\ldots,x_n\rangle$ has the form:*

$$|x_1,\ldots,x_n\rangle \mapsto e^{\frac{\pi}{4}\mathrm{i}p(x_1,\ldots,x_n)}|g(x_1,\ldots,x_n)\rangle,$$

$$\textit{with } p(x_1,\ldots,x_n) = \sum_{i=1}^{l}(c_i \bmod 8) f_i(x_1,\ldots,x_n), \quad (3.1)$$

where $g : \mathbb{B}^n \to \mathbb{B}^n$ is a linear reversible function, p is a linear combination of linear Boolean functions $f_i : \mathbb{B}^n \to \mathbb{B}$ with the coefficients reduced modulo 8. The coefficients $c_i \in \mathbb{Z}$ measure the number of rotations of $\pi/4$ that are applied to the corresponding f_i, e.g., each T gate gives a $\pi/4$ rotation ($c_i = 1$), an S gate gives a $\pi/2$ rotation ($c_i = 2$), a $T^\dagger$ gate gives a $7\pi/4$ rotation ($c_i = 7$). The phase polynomial notation is uniquely specified by:

$$g, f_i, c_i \textit{ for } i = 1,\ldots,l$$

where l is the number of phase gates.

Example 3.2.1 *The circuit in Fig. 3.3 implements a transformation on the input qubit states x_1, x_2, x_3 characterized by the linear reversible function*

$$g : |x_1\rangle|x_2\rangle|x_3\rangle \mapsto |x_1\rangle|x_1 \oplus x_2\rangle|x_1 \oplus x_2 \oplus x_3\rangle$$

and by a phase polynomial

$$p(x_1, x_2, x_3) = 1(x_1 \oplus x_2) + 7(x_1 \oplus x_2 \oplus x_3)$$

$$\textit{with } c_1 = 1, c_2 = 7, f_1 = x_1 \oplus x_2, f_2 = x_1 \oplus x_2 \oplus x_3.$$

The T gate gives a phase of $\pi/4$ while its complex conjugate $T^\dagger$ gives a phase of $7\pi/4$.

3.3 Summary

This chapter introduced some basic concepts of quantum computing needed in the remainder of this thesis. In particular, I defined qubits and their possible states in Section 3.1, as well as superposition and entanglement. Then, I described how such states can be modified using unitary gates to perform computation. The gates belong to a universal quantum library, which enables every possible unitary computation. In fault-tolerant quantum computing, the selected library consists of the Clifford+T discrete set, described in Section 3.1.3. When error-correcting codes are used to generate logic qubits, the T gate is more expensive to be executed when compared to the other gates of the library. For this reason, the methods for quantum compilation developed in the context of this thesis aim to minimize the number of T gates, or T-count, and the number of T-stages, also called T-depth. This chapter also illustrated the *phase polynomial representation* (Section 3.2), which is used to describe a category of quantum circuits that only consists of CNOT and T gates. The representation is used in this thesis to generate the SAT encoding for the exact synthesis of {CNOT, T} circuits with the minimum number of CNOT gates.

4 Reversible circuits

4.1 Reversible gates

This section describes the building blocks of reversible computation. I use the following notation to describe reversible gates. A reversible gate performs an n-input reversible function which is applied to qubits (lines) $X = \{1, \ldots, n\}$. I further consider literals based on the elements in X, i.e., given $x \in X$, l is the positive literal and $\bar{l}$ is the negative literal of x. Note that $\bar{\bar{l}} = l$ and $|l| = |\bar{l}| = x$. Finally, let $l \oplus 0 = l$ and $l \oplus 1 = \bar{l}$. Also, for a given set of literals L, let $|L| = \{|l| \mid l \in L\}$ be the set of all variables of L.

Definition 4.1.1 (Single-target gate) *Let $c : \mathbb{B}^k \to \mathbb{B}$ be a Boolean function, called the* control function. *Also, let $C = \{c_1, \ldots, c_k\} \subset X$ be a set of* control lines *and let $t \notin C$ be a* target line. *Then the* single-target gate $\mathrm{T}_c(C, t) : \mathbb{B}^n \to \mathbb{B}^n$ *is a reversible Boolean function which maps*

$$(x_1, \ldots, x_n) \mapsto (x'_1. \ldots, x'_n) \quad \textit{where}$$
$$x'_i = \begin{cases} x_i & \textit{if } i \neq t, \\ x_t \oplus c(x_{c_1}, \ldots, x_{c_k}) & \textit{otherwise.} \end{cases}$$

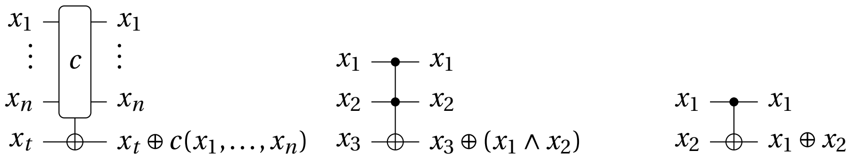
In other words, a single-target gate inverts the target, if and only if the control function evaluates to true. An example is shown in Fig. 4.1 (a).

Definition 4.1.2 (Multiple-controlled Toffoli gate) *If c can be expressed as a single product term*

$$c = \bigwedge_{i=1}^{k} (x_{c_i} \oplus p_i)$$

in a single-target gate $\mathrm{T}_c(C, t)$, *with $p_i \in \mathbb{B}$ giving the polarity of control i, then the gate is a* multiple-controlled Toffoli gate.

The 2-controlled Toffoli gate, simply known as *Toffoli*, is shown in Fig. 4.1 (b). A particular case



(a) Single-target gate (b) Toffoli (c) CNOT

Figure 4.1 – Examples of reversible gates.

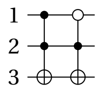


Figure 4.2 – Notations for the multiple-controlled Toffoli: the left gate is $T_{1\wedge 2}(\{1,2\},3)$ in the complete notation and $T(\{1,2\},3)$ in the special notation; for the second gate $T_{\bar{1}\wedge 2}(\{1,2\},3)$ and $T(\{\bar{1},2\},3)$.

is the CNOT gate, in Fig. 4.1 (c), which has a single control line. A special notation T(C', t) is introduced for these gates, where $C' = \{l \oplus p_l \mid l \in C\}$. This notation is explained by Fig. 4.2.

4.2 Synthesis of reversible circuits

This section reports state-of-the-art algorithms to synthesize a garbage-free reversible circuit implementing a possibly non-reversible Boolean function f. This is the first step of several automatic algorithms for the compilation of a quantum oracle for f.

Some state-of-the-art algorithms require to represent the input function f as a truth table that lists all input-output mappings explicitly [91, 96, 155]. Further, these algorithms require f to be already reversible, and they realize the reversible mapping without using any helper qubit, i.e., additional qubits initialized to $|0\rangle$. However, being truth-table based, such approaches are not suited to the compilation of large functions. If the target Boolean function is non-reversible, a preliminary step of embedding it into a reversible function is required to apply these algorithms. The embedding generates garbage outputs that are not compatible with quantum computation. For this reason, more gates are needed to uncompute such garbage outputs. There exist symbolic variants of these truth table based algorithms (see, e.g., [101, 156, 157]), in which the input function is represented symbolically, e.g., as binary decision diagram (BDD [158]) or using SAT encoding [146]. These methods can be applied to larger functions, but are not robust and still require a long runtime. In the remainder of this section, I will discuss two state-of-the-art methods that either are used by algorithms presented in this thesis or inspired them.

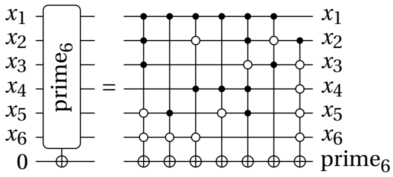


Figure 4.3 – Example of mapping a single-target gate into Toffoli gates using ESOP decomposition.

4.2.1 ESOP-based reversible logic synthesis

One can always decompose a single-target gate controlled by the function to synthesize $\mathrm{T}_c(\{x_1,\dots,x_k\},x_{k+1})$ into a cascade of Toffoli gates

$$\mathrm{T}_{c_1}(X_1,x_{k+1})\circ\mathrm{T}_{c_2}(X_2,x_{k+1})\circ\cdots\circ\mathrm{T}_{c_l}(X_l,x_{k+1}),$$

where $c=c_1\oplus c_2\oplus\cdots\oplus c_l$, each c_i is a product term or 1, and $X_i\subseteq\{x_1,\dots,x_k\}$ is the support of c_i. This decomposition of c is also referred to as *ESOP decomposition* [116, 117, 115]. ESOP minimization techniques can be applied to reduce the size of the ESOP expression.

ESOP-based reversible synthesis methods derive from the observation that an ESOP can be directly translated into a reversible circuit, as each term of the expression corresponds to a multiple-controlled Toffoli gate [97, 159]. The method generates as many Toffoli gates as cubes in the expression, performed in sequence and targeting the same line. By construction, the generated circuits are garbage-free.

Example 4.2.1 *The Toffoli network in Fig. 4.3 corresponds to the ESOP expression of the $prime_6$ function:*

$$prime_6=x_1x_2x_3\overline{x}_5\overline{x}_6\oplus x_1x_5\overline{x}_6\oplus x_1\overline{x}_2x_4\overline{x}_6\oplus x_1x_4\overline{x}_5\oplus x_1x_2\overline{x}_3x_4x_5\oplus x_1\overline{x}_2x_3\oplus x_2\overline{x}_3\overline{x}_4\overline{x}_5\overline{x}_6$$

The decomposed circuit reflects the quality of the ESOP expression, so the synthesis process is crucial for this application.

4.2.2 Hierarchical reversible synthesis

Hierarchical reversible synthesis starts from a multi-level logic network representation implementing the function to be synthesized. It exploits helper qubits to store intermediate results computed by the logic network's nodes. A simple variant of this approach was already presented by Tommaso Toffoli himself [160]. The starting gate-level netlist consisted of AND and NAND gates and the result of each gate was computed by a corresponding Toffoli and saved on an helper line. The approach has been picked up and integrated into the quantum programming language Quipper [161]. Based on the synthesis idea from Toffoli, other hierarchical synthesis approaches were presented that mainly differ in the underlying logic network representation, e.g., decision diagrams [162, 94, 163], functional decomposition [103], fanout-free regions in And-Inverter Graphs [104], or XOR-majority graphs [105].

The main drawback of these methods is the redundancy of the obtained reversible circuit and the large number of required helper qubits. Restructuring and optimizing the underlying logic network can lead to a significant reduction in the number of nodes and consequently in the number of qubits and gates of the resulting circuit.

4.3 Decomposing reversible gates into quantum circuits

Given that all quantum circuits must be reversible, often the quantum circuit implementation of combinational logic functions is derived from an intermediate reversible representation.

4.3.1 Gray synthesis

In this section, I describe a state-of-the-art method to synthesize a quantum circuit composed of CNOT, H and $R_z(\theta)$ gates from a single-target gate [95]. Any unitary matrix U_f realizing the single-target gate on qubit x_n with control function $f(x_1,\dots,x_{n-1})$ can be decomposed as follows:

$$U_f = (I_{2^{n-1}} \otimes H) \cdot diag(\hat{g}_0,\dots,\hat{g}_{2^n-1}) \cdot (I_{2^{n-1}} \otimes H)$$

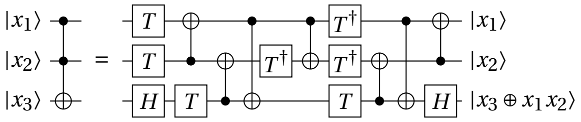
where $\hat{g}_0 \dots \hat{g}_{2^n-1}$ is the truth table of the function $g = x_n \wedge f$ in the $\{-1,1\}$ encoding. The authors of [99] proved that the matrix $diag(\hat{g}_0,\dots,\hat{g}_{2^n-1})$ can be implemented only using CNOT and R_z gates. The matrix is equivalent to the unitary performing the mapping

$$|x\rangle \mapsto e^{i\pi s(x)/2^n}\,|x\rangle\,,$$

where the phase polynomial $s(x)$ assigns to each parity combination of the inputs a coefficient equal to the corresponding coefficient in the Rademacher-Walsh spectrum of the Boolean function g. Authors of [98] presented an efficient heuristic algorithm for the synthesis of small parity networks, which is capable of generating all the parity functions in the phase polynomial $s(x)$ with a reduced amount of CNOT gates. The described synthesis approach owes its name to this algorithm. For each parity function generated, a rotation gate $R_z(\theta_i)$ is performed, where $\theta_i = \pi s_i/2^n$ and s_i is the spectral coefficient corresponding to that parity function. It follows that, the more zero coefficients in the Rademacher-Walsh spectrum, the fewer rotation gates in the generated quantum circuit.

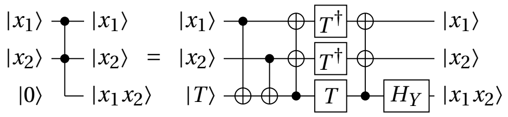
4.3.2 Quantum circuits for the Toffoli gate

This section describes state-of-the-art quantum implementations of the cited Toffoli gate. The 2-control Toffoli gate has a Clifford+T implementation that requires 7 T gates [90], which is optimum [164, 165], i.e., minimum T-count and T-depth:



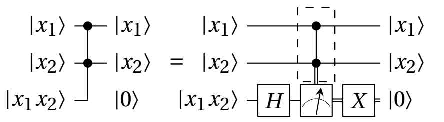
(4.1)

When the Toffoli gate is computed on a qubit initialized to $|0\rangle$, it can be implemented using 4 T gates, with a T-depth of 2, and without requiring any additional qubit [166, 167]:



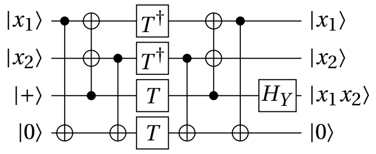
(4.2)

where $H_Y = SH$, $S = T^2$ and $|T\rangle = TH|0\rangle$. Besides, when the result of the Toffoli is uncomputed, meaning that a qubit storing $|x_1 x_2\rangle$ is brought back to $|0\rangle$, this can be performed without the use of any T gate, exploiting measurement-based uncomputation, as shown:



(4.3)

There exists also another AND gate implementation with a T-depth equal to 1, which combines the AND circuit from [167] and the Toffoli gate implementation in [168] characterized by a T-depth of 1. The circuit requires one extra qubit with respect to the implementation in (4.2):



(4.4)

where $|+\rangle = H|0\rangle$.

Several works from the literature describe how to map multiple-controlled Toffoli gates into Clifford+T gates (see, e.g., [90, 164, 168, 169]). Among them, a method proposed by *Barenco et al.* [89] allows us, provided an additional qubit, to map any multiple-control Toffoli gate into a sequence of 2-control Toffoli gates, which can be implemented using the optimum circuits in (4.1) and (4.2). In general, a k-controlled Toffoli gate can be realized with at most $16(k-1)$ T gates. If the number of helper lines is larger or equal to $\lfloor \frac{k-1}{2} \rfloor$, then $8(k-1)$ T gates suffice [90, 89]. Future improvements to the decomposition of multiple-controlled Toffoli gates into Clifford+T circuits will have an immediate positive effect on some of the algorithms proposed in this thesis.

4.4 Summary

This chapter introduced reversible circuits and described the reversible gates used in the remainder of this thesis. Indeed, methods to generate quantum circuits implementing non-reversible Boolean functions take advantage of reversible circuits as intermediate representations. In Section 4.2, I reported state-of-the-art methods to synthesize reversible circuits for non-reversible Boolean functions, including hierarchical methods largely investigated in this thesis. Finally, Section 4.3 illustrated state-of-the-art decomposition techniques to map reversible gates into Clifford+T quantum circuits.

Part II

Quantum circuit synthesis and optimization

5 Hierarchical reversible synthesis

This chapter addresses the problem of compiling a particular category of quantum circuits, i.e., quantum *oracles*, which implement classical Boolean functions. I describe a method capable of synthesizing quantum circuits from high-level definitions of Boolean functions, i.e., multi-level logic networks. This method, called *hierarchical reversible synthesis*, generates a reversible network that serves as intermediate representation. Each reversible gate of this network is further decomposed into a quantum circuit.

I first give an overview of the hierarchical synthesis process, which is characterized by three main steps: (1) decomposing the network, (2) mapping the decomposed logic into qubits using reversible gates, and (3) decomposing each reversible gate into a quantum circuit.

Then, I propose and describe new methods for steps (1) and (2) to improve the state-of-the-art flow: Section 5.2 proposes a decomposition method specifically designed for this application, and Section 5.3 discusses a SAT-based technique to optimally perform the second step according to a given constraint in the number of qubits. Chapter 7 will focus on step (3), which is a self-contained problem.

5.1 Method overview

Given an n-input m-output Boolean function $f(x_1,\dots,x_n) = (y_1,\dots,y_m)$ with $x = x_1\dots x_n$ and $y = y_1\dots y_m$, the mapping

$$O_f : |x\rangle|y\rangle|0^l\rangle \mapsto |x\rangle|y \oplus f(x)\rangle|0^l\rangle, \tag{5.1}$$

describes a $2^{n+m+l} \times 2^{n+m+l}$ unitary operation, which permutes the amplitudes of a quantum state according to $f(x)$. It can be seen that the n inputs are passed through n qubits $|x\rangle$ that can be in an arbitrary quantum state. The outputs are computed onto m qubits, which can be in an arbitrary state $|y\rangle$. Note that '$\oplus$' refers to the bitwise XOR operation. Finally, there are l qubits that are all in state $|0\rangle$ before and *after* the computation. These are called *helper qubits* and can be used to store intermediate temporary results.

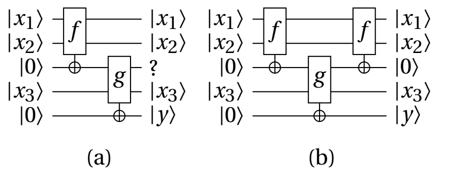


Figure 5.1 – (a) Circuit computing $y = g(f(x_1, x_2), x_3)$ with two single-target gates generating an unknown intermediate result. (b) Garbage-free circuit where the intermediate result has been uncomputed by applying f twice.

The described unitary operations are used in many quantum algorithms, e.g., Shor's algorithm for integer factorization [18] and Grover's algorithm for unsorted database search [76]. Compiling such unitaries means finding a quantum circuit that realizes the function without the need to explicitly represent O_f but only based on a symbolic representation of f, e.g., given as a logic network. Restoring the helper qubits to the $|0\rangle$ state is crucial, since intermediate computation results that remain in the output quantum state are perceived as noise by subsequent operations in the quantum algorithm, eventually leading to wrong results. The number of additional qubits is application- and hardware-specific. As an example, in order to realize an oracle performing an 8-input, 8-output Boolean function on a 20-qubit quantum computer, one can use at most $l = 4$ helper qubits.

k-LUT-based hierarchical synthesis uses k-LUT decomposition as the first step of the synthesis flow and is capable of finding a quantum circuit for O_f when f is represented as a logic network. The method is called *hierarchical* because it starts from a multi-level logic network representation of the Boolean function and then *hierarchically* traverses the graph translating it into a reversible network. The reversible network is built using single target gates—a gate that complements the target if its Boolean control function evaluates to true, see Definition 4.1.1. To comply with Equation 5.1, intermediate reversible networks are required to be *garbage-free*, which means that all intermediate results need to be uncomputed.

Example 5.1.1 *Fig. 5.1 (b) shows a garbage-free reversible circuit that performs the mapping* $|x_1x_2x_3\rangle\,|0^2\rangle\,|0\rangle \mapsto |x_1x_2x_3\rangle\,|0^2\rangle\,|g(f(x_1,x_2),x_3)\rangle$*, in which the intermediate state* $|f(x_1,x_2)\rangle$ *is uncomputed by repeating a single-target gate controlled by* f*.*

As illustrated by Fig. 5.2, k-LUT-based hierarchical reversible synthesis is performed in three steps:

1. **k-LUT mapping.** The step decomposes f into a k-LUT network using k-LUT mapping. Fig. 5.2 shows a 2-LUT mapping performed on the logic network representing f in Fig. 5.2 (a), generating the network with LUTs $n_1, \ldots, n_4$ in Fig. 5.2 (b). This initial step has a relevant impact on the result of the compilation. The k parameter allows us to control the maximum permissible number of inputs of each sub-network: a smaller k will generate many sub-networks controlled by small functions, which can

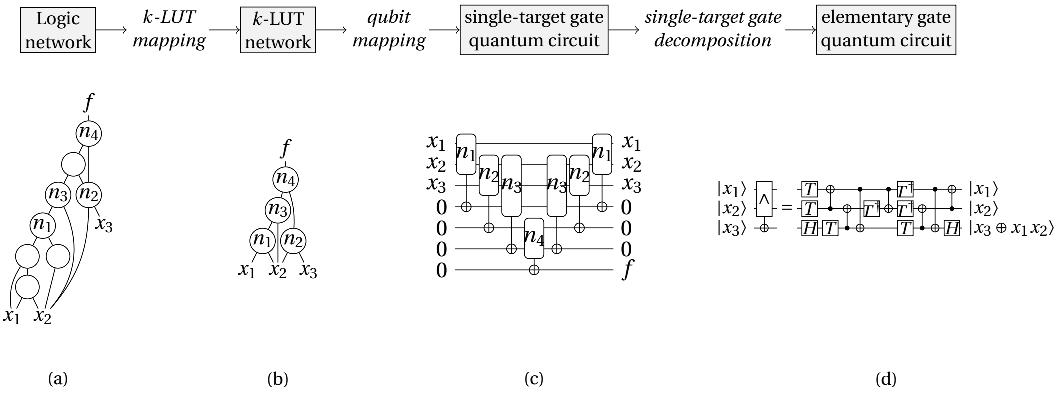


Figure 5.2 – k-LUT-based hierarchical quantum compilation flow illustrating the example of a three-input multi-level logic network performing the Boolean function f (a), mapped into a 2-LUT network for f (b), translated into the reversible circuit for the 2-LUT network using the Bennett clean-up strategy (c) then decomposed into a quantum circuit (d).

be synthesized by mapping methods that do not scale well with the function size. At the same time, having many sub-networks leads to many intermediate results to be allocated in the quantum memory, hence more helper qubits.

2. **Qubit mapping.** This step assigns the logic gates of the k-LUT network to the $n+m+l$ qubits in the resulting quantum circuit, making sure that no more than l temporary values are stored at any time. This step is illustrated in Fig. 5.2: one single-target gate is created for each LUT in Fig. 5.2 (b) and placed on the reversible circuit in Fig. 5.2 (c). Each single-target gate computes on a helper line an intermediate value using the logic function of the corresponding LUT. There are several possible strategies to perform this step [170, 171], which explore the trade-off between helper lines and single-target gates.

3. **Single-target gate (STG) decomposition.** The step maps each single-target gate into a quantum circuit of elementary gates supported by the targeted quantum device. It requires compilation methods capable of generating quantum circuits with a limited T-count, T-depth and number of qubits, when the Clifford+T library is selected. Fig. 5.2 (d) shows a well known decomposition of a single-target gate controlled by the 2-input AND function, i.e., a Toffoli gate, into a quantum circuit with 7 T gates.

Hierarchical methods for the synthesis of quantum circuits have shown the ability to synthesize large functions and enable a certain control over the number of helper qubits, through the parameter k of the decomposition. In the next sections, I will introduce new techniques that improve the described state-of-the-art hierarchical synthesis method, which is called *LUT-based Hierarchical Reversible Synthesis (LHRS)* and was initially proposed in [106]. In particular, I describe a new k-LUT mapper to be used to perform the first step of the synthesis flow, and a memory management technique to be used in the second step of the described flow to explore the trade-off between number of operations and number of qubits.

5.2 Quantum-aware k-LUT mapping

Hierarchical reversible synthesis exploits k-LUT mapping to decompose the initial network. The state-of-the-art hierarchical synthesis method LHRS [106] exploits an area-oriented LUT mapper called *mf*, which is part of the logic synthesis framework *abc* [119], to generate a LUT network that is used as a starting representation for the synthesis flow. In LHRS, k-LUT mapping is performed considering metrics such as delay and area, which do not directly impact the cost functions of quantum circuits. When k-LUT mapping is used in the context of quantum circuit synthesis, the classical metrics must be changed to relevant ones in this application. In this thesis, this criticality is addressed by developing a new quantum-aware k-LUT mapper that aims at minimizing the number of gates required to synthesize the quantum circuit of each LUT.

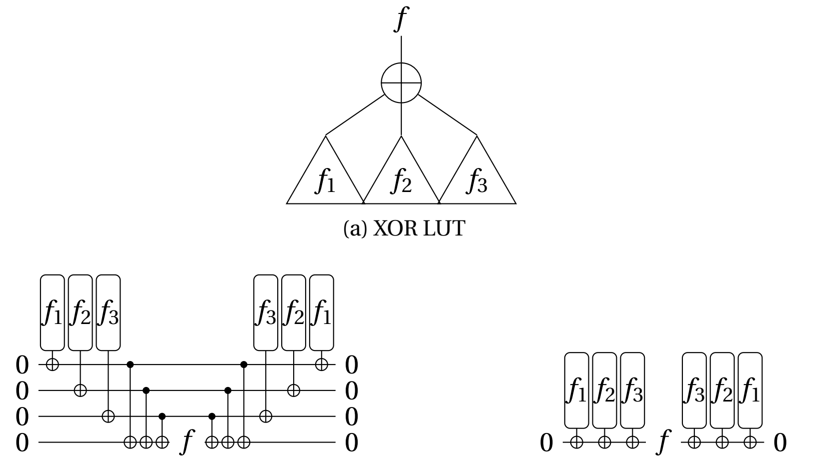
As already described, the choice of the k parameter defines the number and the complexity of the obtained sub-networks, which are then translated into single-target gates. Besides, it is possible to tune this decomposition step to select sub-networks implementing functions which can be "easily" translated into quantum circuits, meaning that the compilation method produces better results.

Consider the case in which the framework uses a specific decomposition method to decompose each single-target gate into Clifford+T circuits: the Gray synthesis method [98, 99]. This technique decomposes each single-target gate with control function f into a quantum circuit that consists of the following quantum operations: CNOT, H, $R_z(\theta)$ (see section 4.3.1). The method is characterized by a direct dependence between non-zero coefficients in the Rademacher-Walsh spectrum of the function f (see Definition 2.2.3) and number of CNOT and R_z gates to be synthesized. This dependence is exploited by selecting the number of non-zero coefficients as cost function for each LUT, trying to obtain LUTs with as fewer non-zero coefficients as possible.

5.2.1 Cut enumeration and costing

The logic network used as input of the proposed mapping method is the Xor-And-inverter Graph (XAG): a graph in which each node performs the XOR or the AND operation and edges can be complemented to perform inversion. The choice of this data structure reflects the fact that it is relatively inexpensive to perform the XOR operation in fault-tolerant quantum circuits. Only one CNOT gate is needed to perform the XOR between two qubits, and in general $m-1$ CNOT gates are needed to perform an m-input XOR gate. If the result is stored on a helper qubit, m CNOT gates are needed in total.

The first step performed by the k-LUT mapper is the so-called *cut enumeration*. This step consists of enumerating all possible cuts for each node of the input graph, traversing the graph from the bottom to the top. Only the best cuts are stored for each node, a technique called *priority cuts* that reduces the memory requirement of *cut enumeration* [172]. During cut



(a) XOR LUT

(b) Conventional mapping into qubits

(c) Improved mapping into qubits

Figure 5.3 – Mapping into qubits LUTs performing the XOR function.

enumeration, each node is assigned to a set of p cuts that are ordered following a user defined cost criterion. As this mapper is developed in the context of a quantum synthesis framework, the number of non-zero spectral coefficients of the function performed by the selected cut is used as cost function. As previously pointed out, the Gray synthesis method generates smaller quantum circuits when the input function presents a spectrum with many zeros. At the end of the cut enumeration step, each node of the XAG is assigned with an ordered set of p cuts, with the order criteria defined according to the spectrum of the corresponding function.

5.2.2 XOR-block matching

After *cut enumeration*, the mapper has to extract the LUT mapping. This corresponds to marking certain nodes of the graphs, which will be the output nodes of the sub-networks. A standard mapper would start by marking each output node and by selecting the first cut in the ordered set of cuts relative to that node. Then, it would mark all the leaves of the selected cuts and find the best cut for these leaves as well. This is repeated until all the primary inputs are marked.

The quantum-aware mapping identifies the cuts performing the parity function and makes sure to always select them during the mapping extraction, as they can be conveniently synthesized as multi-input XOR gates. For this reason, after *cut enumeration*, cuts are adjusted to isolate multi-input XOR operations, called XOR blocks.

If some leaves of an XOR block have a fan-out size of 1, i.e., they only fan-in into the XOR block, an alternative mapping strategy is possible, which leads to reduction of qubits and gates. Fig. 5.3 illustrates the improved mapping strategy for an XOR cut with three inputs. In Fig. 5.3 (a) this LUT is drawn as an XOR symbol. The conventional mapping into qubit, shown in Fig. 5.3 (b), maps each child into a clean helper qubit, and then uses another clean qubit to map the result of the XOR cut. The result of that cut, f, can then be used as input by subsequent gates. Fig. 5.3 illustrates this fact by simply annotating the circuit line where it represents the value f. However, since in this case the child cuts f_1, f_2, and f_3 are composed via the XOR operator, one can directly map them in to a single qubit without the need of requiring an additional qubit for each child LUT, see Fig. 5.3 (c).

Note that the size of the XOR gates does not need to be bounded by the LUT size k. In order to build XOR blocks in XAGs, first 2-input XOR gates are detected. Then, sub-trees of XOR gates are grouped together. Finally, the modified cut enumeration assigns cost 0 to the XOR cuts, in order to force the LUT mapping to prefer XOR blocks.

5.3 Quantum memory management

Single-target gate quantum circuits, used as reversible intermediate representation by the k-LUT hierarchical synthesis, are required to be garbage-free, which means that no intermediate result is accessible from the outputs. Otherwise, as many states can be entangled together, measurement of intermediate results may compromise the computation. This requirement is enforced in step (2) of the hierarchical synthesis flow described in Section 5.1, which aims at mapping the logic functions performed by each LUT into the qubits.

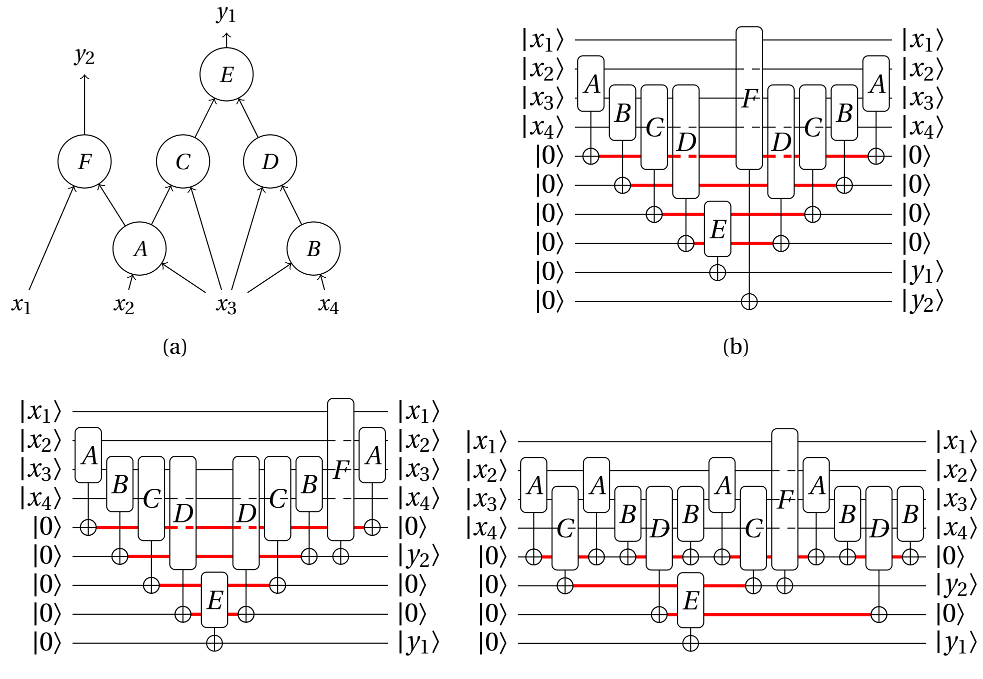
Example 5.3.1 *In the remainder of this section, consider the example of a quantum algorithm that performs the following mapping:* $|x_1\rangle|x_2\rangle|x_3\rangle|x_4\rangle|0\rangle|0\rangle \mapsto |x_1\rangle|x_2\rangle|x_3\rangle|x_4\rangle|y_1\rangle|y_2\rangle$ *where*

$$z_1 = A(x_2, x_3) \qquad z_2 = C(z_1, x_3) \qquad z_3 = B(x_3, x_4)$$
$$z_4 = D(z_3, x_3) \qquad y_1 = E(z_2, z_4) \qquad y_2 = F(x_1, z_1)$$

with A, B, C, D, E, F *being some generic 2-input Boolean operations and* z_1, z_2, z_3, z_4 *the intermediate results. Such computation is represented in Fig. 5.4 (a) using a Directed Acyclic Graph (DAG), in which each node corresponds to one part of the operations, and edges define data dependencies: an edge is drawn from a node* v *to a node* w *if* w *requires the value of* v *(see Fig. 5.4 (a)).*

The hierarchical synthesis method traverses the DAG and maps each logic operation to the available helper qubits. There are several ways of performing this mapping, leading to a different number of required helper qubits. Three different reversible circuits resulting from the DAG in Fig 5.4 (a) are shown in Fig. 5.4 (b-d).

The three circuits are different in the order in which each single-target gate is performed. Values stored on helper qubits can be computed and uncomputed more than once, such that when the results y_1 and y_2 have been computed, all the intermediate values z_1, z_2, z_3, z_4 are cleaned up. A simple solution is the one proposed in Fig. 5.4 (b), which is referred to as the



(a) (b)

(c) (d)

Figure 5.4 – (a) Example of a DAG. (b)(c)(d) Three different uncomputing strategies: (a) Bennett strategy; (b) space-optimized by reordering; (c) space-optimized by increasing the number of gates.

Bennett strategy [170]. It consists of computing all the operations in a bottom-up order, and then uncomputing the intermediate results in a top-down fashion, so that all the nodes have their inputs available. This strategy always leads to the minimum number of single-target gates, and to the maximum number of helper qubits. The order in which the DAG is converted into a reversible circuit can have a significant effect on how the memory is managed. For example, the strategy illustrated in Fig. 5.4 (c) saves one qubit only by changing the order of the operations, without increasing their number. Finally, the number of helper qubits can be further reduced to 4 by allowing an increase in the number of gates. In this case, some functions are computed several times, see Fig. 5.4 (d).

In Fig. 5.4 (b-d) lines are drawn in red whenever their corresponding qubit is storing an intermediate result. The first two strategies store values for a long time during which they are not needed, whereas the last strategy makes a better usage of fewer memory locations. The three circuits are useful to visualize the trade-off between space (i.e., qubits) and time (i.e., gates).

This section introduces a memory management technique that enables to select the best strategy to perform the mapping into qubits, achieving a garbage-free reversible circuit using the minimum number of single-target gates, while relying on a fixed number of helper qubits.

5.3.1 Reversible pebbling game

Finding the best strategy to uncompute all intermediate results corresponds to solving the reversible pebbling game problem. For this reason, the term *pebbling strategy* is used in the remainder of this thesis.

The reversible pebbling game problem was introduced by Bennett in [170], in the context of exploring space/time trade-off in reversible computation. The game is played by placing and removing pebbles from the nodes of a DAG modeling the computation. When a pebble is placed on a node, the node is *pebbled*, meaning that the value computed by the operation carried out by that node is available on a qubit. Initially, no node is pebbled. A pebble can be placed on a node if all its children are pebbled. While a pebble can be removed from a node at any time according to the rules of the "standard" pebbling game, the reversible version requires that all the children of the nodes are pebbled to perform such a move. The game ends if all the outputs are pebbled and all the other nodes are unpebbled. Solving the problem returns a valid strategy to clean up intermediate results stored on helper qubits.

The problem complexity has been studied in [173], where the author proves that finding the minumum number of pebbles for a given DAG is PSPACE-complete, as in the case of the non-reversible pebbling game. Besides, the problem is PSPACE-hard to approximate up to an additive constant [174]. An explicit asymptotic expression for the best time-space product is given in [175]. Asymptotic space bounds are known for chain graphs [171] and complete binary trees [176]. The asymptotic behavior on trees is studied in [177]. Complexity trade-off results for the problem applied to general DAGs are given in [178].

Solving the reversible pebbling game corresponds to finding a *reversible pebbling strategy*, which is a sequence of *reversible pebbling configurations.*

Definition 5.3.1 (Reversible pebbling configuration) *A reversible pebbling configuration of a DAG $G = (V, E)$ is a subset $P \subseteq V$ of all the pebbled vertices.*

Definition 5.3.2 (Reversible pebbling strategy) *A reversible pebbling strategy of a DAG G is a sequence of reversible pebbling configurations $P = (P_1, ..., P_m)$ such that $P_1 = \{\}$ and $P_m = O$, where O is the set of all the outputs of G. For each $1 < i \leq m$, we have that if $P_i = P_{i-1} \cup \{v\}$ or $P_i = P_{i-1} / \{v\}$ and $P_i \neq P_{i-1}$, all the children of v are in P_{i-1}.*

Fig. 5.5 shows two possible pebbling strategies for the algorithm in Example 5.3.1, described by the DAG in Fig. 5.4 (a). Each row of the grids represents a node and each column, from left to right, corresponds to a single step. A black square means that the corresponding node is pebbled. In accordance with the rules of the game, the initial configuration is empty and the last only contains output vertices. In these examples only one move per step is allowed. The first strategy is the one reported by Bennett [170], which naively computes all the nodes and then uncomputes the intermediate results. This pebbling requires a number of pebbles equal

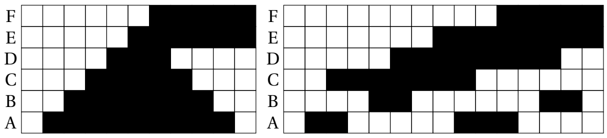


Figure 5.5 – Two different pebbling strategies for a given DAG.

to the number of nodes, 6 in the example, and only 10 steps, that is minimum. The complete sequence of pebbling configurations for the first example is:

$$\begin{array}{ll}
P_0 = \{\} & P_6 = \{A, B, C, D, E, F\} \\
P_1 = \{A\} & P_7 = \{A, B, C, E, F\} \\
P_2 = \{A, B\} & P_8 = \{A, B, E, F\} \\
P_3 = \{A, B, C\} & P_9 = \{A, E, F\} \\
P_4 = \{A, B, C, D\} & P_{10} = \{E, F\} \\
P_5 = \{A, B, C, D, E\} &
\end{array}$$

The second approach is a strategy that only uses 4 pebbles. To reduce the number of pebbles, it computes twice the nodes a and b, increasing the number of steps to 14. The complete sequence of pebbling configurations for the second example is:

$$\begin{array}{ll}
P_0 = \{\} & P_8 = \{A, C, D, E\} \\
P_1 = \{A\} & P_9 = \{A, D, E\} \\
P_2 = \{A, C\} & P_{10} = \{A, D, E, F\} \\
P_3 = \{C\} & P_{11} = \{D, E, F\} \\
P_4 = \{B, C\} & P_{12} = \{B, D, E, F\} \\
P_5 = \{B, C, D\} & P_{13} = \{B, E, F\} \\
P_6 = \{C, D\} & P_{14} = \{E, F\} \\
P_7 = \{C, D, E\} &
\end{array}$$

I have shown here two possible pebbling configurations, which correspond to valid uncomputing strategies for all intermediate results. The problem of finding the best configuration is encoded as a Boolean satisfiability problem and can be solved using state-of-the-art SAT solvers.

5.3.2 SAT-encoding

In [179], we proposed to solve the reversible pebbling game by casting it as a satisfiability problem and by exploiting state-of-the-art SAT solvers. In particular, we aimed at minimizing the number of steps of a valid pebbling strategy while constraining the maximum number of pebbles per step, which corresponds to the required number of helper qubits.

Problem 5.3.1 (Optimal reversible pebbling) *Given a DAG and P pebbles, find a valid pebbling strategy using the minimum number of steps.*

To use SAT solvers to extract a solution, this problem has to be decomposed into many SAT problems.

Problem 5.3.2 (Bounded reversible pebbling) *Given a DAG and P pebbles, does a valid pebbling strategy with K steps exist?*

The solver can either find a solution and return a pebbling strategy, or state that no solution exists. In this case the number of steps is increased to $K+1$ until a satisfying solution is found.

Following the definition of a reversible pebbling game given in Section 5.3.1, this section describes the SAT encoding of Problem 5.3.2. The set of declared Boolean variables and the constraints (clauses) describing a valid solution to the problem are defined. The input DAG $G = (V, E)$ is characterized by a set $O \subseteq V$ of nodes that compute output values. Note that the primary inputs are *not* nodes of the DAG. Also, $C(v) = \{w \mid w \rightarrow v\}$ is defined as the set containing all the children of node v.

Example 5.3.2 *The DAG in Fig. 5.4 (a) has six nodes $\{A, B, C, D, E, F\}$ and two outputs $O = \{E, F\}$. Note that, e.g., $C(A) = \{\}$, since the primary inputs are not part of the DAG.*

Variables

Problem 5.3.2 is encoded in terms of the *pebble state variables $p_{v,i}$*. For $v \in V$ and $0 \le i \le K$, these are Boolean variables that evaluate to true if the node v is pebbled at time i. Note that the SAT formula encodes $K+1$ pebble configurations with K steps describing the transition from one configuration to the next.

Clauses

The following set of clauses describes the reversible pebbling problem.

- *Initial and final clauses:* at time 0 all the nodes are unpebbled, and at time K all the outputs need to be pebbled and all the intermediate results unpebbled

$$\bigwedge_{v \in V} \bar{p}_{v,0} \wedge \bigwedge_{v \in O} p_{v,K} \wedge \bigwedge_{v \notin O} \bar{p}_{v,K}$$

- *Move clauses:* if a node is pebbled or unpebbled at time $i+1$, then all its children are pebbled at time i and time $i+1$

$$\bigwedge_{i=1}^{K} \bigwedge_{w \in C(v)} ((p_{v,i} \oplus p_{v,i+1}) \rightarrow (p_{w,i} \wedge p_{w,i+1}))$$

- *Cardinality clauses:* at each step, at most P pebbles are used

$$\bigwedge_{i=0}^{K} (\sum_{v \in V} p_{v,i} \leq P)$$

5.3.3 Show-cases

This section illustrates how quantum memory management, achieved by solving the reversible pebbling game using the open-source SAT solver Z3 [148], allows us to optimally exploit limited qubit resources.

Show-case: Straight-line programs

The first example illustrates how the proposed method can be applied to the synthesis of straight-line programs used in cryptographic applications. Such programs are combinations of modular arithmetic operations, e.g., addition, subtraction, multiplication, and squaring. One can assume that for each operation a quantum implementation exists, and will have a given cost in terms of quantum gates and qubits. The method allows us to estimate the cost of an algorithm implementation of these functions in terms of number of different operations, according to a fixed number of available qubits.

A straight-line program that implements the addition between two points of an Edward curve in projective coordinates from [180] is selected as benchmark. The resulting DAG is pebbled using different numbers of pebbles, corresponding to different numbers of qubits. Fig. 5.6 shows the pebbling strategies obtained with 24, 20, 16, 12, and 10 pebbles. The figure reports, for each case, a different number of operations. For example, the first implementation performs a total of 74 operations: 28 additions, 20 subtractions, 15 squaring and 11 multiplication. It is clear how the tool manages to fit the desired computation into limited number of qubits, by increasing the number of required steps. As a consequence, the last implementation has a higher cost in terms of operations: 110 in total. The overall cost of the algorithm on different hardware can be evaluated, provided some estimates of the real cost of each operation. The dynamic change in the memory employed during the computation is shown at the top of each grid. A flat dynamic suggests that a constant number of qubits is used through the whole computation. While a solution with a lower peak requires fewer qubits.

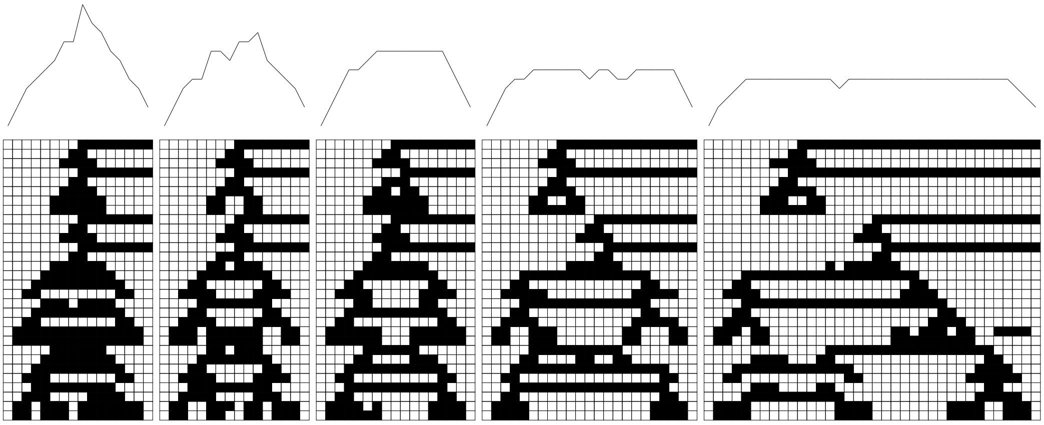


Figure 5.6 – Illustration of how *pebbling* maps the given computation into a fixed number of helper qubits: respectively 24 (Add:28, Sub:20, Sqrt:15, Mult:11), 20 (Add:36, Sub:32, Sqrt:21, Mult:9), 16 (Add:28, Sub:24, Sqrt:17, Mult:13) , 12 (Add:24, Sub:34, Sqrt:19, Mult:13) and 10 (Add:34, Sub:38, Sqrt:25, Mult:13).

Show-case: Comparison with Bennett strategy

The second show-case has the purpose of quantifying the ability of the program to map a design into a limited number of qubits. As benchmark, consider the operator called H (different from the Hadamard gate) that is used internally to the algorithm that computes the doubling of two points referred before [180]. This operator is a composition of modular additions (+) and modular subtraction (−); it has a, b, c, d as inputs and four outputs x, y, z, t, where:

$$t_1 = a + b \quad t_2 = c + d \quad t_3 = a - b \quad t_4 = c - d$$

$$x = t_1 + t_2 \quad y = t_1 - t_2 \quad z = t_3 + t_4 \quad t = t_3 - t_4$$

Experiments reported in Table 5.1 show a comparison between the Bennett pebbling method and the strategy obtained pebbling the corresponding DAG. The different designs correspond to the H operator with different bitwidths and modulus, and to the well known ISCAS benchmark. The graph representation for the function has been extracted from an XOR-majority graph using the open-source tool *mockturtle* [181]. The number of pebbles reported in Table 5.1 corresponds to the minimum one for which the solver could find a solution within 2 minutes. Even with this short timeout, the algorithm finds a solution for a significantly reduced number of pebbles. The average percentage reduction is 52.77%. As the pebbles are reduced, the number of steps (single-target gates) is in average 2.68× the one of the Bennett method. With the increase of the size of the DAG, the obtained pebble reduction is smaller. The reason is in the chosen timeout, as the solver requires more time to deal with larger designs: the number of variables of the SAT problem is proportional to n^2, where n is the number of nodes of the DAG. Besides, more steps are required to pebble a design with a larger number of nodes. This also depends on the timeout. Indeed, the algorithm is capable of finding many solutions

Table 5.1 – Comparison between the Bennett and the pebbling strategy.

				Bennett		Pebbling strategy				
	pi	po	nodes	P	K	P	K	runtime	%P	×K
b2_m3	8	8	74	66	124	30	186	0.17	54.55	1.5
b3_m4	12	12	59	47	82	20	117	121.37	57.45	1.43
b4_m5	16	16	203	187	358	83	778	55.75	55.61	2.17
b5_m7	20	20	256	236	452	106	888	31.15	55.08	1.96
b6_m7	24	24	310	286	548	130	1132	35.72	54.55	2.07
b8_m7	32	32	422	390	748	187	1884	11.59	52.05	2.52
b10_m7	40	40	535	495	950	264	2938	28.66	46.67	3.09
b12_m7	48	48	646	598	1148	331	4228	56.33	44.65	3.68
b16_m23	64	64	881	817	1570	480	6218	133.45	41.25	3.96
c17	5	2	12	7	12	4	12	0.01	42.86	1
c432	36	7	208	172	337	60	685	23.70	65.12	2.03
c499	41	32	219	178	324	77	610	60.08	56.74	1.88
c880	60	26	334	274	522	82	1280	43.52	70.07	2.45
c1355	41	32	219	178	324	77	594	2.63	56.74	1.83
c1908	33	25	220	187	349	70	875	57.97	62.57	2.51
c2670	157	63	554	397	731	160	1948	47.94	59.7	2.66
c3540	50	22	856	806	1590	416	5434	111.20	48.39	3.42
c5315	178	123	1257	1079	2035	498	7635	118.38	53.85	3.75
c6288	32	32	1011	979	1926	640	10232	101.31	34.63	5.31
c7552	207	108	1151	944	1780	540	7757	124.1	42.8	4.36

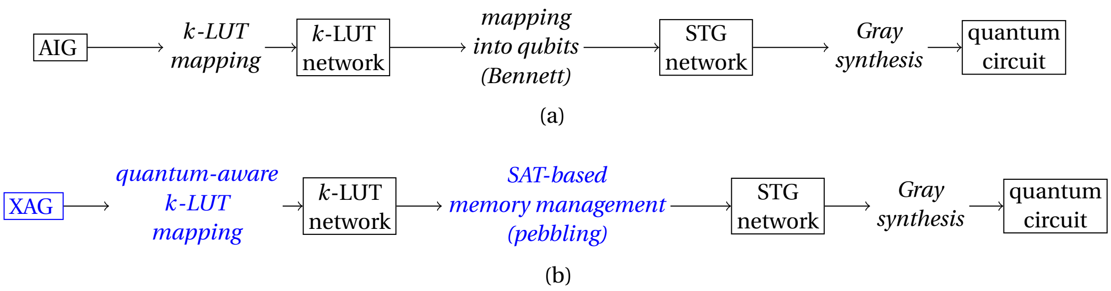
Average percentage reduction of pebbles = 52.77
Average multiplicative factor for the number of steps = 2.68

with different number of pebbles but same number of steps, while more constrained solutions (in number of steps) require a longer runtime.

5.4 Results

This section presents a comparison between the the state-of-the-art flow LHRS [106] and the hierarchical synthesis method equipped with the techniques described in this chapter, called Resource-constrained Oracle Synthesis (ROS) [182]. The efficiency of the proposed approach is demonstrated by synthesizing some quantum oracles.

By combining the quantum-aware mapping method, SAT-based quantum memory management, together with the Gray synthesis method, the results show that it is possible to improve both the number of qubits and the number of gates, compared to the state-of-the-art method. The two compared flows are shown in Fig. 5.7. Fig. 5.8 shows a qualitative description of the expected performance advantages. In the plot, the state-of-the-art result is the one marked as M/B (corresponding to the *LHRS* synthesis framework). If one only applies the memory management technique in Section 5.3, but not the new mapper proposed in Section 5.2, then a circuit with fewer qubits and more gates is obtained, which corresponds to M/P in the figure. If instead one embeds into *LHRS* the quantum aware k-LUT mapper, but no quantum memory management, then a circuit with fewer gates but more qubits is obtained: S/B. Only the combination of both techniques (S/P) can beat the state-of-the-art tool in both qubits and gates. Indeed, the approach can be tuned to only improve the qubit count while not increasing



(a)

(b)

Figure 5.7 – (a) LHRS flow; (b) ROS flow.

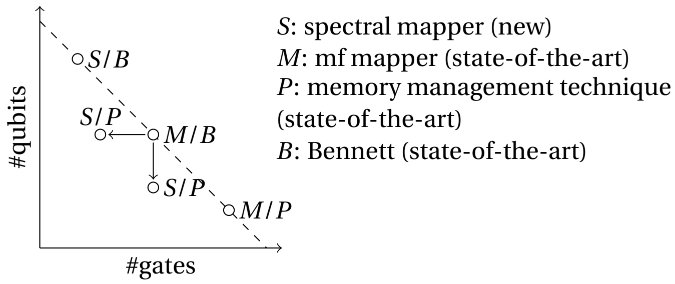


Figure 5.8 – Qualitative description of ROS's capability.

the gate count, or vice versa. This qualitative description is supported by the results described in this section.

The experimental evaluation consists of synthesizing oracle circuits for Grover's algorithm, assuming to perform equivalence checking between two logic designs. Equivalence checking is a well-known problem in logic synthesis that has been addressed by many logic synthesis tools, as for example *abc* [119] or *Formality*®. The function f performed by an oracle circuit is satisfied when the two graphs perform different operations. The algorithm would either prove that the two circuits are equivalent, or would provide the input set for which the two functions evaluate differently.

The synthesized Boolean functions are represented using XAGs. Each graph represents an equivalence checking miter of two circuits that perform the same function but using a different network structure. The miter of two networks is a network built by joining their input sets and by computing the 2-input XOR between their outputs. Furthermore, one or more injected faults (a node performing a different computation) are injected in one of the two circuits. Three types of oracles are synthesized: *addassoc*, where the algorithm should verify the validity of the associative property of addition; *multassoc*, where the two designs should be equivalent thanks to the associativity of the multiplication, and *multdistr*, to prove the distributivity of the multiplication. Each benchmark is considered with bitwidths w from 4 to 10. Consequently, each benchmark has $3 \times w$ inputs and 1 output.

The experimental results are reported in Table 5.2. The first two columns show the results of

	M/B		S/B		S/P_match_q		S/P_match_g		M/P	
	gates	qubits	gates	qubits	gates	qubits	gates	qubits	gates	qubits
addassoc4	1376	25	1029	34	1141	25	1371	22	1904	19
addassoc5	2987	36	1586	49	1798	36	1804	31	5365	24
addassoc6	2394	43	1445	58	1513	42	1729	35	8268	26
addassoc7	3243	51	1941	70	2201	50	2361	44	4383	36
addassoc8	3221	62	2018	79	2312	57	2430	49	4787	40
addassoc9	3603	70	2385	89	2453	67	2773	56	5569	42
addassoc10	4528	80	2835	97	3575	70	3549	58	6142	50
multassoc4	6682	34	2751	60	3057	34	3193	33	10834	19
multassoc5	10519	54	4811	104	5321	55	5321	55	16687	31
multassoc6	17653	93	7395	172	8565	96	8565	96	22933	53
multassoc7	25395	138	11099	240	15425	135	14607	128	37717	74
multassoc8	32443	181	13781	323	20713	179	22997	166	51757	94
multassoc9	37599	212	17881	394	34305	203	32489	200	66267	110
multassoc10	47795	289	22843	525	41825	281	41081	262	101627	143
multdistr4	4812	29	2368	54	3262	29	3694	25	5034	19
multdistr5	9011	54	4569	94	5441	54	5441	54	22717	25
multdistr6	13327	78	6092	143	7138	80	7138	80	15169	46
multdistr7	18268	110	8771	200	13849	109	13849	109	21746	63
multdistr8	26151	149	11888	276	17896	149	17520	143	39449	81
multdistr9	30427	184	14477	332	22917	182	22445	181	43819	99
multdistr10	37571	226	17808	414	29714	214	31570	219	55583	122
S/P vs M/B average results					-32.31%	-1.86%	-29.77%	-8.38%		

Table 5.2 – Comparison between *ROS* and *LHRS*.

the state-of-the-art (M/B) synthesis flow, that uses a classic k-LUT mapper and the Bennett strategy to deal with garbage results (*LHRS*).

As expected, data show that only changing the k-LUT mapper (S/B) always reduces the number of gates, paying in an increased number of qubits. On the other hand, by only applying the quantum garbage management technique (M/P), the number of qubits is always reduced, and the number of gates is increased.

In the S/P_match_q experiment ROS is used with a number of qubits that matches M/B. In most of the cases, an improvement in both qubits and gates is obtained, with the exception of *multassoc5*, *multdistr6* and *multassoc6*. For the latter cases, the SAT solver used in the quantum garbage management technique reached the limit of 50000 conflicts. For this reason, the number of qubits has been slightly increased, still obtaining in all cases a reduction in gates with respect to M/B. *ROS* in this setting reduces the number of gates of 32.31% and the number of qubits of 1.86% on average.

The S/P_match_g experiment starts from the results in S/P_match_q and tries to beat them, by decrementing the number of qubits, as long as the number of gates does not exceed the one in M/B. Also here ROS manages to obtain better results than the state-of-the-art flow both in gates and qubits. Gates are reduced of 29.77% on average, while qubits are reduced of 8.38% on average, with respect to M/B.

Most of the synthesis runs completed within a few seconds, none required more than one minute in the worst-case.

5.5 Summary

This chapter introduced the hierarchical reversible synthesis, which can compile quantum circuits implementing large Boolean functions. Section 5.1 defined the problem of oracle compilation and detailed the various steps of the state-of-the-art hierarchical compilation process. Section 5.2 explained how this thesis improved the state-of-the-art method by integrating of a k-LUT mapper that performs a decomposition tailored to reduce the final quantum circuit cost. Section 5.3 focused on a second proposed improvement: exploting SAT-based quantum memory management to reduce the number of qubits. Finally, Section 5.4 highlighted the performances of the new hierarchical synthesis method, featuring both a quantum-aware k-LUT mapper and quantum memory management, if compared to the state-of-the-art solution. Experimental results prove the ability of the proposed method to break the border of the Pareto-point synthesis results, beating the state-of-the-art framework in both numbers of qubits and gates. Gates are reduced by 29.77% while qubits are reduced by 8.38% on average. This technology can enable certain computations on systems with a constrained number of qubits, when this would not be possible using the compilation strategies available in the literature.

The memory management technique can also be applied whenever designing a quantum algorithm composed of several interconnected parts. Indeed, to deal with the complexity of a quantum algorithm, it is customary to decompose it into several parts that are then independently optimized. A DAG can be defined to describe data dependency between the parts, and it can be pebbled to find the best strategy to compose such parts together, given a limit on the number of pebbles (helper qubits). Being able of evaluating the resources required to optimally perform such composition, see e.g., the show-case in Section 5.3.3, is a fundamental step to draw conclusions on the applicability of certain quantum algorithms on a given system and to identify critical components. Future work could focus on a general toolkit to perform resource estimation related to the composition of database components, based on solving the reversible pebbling game. In this application, variations of the classical problem may provide a tighter estimation. For example, the toolkit may be capable of performing a weighted pebbling problem, where each node of the DAG is characterized by a weight representing its implementation cost and the solution is a pebbling strategy with a minimized total weight. A variation of this problem is used in Section 6.3.1 to optimize an already found pebbling strategy. Exactly solving the weighted pebbling using SAT is impractical, hence such resource estimation toolkit should rely on heuristic strategies.

6 Compiling xor-and-inverter graphs

This chapter describes hierarchical synthesis algorithms specifically designed to work on 2-LUT networks, where each node performs the 2-input XOR, the 2-input AND and inversion. This network is equivalent to an Xor-And-inverter Graph (XAG). The proposed algorithms compile into the Clifford+T universal library and focus on minimizing the following cost functions, typical of fault-tolerant quantum computing: the T-count—the number of generated T gates; the T-depth—the maximum number of T gates to be performed sequentially, also referred to as number of T-stages; and the number of qubits.

The characteristics of the initial network impact the resource footprint of the compiled circuit and this chapter elaborates on how the network could be modified to achieve better compilation results using, e.g., state-of-the-art minimization strategies [183, 184]. Finally, The trade-off between qubits and gates is explored by leveraging a memory management SAT-based technique specifically designed to work with the XAG representation. The XAG-based compilation techniques are used to synthesize quantum circuits implementing cryptographic and arithmetic logic functions with application in post-quantum cryptography and fault-tolerant quantum computing.

6.1 Methods overview

Chapter 5 presented an automatic hierarchical synthesis method that leverages network decomposition into arbitrary LUTs. Such method has the advantage of being applicable to any logic network, independently of the Boolean function implemented by its nodes. More importantly, it enables us to control the number of generated qubits: the network is decomposed into several single-output sub-networks whose results are stored into extra qubits. By controlling the size of the sub-networks, it is possible to control the number of extra qubits. However, the method is not able to efficiently optimize the gate count. Typically, when the number of qubits is heavily constrained, the number of gates significantly increases. This happens because large sub-networks will be generated and, with no control on the Boolean functions they implement, they will likely be compiled into a large circuit. In addition, LUT decomposition causes a windowing effect: parts of the networks are prevented from being synthesized together, resulting into more gates. This issue is addressed by the new LUT decomposition strategy proposed in Section 5.2, which allows some control on the grouped logic, reducing the number of gates.

In this chapter, I introduce a different approach to enable better control over all the cost functions. This approach is based on identifying repeated patterns in the XAG network, which conveniently translate into quantum circuits characterized by few gates. In particular, the method works by isolating the parts of the graph that can be implemented by one single Toffoli gate. By doing so, it identifies a direct correlation between the features of the network and the cost in terms of T gates (T-count and T-depth) and number of qubits.

The first XAG-based compilation algorithm is a method that targets the minimization of the T-count, which we published in [185]. It correlates the number of AND nodes in the XAG (*multiplicative complexity*) with the T-count. The final circuit achieves the upper-bound in the number of T gates of four times the multiplicative complexity of the input network. Indeed, each AND node can be implemented by a Toffoli gate acting on a helper line initialized at state $|0\rangle$ using the implementation in Section 4.3 (4.2), which needs 4 T gates.

The second constructive algorithm minimizes the T-depth by relating it to (i) the maximum number of levels in the graph with AND nodes, i.e., the *multiplicative depth*, and (2) the number of AND nodes in the same level sharing input signals. This second algorithm achieves a T-depth that is equal to the multiplicative depth of the graph.

Finally, the third compilation algorithm performs quantum memory management to explore the trade-off between qubits and T-count. In particular, it exploits SAT solvers to find a strategy to fit the logic into a constrained number of qubits. Section 5.3 introduced the problem of quantum memory management and presented a solution based on SAT. The idea is to enable re-utilization of helper qubits by uncomputing intermediate results, solving the reversible pebbling game. Here, this method is expanded to work at a wider level of granularity. In other words, while the previous method was enabling computation and uncomputation of every single node in the XAG separately, in this new approach selected sets of nodes are grouped together. This allows us to control the overhead in the number of gates generated when constraining the number of qubits. In addition, it simplifies the SAT problem by reducing variables and clauses allowing us to find an optimized solution with respect to the T-count. I also present in this chapter an optimization method to further improve the compilation results. The number of helper qubits can be selected as a parameter of the compilation, the SAT solver will return a valid compilation solution to not exceed the given qubit constraint, then the number of T gates is optimized.

6.2 Xor-and-inverter graphs

In classical logic synthesis, a good method is based on the synergy between data structure and algorithm, working together to minimize the target functions. The methods presented in this chapter rely on the convenient representation of the logic as an Xor-And-inverter Graph (XAG). This network has been already introduced in Section 2.1.4, but a more formal description is here provided.

An XAG is a logic network over the gate basis $\{\wedge, \oplus, \neg\}$, meaning that each node of the network either computes the 2-input AND operation, the exclusive-OR operation, i.e., the 2-input XOR, or the invertion operation $\neg x = 1 \oplus x = \bar{x}$. $\bar{x}$ denotes the Boolean complement of $x = 1 - x$, and $x^0 = \bar{x}$ and $x^1 = x$. A simple XAG computing the majority-of-three Boolean function is shown in Fig. 6.2 (a).

The logic network for an n-variable Boolean function with inputs $x_1, \dots, x_n$ is modeled as a Boolean chain with steps

$$x_i = x_{j(i)} \oplus x_{k(i)} \qquad \text{or} \qquad x_i = x_{j(i)}^{p(i)} \wedge x_{k(i)}^{q(i)}, \tag{6.1}$$

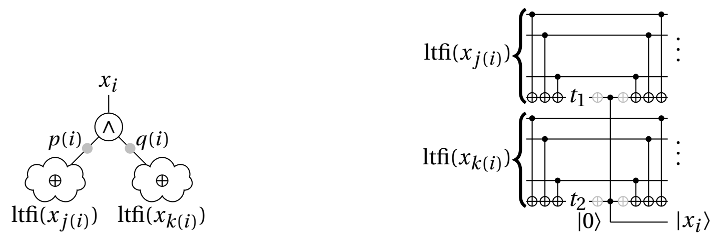
for $n < i \leq n + r$, depending on whether the step computes the XOR or the AND operation, where r is the number of steps. The constant values $1 \leq j(i) < k(i) < i$ point to input or previous steps in the chain. When a step computes the AND operation, the Boolean constants $p(i)$ and $q(i)$ are used to possibly complement the gate's fan-in. Note that complemented inputs of XOR gates can be propagated to their outputs, hence it is not necessary to define $p(i)$ and $q(i)$ for the XOR steps. The value of a single-output function is computed by the last step of the chain $f = x_{n+r}^{p}$, which may be complemented. In the case of multi-output functions, there will be a set of steps which computes the function's values: $f_o = x_o^p$ where $o \in O$—the list of all the output indices. I write $\circ_i = \wedge$, if step i computes an AND gate, and $\circ_i = \oplus$, if step i computes an XOR gate.

Definition 6.2.1 (Multiplicative complexity) *The* multiplicative complexity of the logic network *is the number of AND gates it contains:* $\tilde{c} = |\{i \mid \circ_i = \wedge\}|$. *The* multiplicative complexity of the Boolean function *is the minimum number of AND nodes required to represent it using an XAG.*

Clearly, the multiplicative complexity of a network is an upper bound on the multiplicative complexity of the Boolean function it realizes.

Every AND node acts on two multi-input parity functions. When the input to the AND node is either a primary input, another AND node, or a network's output, the arity of this function is equal to 1.

Definition 6.2.2 (Linear transitive fan-in) *Let the* linear transitive fan-in *(*ltfi*) of a step x_i in*



(a) AND step in logic network

(b) Quantum circuit construction for AND step

Figure 6.1 – Illustration of the compilation of an AND step of an XAG into a quantum circuit computing its linear transitive fan-in cones in-place using CNOT gates.

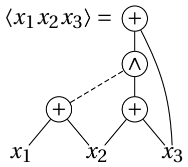


Figure 6.2 – An XAG graph representing the majority-of-three Boolean function.

the logic network be defined using the recursive function

$$\mathrm{ltfi}(x_i) = \begin{cases} \{x_i\} & \text{if } i \le n \text{ or } \circ_i = \wedge \text{ or } i \in O, \\ \mathrm{ltfi}(x_{j(i)}) \triangle \mathrm{ltfi}(x_{k(i)}) & \text{otherwise,} \end{cases} \tag{6.2}$$

where '$\triangle$' denotes the symmetric difference of two sets.

It is easy to see that all elements in ltfi(x_i) are either inputs, outputs, or steps that compute the AND function. Fig. 6.1 illustrates an AND node and its two linear transitive fan-in cones.

Example 6.2.1 *Consider the network in Fig. 6.2 (a), which implements the majority-of-three function $\langle x_1 x_2 x_3 \rangle = x_1 x_2 \vee x_1 x_3 \vee x_2 x_3$ in four steps:*

$$x_4 = x_1 \oplus x_2, \qquad x_5 = x_2 \oplus x_3,$$
$$x_6 = \bar{x}_4 \wedge x_5, \qquad x_7 = x_2 \oplus x_6.$$

For this network

$$\mathrm{ltfi}(x_4) = \{x_1, x_2\}$$
$$\mathrm{ltfi}(x_5) = \{x_2, x_3\}$$
$$\mathrm{ltfi}(x_6) = \{x_6\}$$
$$\mathrm{ltfi}(x_7) = \{x_2, x_6\}.$$

Finally, I introduce the concept of *level* in the XAG network. Every step x_i of the network, with

$1 \leq i \leq n+r$ is characterized by a quantity called *level* and defined as:

$$L(x_i) = \begin{cases} \max\limits_{t \in C}(L(t)) + 1 \text{ with } C := \text{ltfi}(x_j(i)) \cup \text{ltfi}(x_k(i)), & \text{if } i > n \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

In other words, a network's node x_i is at level $L(x_i) = l$ only if the node with the maximum level among all the ones in the linear transitive fan-in cones of x_i is at level $l-1$. This means that only AND nodes and outputs “count" to define the depth of the network, because only AND and output nodes appear in the ltfi sets.

Definition 6.2.3 (Multiplicative depth) *The* multiplicative depth *of the network is defined as*

$$\tilde{d} = \max_{n<i\leq n+r} L(x_i)$$

as the maximum level among the steps of the Boolean chain.

In addition, to provide a very compact representation for Boolean functions, XAG networks have another characteristic that makes them excellent data structures for quantum compilation: each node represents a logic function for which a convenient quantum circuit implementation exists. This allows us to recognize the existence of a dependency between the network characteristics, e.g., the multiplicative complexity, and the synthesized quantum circuit. It is indeed possible to derive an upper bound on the number of expensive gates from characteristics of the XAG.

6.3 Methods

6.3.1 Algorithm 1: minimizing the T-count

The first described algorithm achieves an upper bound on the number of T gates that is proportional to the multiplicative complexity of the input network $\tilde{c}$. Indeed, the final quantum circuit has $4\tilde{c}$ T gates.

The key insight is that each AND node in the logic network is driven by two multi-input parity functions of variables which are either inputs or other AND nodes in the lower levels of the logic network. Fig. 6.1 shows the node x_i and the two parity functions with the respective linear transitive fan-ins. The polarity variables $p(i)$ and $q(i)$ take into account possible inversion of the inputs of the AND node. The pseudo-code of the algorithm is given in Alg. 6.1. Lines 19–22 show that, at first, all the steps of the network that perform the AND function (or compute an output) are compiled using the function ‘*compute*’. Then, all the intermediate results are restored to $|0\rangle$ by uncomputing ‘*compute*’. The function *compute* (lines 1–18) builds the circuit for each step x_i as illustrated in Fig. 6.1. In particular, it identifies two signals in the ltfi cones that are not shared between the cones, namely t_1 and t_2. Then, the parity functions are computed in-place onto the qubits corresponding to t_1 and t_2. In the next steps, the complemented edges are evaluated and X gates are applied if necessary (see Fig. 6.1). In lines 12–13 the step x_i is finally computed on a new qubit, using a CNOT gate (XOR output) or the implementation of the AND node described in (4.2), which has T-count equal to 4 and T-depth equal to 2. Finally, the parity functions are uncomputed.

Algorithm 6.1 Low T-count compilation algorithm.

```
Input: Logic network with gates $x_{n+1}, \dots, x_{n+r}$
Output: Quantum circuit for $U_f$
 1: function COMPUTE($x_i^p$)
 2:     $j \leftarrow j(i)$, $k \leftarrow k(i)$
 3:     $L_1 \leftarrow \mathrm{ltfi}(x_j)$, $L_2 \leftarrow \mathrm{ltfi}(x_k)$
 4:     $x_i \leftarrow$ request_helper
 5:     if $L_1 \subseteq L_2$ then swap $L_1 \leftrightarrow L_2$ and $p \leftrightarrow q$
 6:     let $t_1$ be some element in $L_1 \setminus L_2$
 7:     let $t_2$ be some element in $L_2$
 8:     CNOT($x, t_1$) for all $x \in L_1 \setminus \{t_1\}$
 9:     CNOT($x, t_2$) for all $x \in L_2 \setminus \{t_2\}$
10:     if $p(i)$ then NOT($t_1$)
11:     if $q(i)$ then NOT($t_2$)
12:     if $o_i = \wedge$ then AND$_{T-depth=2}$($t_1, t_2, x_i$)
13:     else XOR($t_1, t_2, x_i$)
14:     if $i \in O$ and $p$ then NOT($x_i$)
15:     if $p(i)$ then NOT($t_2$)
16:     if $q(i)$ then NOT($t_1$)
17:     CNOT($x, t_2$) for all $x \in L_2 \setminus \{t_2\}$
18:     CNOT($x, t_1$) for all $x \in L_1 \setminus \{t_1\}$
19: for $i = n+1, \dots, n+r$ where $\circ_i = \wedge$ or $i \in O$ do
20:     compute($x_i^p$)
21: for $i = n+r, \dots, n+1$ where $\circ_i = \wedge$ do
22:     compute$^\dagger$($x_i^p$)
```

Note that I assume that $L_1 \neq L_2$. If this is not the case, it means that the parity functions that are input to the AND node are equal, making AND node itself redundant. Also, note that the intersection of L_1 and L_2 may not be empty. Since the value of L_1 is computed in-place on some signal $t_1 \in L_1$, then it must be ensured that $L_1 \nsubseteq L_2$. If the latter condition applies, it is sufficient to swap L_1 and L_2.

In addition, when $L_2 \subseteq L_1$, the value computed by L_2 could be reused to compute L_1. This is achieved by modifying the elements in L_1 such that $L_1 = (L_1 \setminus L_2) \cup \{x_k\}$. An example is shown in Fig. 6.3. In this case $\mathrm{ltfi}(x_j)$ includes $\mathrm{ltfi}(x_k)$ and $\mathrm{ltfi}(x_j) \setminus \mathrm{ltfi}(x_k) = \{t_0\}$. This leads to a reduction in the number of CNOT operations.

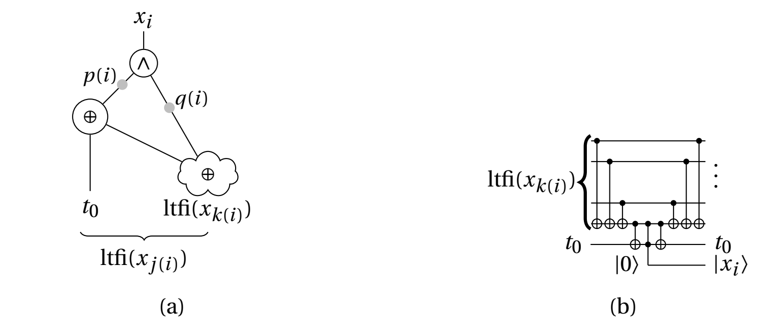
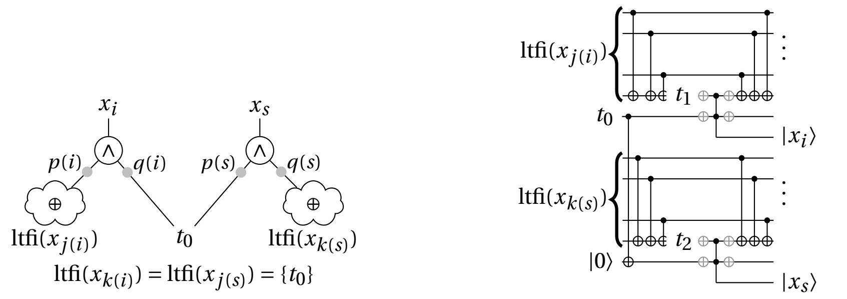


Figure 6.3 – Example in which one transitive fan-in is included in the other and the computed intermediate value can be reused.



(a) A level with two AND nodes and a shared input

(b) Quantum circuit construction for the level.

Figure 6.4 – Illustration of how algorithm 6.2 compiles one level with two AND nodes, x_i and x_s, into a quantum circuit with one single T-stage.

6.3.2 Algorithm 2: minimizing the T-depth

The second algorithm targets the reduction of the T-depth. Compared to the previous one, it uses implementation (4.4) of the AND operation that has 4 T gates, 4 qubits and 1 T-stage. I refer to $X_l = \{x_i \mid L(x_i) = l\}$, as the set of all the nodes at level l. The key idea is that if two AND nodes in the same level do not share any of their input in the ltfi sets, then they can be computed with only one T-stage using implementation (4.4). Obviously, this is not always the case, as AND nodes often share the same inputs. To overcome this problem, the algorithm copies every overlapping set of inputs on a new helper qubit. This procedure, described in Alg. 6.2, obtains circuits with a number of T-stages equal to the multiplicative depth $\tilde{d}$ of the networks. While the previously described algorithm proceeds in topological order, this one proceeds level by level (see lines 11–15). For each level, the function *copy_overlaps* will assign to each node a set of two qubits on which the parities of the two fan-in cones are computed. If the node shares some inputs with another, a new qubit will be assigned to compute the corresponding parity function. This means that if a node $x_i \in X_l$ has inputs t_1, t_3, t_5 (on qubits q_1, q_3, q_5) in common with node $x_j \in X_l$, then a new qubit q_i will be used as target of three CNOT gates with the shared input qubits as controls. As it can be seen in line 12, the copies are

Algorithm 6.2 Low T-depth compilation algorithm.

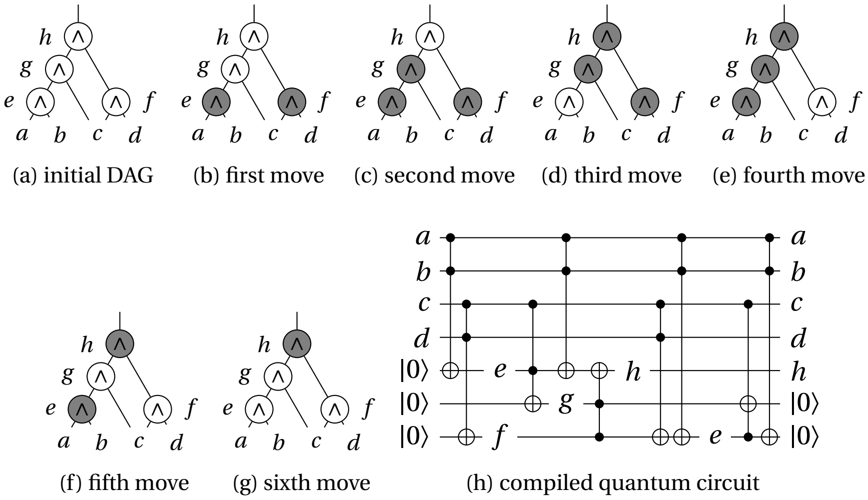
```
Input: Logic network with gates $x_{n+1}, \ldots, x_{n+r}$
Output: Quantum circuit for $U_f$
 1: function COMPUTE_ON_COPIES($x_i^p$, $CP$)
 2:     $t_1, t_2 \leftarrow CP[i]$
 3:     $x_i \leftarrow$ request_helper
 4:     if $p(i)$ then NOT($t_1$)
 5:     if $q(i)$ then NOT($t_2$)
 6:     if $o_i = \wedge$ then AND$_{T-depth=1}(t_1, t_2, x_i)$
 7:     else XOR($t_1, t_2, x_i$)
 8:     if $i \in O$ and $p$ then NOT($x_i$)
 9:     if $p(i)$ then NOT($t_2$)
10:     if $q(i)$ then NOT($t_1$)
11: for $l = 1, \ldots, \tilde{d}$ do
12:     $CP \leftarrow$ copy_overlaps($X_l$)
13:     for $x_i^p \in X_l$ do
14:         compute_on_copies($x_i^p$, $CP$)
15:     copy_overlaps$^\dagger$($X_l$)
16: for $l = \tilde{d}, \ldots, 1$ do
17:     for $x_i^p \in X_l$ where $i \notin O$ do
18:         compute$^\dagger$($x_i^p$)
```

performed before compiling any of the AND nodes in the level, thus allowing the actual AND implementations to act on non-overlapping qubits, resulting in a single T-stage. Once the copies are being computed each node is passed to the function *compute_on_copies* (lines 1–10) which uses the qubits associated by the mapping CP to each fan-in parity function as controls to compute the AND. Once all AND nodes in the level are computed, the parities are uncomputed (lines 15). Finally the levels in the XAG are uncomputed from higher to the lower level. Every node, independently from having shared fan-ins can be uncomputed without using copies (lines 16–end), using the function *compute* as defined in Alg. 6.1. An illustrative example is shown in Figure 6.4, where the algorithm is applied to a simple level $X_l = x_i, x_s$ with one overlapping input t_0, such that $\text{ltfi}(x_{j(i)}) \cap \text{ltfi}(x_{k(s)}) = \{\}$ and $\text{ltfi}(x_{j(s)}) = \text{ltfi}(x_{k(i)}) = \{t_0\}$. The figure shows how the overlapping input is copied to a new qubit before computing the parity functions: then the two AND can be computed in parallel with a T-depth equal to 1.

6.3.3 Algorithm 3: minimizing the number of qubits

All the algorithms described so far compute and uncompute every AND node at most once, and the compiled circuit is uniquely determined by the features of the input network. This section presents a method that, instead, allows us to explore the solution space, by computing and uncomputing nodes several times.

The third algorithm seeks the best strategy to uncompute the intermediate results in order to optimize the memory usage. The problem is equivalent to the *reversible pebbling game*



(a) initial DAG (b) first move (c) second move (d) third move (e) fourth move

(f) fifth move (g) sixth move (h) compiled quantum circuit

Figure 6.5 – Illustration of a pebbling strategy for the input DAG (a) using 3 pebbles and 6 moves (b)-(g), and the corresponding compiled reversible circuit of Toffoli gates (h).

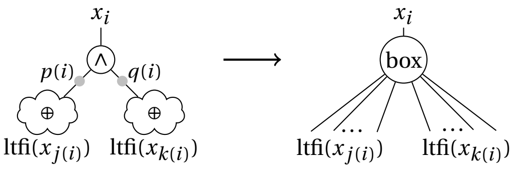


Figure 6.6 – Illustration of how sections of the XAG are compressed in a box node of the abstract network.

introduced in Section 5.3.

Example 6.3.1 *Fig. 6.5 illustrates how a network with only AND nodes can be compiled as a reversible network of Toffoli gates out of a pebbling solution with 3 pebbles and 6 steps. Note that the final circuit will use only 2 helper qubits, that is the number of pebbles used, minus the number of outputs. The overall width will be equal to 7: the number of inputs plus the number of pebbles.*

XAGs are DAG in which each node computes the AND or the XOR function. It follows that it is possible to play the reversible pebbling game directly on the XAG. Nevertheless, this does not exploit the structural properties of the XAG. For this reason, a different DAG is constructed from the XAG, called *abstract graph*. Each AND node (and its two input parity functions) corresponds to a 'box' node of the abstract graph, as shown in Fig. 6.6. Once a strategy for pebbling the abstract graph is found, each time a pebble is placed on a box node which compresses x_i the '*compute*(x_i)' function will be called, while whenever a pebble is removed from a node, the '*compute*$^\dagger(x_i)$' function will be called to uncompute the node.

Optimizing the pebbling solution

While the XAG is compressed into the abstract graph, some information about the number of quantum gates required to compute each node is lost. Indeed, the strategy found by playing the reversible pebbling game on the abstract graph would not take into account the fact that each box node requires a different number of gates to be performed. In addition, the SAT encoding of the standard reversible pebbling game does not include any clause that controls the number of moves, which reflects in the number of generated T gates. An optimization step is introduced to overcome both problems.

The key idea is that it is possible to associate a weight with each *box* node of the abstract graph w_v, which is equal to the number of inputs to the node itself. Indeed, the number of inputs are related to the number of CNOT gates that are needed to compute the parity functions "hidden" in the compressed node.

If every node v is characterized by a weight w_v, it is possible to define the following problem.

Problem 6.3.1 (Weight-bounded pebbling) *Given a DAG $G = (V, E)$, K steps and a total weight W, find a reversible pebbling strategy $P = (P_0, \ldots P_{k+1})$ such that $\sum_{i=1}^{K} \sum_{v \in P_i} w_v \leq W$.*

A new set of variables are defined for the SAT encoding of this problem: *activation variables* $a_{v,i}$. For $v \in V$ and $0 < i \leq K$, those are Boolean variables that evaluate to true if the node v has changed its state at time i. Once a weight-agnostic solution has been found, the following quantity represent the total weight of the strategy:

$$W_s = \sum_{i=1}^{K} \sum_{v \in V} w_v a_{v,i} \tag{6.4}$$

The SAT solver is then asked to solve problem 6.3.1 with a total weight $W = W_s - 1$. This procedure is repeated until the solver returns *unsat* or hits a timeout.

As shown in the result section, this optimization procedure succeeds at reducing the number of T gates with respect to the initial solution. This result can be achieved even if every node has weight equal to one. Indeed, the optimization introduces a cardinality constraint on the activation variables, hence eliminates all the pebbling moves that are not fundamental to terminate the game. If the weights are set to reflect the actual size of the parity functions, then the number of CNOT in the solution can be reduced.

6.4 Results

This section reports the statistics of quantum circuits generated by the described XAG-based algorithms. Results are presented for two publicly available benchmark suites, including arithmetic, cryptographic, e.g., AES, and floating point operations with application in post-quantum cryptography and fault-tolerant quantum computing.

The first benchmark contains the best-known logic networks, in terms of multiplicative complexity and depth, collected by the *Computer Security Resource Center* (CSRC) at the *National Institute of Standards and Technology* (NIST) and by the University of Yale[1]. The benchmark includes: (i) finite field multiplication in $GF(2^6)$ using irreducible polynomial $x^6 + x^3 + 1$ (mx6x31), multiplication in $GF(2^7)$ using irreducible polynomial $x^7 + x^4 + 1$ (mx7x41) and using $x^7 + x^3 + 1$ (mx7x31); (ii) binary multiplication with different input sizes n (bm_n); (iii) a 16-bit and a 8-bit S-box (s16, s8); (iv) finite field multiplication in $GF(2^8)$ using the AES polynomial $x^8 + x^4 + x^3 + x + 1$ (x8x4x31).

In addition, the method is evaluated on a set of circuits used in the context of *Multi-Party Computation* MPC and *Fully Homomorphic Encryption* FHE, collected by the Department of Electrical Engineering (ESAT) at KU Leuven[2] optimized for the number of AND gates using the technique proposed in [183] and available online[3]. The benchmark includes: (i) block ciphers DES in its expanded and non-expanded variant (the latter meaning that the input key is assumed non-expanded); (ii) block cipher AES with 128, 192 and 256 key length; (iii) cryptographic hash functions MD5, Keccak, SHA-256 and SHA-512; (iv) arithmetic functions such as adders, multipliers, and comparators; (v) IEEE floating point operations.

6.4.1 Improving the T-count versus T-depth

Table 6.1 shows the synthesis result of the first two proposed algorithms. Alg. 6.1 minimizes the T-count, while Alg. 6.2 minimizes the T-depth without increasing the number of T gates, but relying on an increased number of additional qubits. The number of T gates achieved is equal to 4 times the multiplicative complexity of the network for both algorithms. The second algorithm reduces the T-depth to be equal to the multiplicative depth of the network. The last two columns of Table 6.1 show a comparison between the two algorithms: the percentage improvement in T-count and in number of qubits of Alg. 6.2 with respect to Alg. 6.1.

6.4.2 Qubits/T-count trade-off

This section reports the results obtained by the third algorithm to control the memory resources during the compilation of the logic design. The method allows us to force the compilation to synthesize a circuit with a limited number of helper qubits. Fig. 6.7 shows different compilation results obtained setting the number of available helper qubits to different values, for a selection of designs. The plots show on the x axis the number of qubits, and on the y axis the obtained T-count. For every fixed number of qubits two points are reported: the non-optimized and the optimized results. The latter obtained by running a post-optimization procedure encoded as a SAT problem on the initial (non optimized) result. It can be seen how the procedure allows us to choose between different qubit/T-count trade-off solutions and

[1] http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html

[2] https://homes.esat.kuleuven.be/~nsmart/MPC/

[3] https://github.com/lsils/date2020_experiments

						Alg.1		Alg.2		Comparisons	
benchmark	I	O	AND	XOR	Tc*	Td	Q	Tdld	Qld	%Td	%Q
mcustom	16	8	27	79	108	8	51	1	116	87.50	-127.45
mx6x31	12	6	27	30	108	6	45	1	112	83.33	-148.89
mx7x41	14	7	40	44	160	7	61	1	164	85.71	-168.85
mx7x31	14	7	40	45	160	7	61	1	164	85.71	-168.85
s16	17	16	113	333	452	48	146	8	283	83.33	-93.84
bm-10	20	19	52	102	208	12	89	1	218	91.67	-144.94
bm-11	22	21	78	108	312	12	119	1	322	91.67	-170.59
bm-12	24	23	81	126	324	12	126	1	332	91.67	-163.49
bm-15	30	29	117	195	468	18	174	1	482	94.44	-177.01
bm-20	40	39	208	314	832	24	285	1	850	95.83	-198.25
bm-30	60	59	351	687	1404	36	468	1	1448	97.22	-209.40
bm-40	80	79	624	1079	2496	48	781	1	2554	97.92	-227.02
bm-50	100	99	676	1847	2704	72	873	1	2774	98.61	-217.75
bm-60	120	119	1053	2253	4212	72	1290	1	4284	98.61	-232.09
bm-70	140	139	1432	2985	5728	72	1709	1	5856	98.61	-242.66
bm-80	160	159	1872	3494	7488	96	2189	1	7582	98.96	-246.37
bm-90	180	179	1989	4561	7956	126	2346	1	8104	99.21	-245.44
bm-100	200	199	2704	5143	10816	144	3101	1	10950	99.31	-253.11
s8	9	8	32	81	128	28	49	6	63	78.57	-28.57
x8x4x31	16	8	48	69	192	8	72	1	194	87.50	-169.44
DES-exp	832	64	9205	13136	36820	2070	10101	214	10352	89.66	-2.48
DES-non-exp	128	64	9048	13092	36192	2186	9240	202	9464	90.76	-2.42
adder-32bit	64	33	32	150	128	33	129	32	130	3.03	-0.78
adder-64bit	128	65	64	284	256	65	257	64	258	1.54	-0.39
comp-32bit-lt	64	1	92	95	368	21	156	20	182	4.76	-16.67
comp-32bit-lteq	64	1	92	97	368	20	156	19	182	5.00	-16.67
md5	512	128	9367	29729	37468	7561	10007	1283	10619	83.03	-6.12
mult-32x32	64	64	1689	4723	6756	324	1816	64	1816	80.25	0.00
Keccak-f	1600	1600	38400	115200	153600	30209	41600	24	44798	99.92	-7.69
aes-128	256	128	6400	28176	25600	874	6976	60	7133	93.14	-2.25
aes-192	320	128	7168	32080	28672	830	7808	72	7870	91.33	-0.79
aes-256	384	128	8832	39008	35328	900	9536	84	9598	90.67	-0.65
sha-256	768	256	22573	109746	90292	14411	23597	1607	23597	88.85	0.00
sha-512	1536	512	57947	284286	231788	40172	59995	3304	59995	91.78	0.00
FP-add	128	64	5346	5629	21384	2460	5538	235	5541	90.45	-0.05
FP-div	128	64	70599	44959	282396	25649	70792	3604	72563	85.95	-2.50
FP-eq	128	64	315	356	1260	10	506	9	526	10.00	-3.95
FP-f2i	64	64	1458	1421	5832	593	1586	94	1683	84.15	-6.12
FP-mul	128	64	18874	10290	75496	1988	19066	118	20907	94.06	-9.66
FP-sqrt	64	64	76925	57165	307700	44782	77054	6498	79589	85.49	-3.29

* Both algorithms achieve the same T-count.

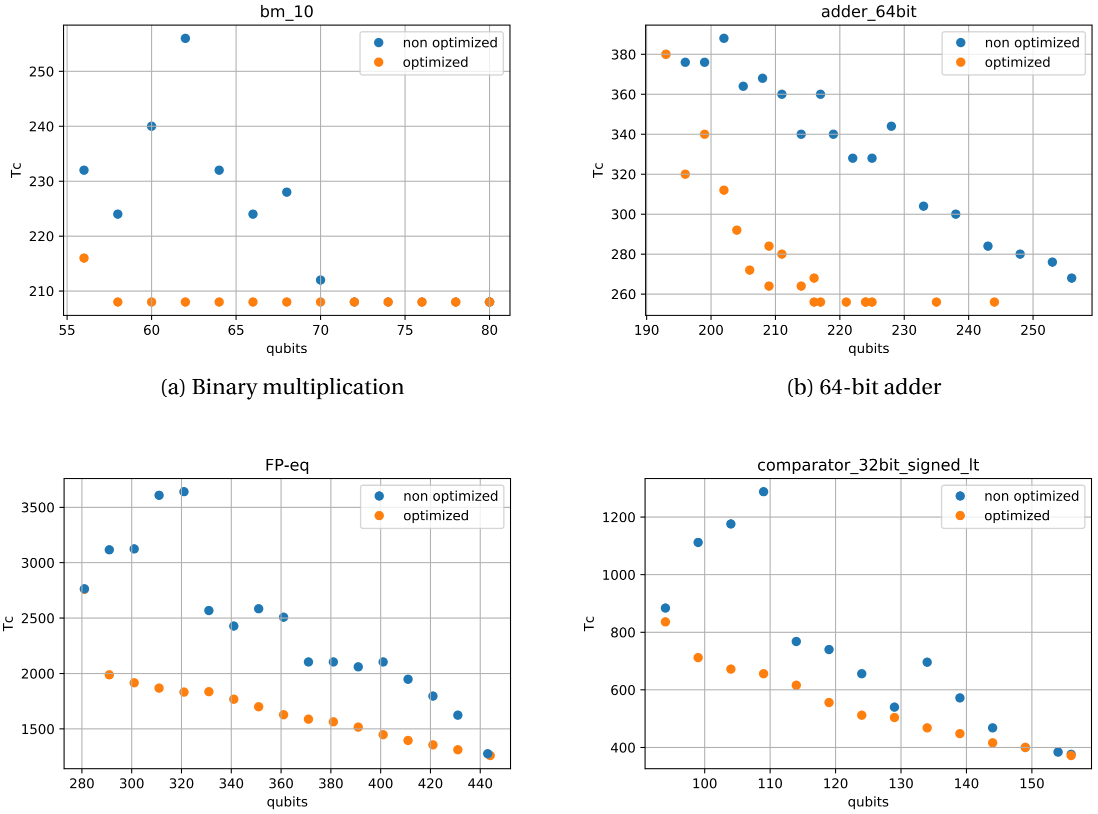
Table 6.1 – Compilation results for several arithmetic, floating point and cryptographic designs.

how the optimization achieves Pareto-optimal results.

6.4.3 Discussion

The first two techniques achieve results that are predictable by inspecting the characteristics of the logic network. In details, given a logic network characterized by a multiplicative complexity $\tilde{c}$, i.e., the number of AND nodes, and by a multiplicative depth:

- both algorithms achieve a T-count equal to $4\tilde{c}$;



(a) Binary multiplication

(b) 64-bit adder

(c) IEEE floating point equality

(d) 32-bit comparator

Figure 6.7 – Optimized and non-optimized pebbling results using different number of pebbles for selected logic networks.

- Alg. 6.2 achieves a T-depth equal to the multiplicative depth;
- the qubit overhead to achieve such T-depth depends on the number of shared inputs in the linear transitive fan-ins of the AND nodes in a level.

This suggests that improving a network with respect the named parameters can strongly and positively impact the synthesized quantum circuits, e.g., as done in [184], to reduce the T-depth by reducing the multiplicative depth of the network.

Inspecting the results shown in the comparison columns of Table 6.1 reveals a trade-off between T-depth and number of qubits. Indeed, while Alg. 6.1 is far from achieving the T-depth performances of Alg. 6.2, it requires fewer qubits. There are two reasons for the increase in qubits which characterizes Alg. 6.2. The first one is that it employs the AND implementation characterized by a single T-stage and presented in Section 4.3 (4.4), which requires one qubit more than the implementation (4.2) used by Alg. 6.1. This means that the compilation will request this extra qubit whenever a AND node is computed. In addition, the implementation of AND nodes used by the second algorithm is characterized by a T gate applied to the controls, as well as to the target qubit. For this reason, if two AND nodes share the same input signal, the corresponding quantum circuit will have a T-depth equal to 2, as each AND implementation will add a T gate to the shared qubit. If all the AND nodes at the same level of an XAG do not share any input, they can be computed within a single T-stage. In order to achieve this result, the second algorithm "copies" inputs that are shared among more AND nodes in a level on new qubits. Hence, the compilation will request a new qubit whenever inputs are shared among AND nodes at the same level in the XAG. In conclusion, summing the number of AND nodes in a level with the number of shared inputs among them, one obtains the number of helper qubits claimed to compile that level. Since helper qubits are cleaned-up after all the nodes in the level are computed, the level for which this amount is greater will dominate and give the total number of helper qubits for the synthesis of the entire network. Further details on the algorithm, including detailed pseudo-code, can be found in Section 6.3. Table 6.1 reports the two extremes that can be reached using XAG-based constructive algorithms. It is also possible to obtain results "in-between", i.e., a smaller improvement in T-depth and a smaller qubit overhead with respect to Alg. 6.2, e.g., by modifying Alg. 6.1 to use the implementation with T-depth equal to one. In addition, as the connectivity of each AND node in a level has an impact on the T depth, different results can be found by changing how the level of each node is computed.

The third algorithm focuses on exploring the trade-off between T-count and number of qubits. Fig. 6.7 shows how the method is capable of providing different compiled solutions, by taking the number of helper qubits as a parameter. The method finds the best way of reusing memory space, by solving the reversible pebbling game. This is a global problem, hard to approximate and decompose [174], hence difficult to be tackled by heuristic techniques. Here, the problem is encoded as a SAT problem and solved globally, returning a valid memory clean-up strategy that guarantees the upper bound on the number of helper qubits while also aiming to minimize the T-count.

Fig. 6.7 shows non-optimized versus optimized pebbling solutions. The non-optimized solution is provided by the SAT solver without any constrains on the number of T-count generated. The optimized solution is obtained starting from the initial solution and running optimization rounds, which iteratively add clauses to the SAT problem to minimize the T-count. The more time is spent in the optimization procedure the better the solution. The optimized points shown in Fig. 6.7 are either optimal or the best result found after 1 and a half hours of running the optimization procedure on a machine with two Intel Xeon E5-2680 v3 (Haswell) CPUs with 2.5 GHz clock frequency and 16 GB of main memory.

The optimization procedure removes unnecessary steps that the solver may insert in the solution. Indeed, none of the clauses used to encode the problem prevents the solver to uncompute nodes even if the limit in pebbles is not reached. Preventing this at the encoding level requires a non-practical increase in the size of the SAT problem. The optimization "reveals" the trade off between qubits and T-count, showing Pareto-optimal solutions.

6.5 Summary

In this chapter, I introduced three different algorithmic methods to perform hierarchical reversible synthesis (see the overview in Section 6.1) of a particular type of LUT network, i.e., the Xor-And-inverter Graph (XAG) formalized in Section 6.2. The three methods were explained in detail in Section 6.3. Section 6.4 reported the statistics of the quantum circuits for the selected benchmark designs and compared the performances of the different techniques.

The described algorithms are all based on deriving, from the initial XAG, an abstract graph that groups together the logic that can be compiled using one Toffoli gate and some CNOT gates. By doing so, they provide a direct correspondence between the T-count and the T-depth of the compiled quantum circuit and some structural properties of the XAG. In particular, Alg. 6.1 and Alg. 6.2 achieve a T-count that is four times the *multiplicative complexity* of the graph, while Alg. 6.2 achieves a T-depth that is equal to the *multiplicative depth* of the graph. This suggests that future works should focus on optimizing such graph's properties, providing an optimization toolkit that leverages many logic network optimization techniques such as rewriting, refactoring, and resubstitution, targeting the reduction of both the multiplicative complexity and the multiplicative depth. Also, the distribution of the XOR nodes in the graph plays an important role by affecting the number of CNOT gates in the compiled quantum circuits, the reduction of which can also be tackled in such a toolkit.

A logic synthesis toolkit already exists that only targets the multiplicative complexity of XAG graphs [183]. The experiments in this chapter provide compilation results for XAGs already optimized by such a toolkit.

7 Single-target gate decomposition

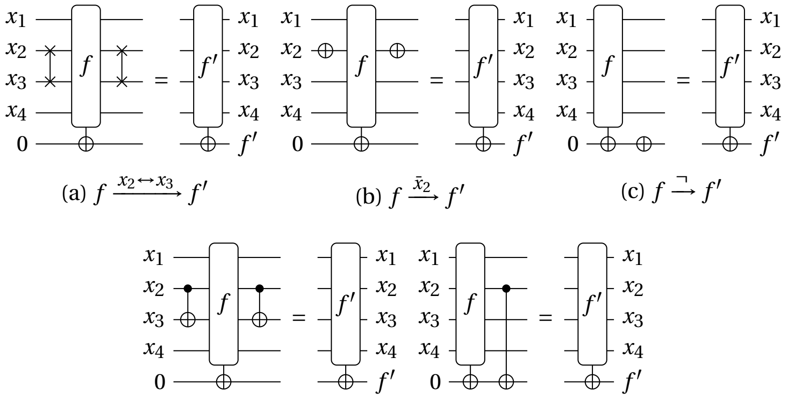
The previous chapters have shown how hierarchical reversible synthesis methods use, as intermediate representation, a reversible circuit built using single-target gates. Such reversible circuit is derived from a k-LUT decomposition of the initial network representing the Boolean function to be compiled into a quantum oracle. This chapter discusses the problem of decomposing each single-target gate into the Clifford+T universal quantum gate library.

Problem 7.0.1 *Given a single-target gate* $\mathrm{T}_c(C,t)$*, where* $c : \mathbb{B}^k \to \mathbb{B}$ *is the* control function, $C = \{c_1, \ldots, c_k\}$ *is the set of* control lines *and* $t \notin C$ *is the target line, and given a set of helper qubits* X_{clean}*, find a Clifford+*T *network that realizes the function* c *on line* t *and restores the initial values on all other lines.*

Two cases are discussed separately. The first case considers single-target gates controlled by Boolean functions with a number of input variables smaller than or equal to 5. In this case, it is possible to precompute optimal results and store them into a database compressed using spectral classification. Section 7.1 describes the characteristics of the proposed database and specifies the algorithms used to generate its entries. The second case considers methods which are capable of decomposing boolean functions independently from their number of inputs. One approach is to apply ESOP-based decomposition. A study on how advanced methods for the synthesis of ESOP expressions can be used in the compilation of oracle circuits is presented in Section 7.2.1. Another approach, applicable to larger Boolean functions, involves using successive k-LUT mappings to fully distribute the computation over all the available helper qubits, presented in Section 7.3.

7.1 Precomputed quantum circuits

This section considers the problem of synthesizing quantum circuits implementing single-target gates characterized by a control function with maximum 5 inputs. Optimized implementations can be precomputed and collected into a database that can serve two purposes:



(a) $f \xrightarrow{x_2 \leftrightarrow x_3} f'$ (b) $f \xrightarrow{\bar{x}_2} f'$ (c) $f \xrightarrow{\neg} f'$

(d) $f \xrightarrow{x_2 \oplus x_3} f'$ (e) $f \xrightarrow{\oplus x_2} f'$

Figure 7.1 – Reversible circuits implementing the five spectral invariant operations.

(1) derive a quantum circuit for any Boolean function up to 5 inputs and (2) be used in the k-LUT-based hierarchical synthesis process.

In general, such a precomputation approach requires a large amount of memory, since the number of Boolean functions grows double-exponentially. For example, there are 256 3-input Boolean functions, 65 536 4-input Boolean functions, 4 294 967 296 5-input Boolean functions, and more than 10 quintillion (10^{19}) 6-input Boolean functions. However, Boolean function classification, which has been introduced in Section 2.2, can be used to reduce the memory requirements and compress the size of the database. Indeed, there are only 2, 3, 8, and 48 spectral equivalent classes for Boolean functions over 2, 3, 4, and 5 inputs, respectively.

The database contains Clifford+T quantum circuits, optimized for the T-count, for the T-depth or for the number of qubits, according to the cost functions of fault-tolerant quantum computing.

7.1.1 Spectral equivalence in quantum compilation

In this section, I explain how a database containing an optimized quantum circuit for each spectral equivalent class can be used to compile any Boolean function, without requiring any extra T gates or qubits than the spectral equivalent representative in the database.

Theorem 7.1.1 *Given two Boolean functions f and f' such that $f \doteq f'$, then $t(f) = t(f')$, where $t(f)$ maps the Boolean function f into the minimum number of T gates required by the quantum oracle implementing f.*

To prove this theorem, recall Definition 2.2.2: two n-input Boolean functions f and g are *spectral-equivalent* ($f \doteq g$), if there exists a set of spectral invariant operations $o_1, \ldots, o_k$ from

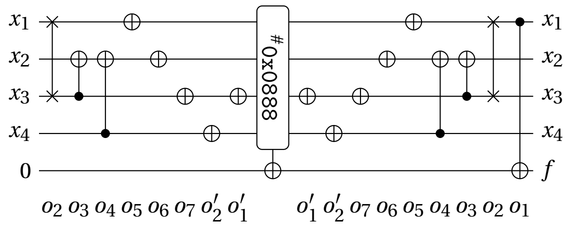

x_1
x_2
x_3
x_4
0
#0x0888
x_1
x_2
x_3
x_4
f
$o_2\ o_3\ o_4\ o_5\ o_6\ o_7\ o'_2\ o'_1$
$o'_1\ o'_2\ o_7\ o_6\ o_5\ o_4\ o_3\ o_2\ o_1$

Figure 7.2 – Synthesized reversible circuit for the function f, obtained performing all the spectral operations to retrieve f from the optimal implementation available in the database f_d ($^{\#}$0x0888).

Definition 2.2.1 such that

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} g.$$

Since all five spectral invariant operations can be implemented by X (negation) and CNOT (controlled-negation) gates, hence without any T gates, it follows that, for any invariant operation o, if $f \xrightarrow{o} g$ then $t(f) = t(g)$.

The spectral invariant operations are illustrated by the reversible circuits in Fig. 7.1, which clearly shows how none of them require T gates. The circuit in Fig. 7.1 (a) performs a SWAP gate, which can be implemented using three CNOT gates.

Consider the problem of compiling a quantum circuit implementing f starting from a database entry which computes f_d, such that $f \doteq f_d$. As discussed in Section 2.2, it is possible to derive a canonical representative $\hat{f} \in [f]$ for any Boolean function f, by applying the spectral invariant operations directly to the Rademacher-Walsh spectrum of f [130, 135]. This procedure can be efficiently implemented and applied to both f and f_d: it returns two sequences of operations $o_1, \ldots, o_k$ and $o'_1, \ldots, o'_l$, such that $f \xrightarrow{o_1} \cdots \xrightarrow{o_k} \hat{f}$ and $f_d \xrightarrow{o'_1} \cdots \xrightarrow{o'_l} \hat{f}$. Since $f \doteq f_d$ the procedure will return the same representative $\hat{f}$. Therefore, since all operations are self-inverse, it is possible to transform f into f_d:

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} \hat{f} \xrightarrow{o'_l} \cdots \xrightarrow{o'_1} f_d.$$

Example 7.1.1 *Assume we want to map the single target gate* $\mathrm{T}_f(\{x_1, x_2, x_3, x_4\}, x_5)$ *with control function* $f = {}^{\#}\mathtt{acab}$*. Knowing that* $f \in [{}^{\#}\mathit{0888}]$*, let* $f_d = {}^{\#}\mathit{0888}$ *be the database entry for this class. The spectral canonization algorithm in [130] finds the operation sequences to transform* f *and* f_d *in a canonical representative function of the equivalent class ($^{\#}$8880).*

$$\begin{aligned} f &\xrightarrow{\oplus x_1} {}^{\#}\mathit{0601} \xrightarrow{x_1 \leftrightarrow x_3} {}^{\#}\mathit{1401} \xrightarrow{x_2 \oplus x_3} {}^{\#}\mathit{4401} \\ &\xrightarrow{x_2 \oplus x_4} {}^{\#}\mathit{1101} \xrightarrow{\bar{x}_1} {}^{\#}\mathit{2202} \xrightarrow{\bar{x}_2} {}^{\#}\mathit{8808} \xrightarrow{\bar{x}_3} {}^{\#}\mathit{8880} \\ f_d &\xrightarrow{\bar{x}_3} {}^{\#}\mathit{8088} \xrightarrow{\bar{x}_4} {}^{\#}\mathit{8880} \end{aligned}$$

Respectively, $o_1 = \oplus x_1, o_2 = x_1 \leftrightarrow x_3, o_3 = x_2 \oplus x_3, o_4 = x_2 \oplus x_4, o_5 = \bar{x}_1, o_6 = \bar{x}_2, o_7 = \bar{x}_3$ *and* $o'_1 = \bar{x}_3, o'_2 = \bar{x}_4$*. We obtain a circuit as illustrated in Fig. 7.2. First, the operations* o'_1, o'_2 *are* *applied to transform* $f_d \rightarrow {}^{\#}8880$ *then* $o_7, \ldots, o_1$ *to transform* ${}^{\#}8880 \rightarrow f$.

7.1.2 Algorithms for the synthesis of the database

The database contains three entries for each representative of the spectral equivalent classes with 4 and 5 inputs. One entry is optimized for the number of qubits, one for the T-count and one for the T-depth. Each different representation was synthesized using a different procedure. In this section, I describe the three procedures selected to synthesize each different quantum circuit in the database.

ESOP-based algorithm to optimize the number of qubits

An ESOP-based procedure has been used to generate the database circuits optimized for the number of qubits. Indeed, ESOP decomposition enables us to generate quantum circuits with at most one helper qubit. The procedure also targets, as a second objective, the minimization of the T-count.

For a spectral class $[f]$, the following procedure is used to find a database entry for this class:

1. Pick any $g \in [f]$.
2. Use ESOP-based decomposition to generate a network of multiple-controlled Toffoli gates, as described in Section 4.2.1.
3. Optimize the multiple-controlled Toffoli network in order to reduce the T-count. This step includes the application of some optimization properties that were initially proposed in [3] and that are reported in this thesis in Section 8.1.1. These properties target the minimization of the number of controls of each multiple-controlled Toffoli, which corresponds to a T-count reduction. Some spectral invariant operations may be performed to enable the application of such properties, leading to a different $g \in [f]$.
4. Map each multiple-controlled Toffoli gate into the Clifford+T library using Barenco decomposition [89] and relative-phase Toffoli mapping [90], enabling only one clean helper line.
5. Optimize the resulting Clifford+T network using the open-source optimization script *T-par*, which has been proposed in [108]. It exploits the phase polynomial representation to reduce the number of T gates, while it uses matroid partitioning to reduce the T-depth. The script is applied repeatedly until no further improvement can be obtained.

XAG-based algorithm to optimize the T-count

The procedure employed to generate circuits with minimal T-count is the constructive hierarchical method based on Xor-And-inverter Graphs XAGs [185] described in Section 6.3.1.

The selected compilation algorithm is capable of generating a quantum circuit using $O(N)$ qubits and $O(N)$ gates, where N is the size of the XAG. The method guarantees an upper bound on the T-count that depends on the *multiplicative complexity* $\tilde{c}$ of the Boolean function representation. Recall that the multiplicative complexity, defined in Section 6.2, represents the number of AND nodes in the network. In particular, the T-count is at most equal to $4 \times \tilde{c}$.

The algorithm takes advantage of the low T-count implementation of the AND function and of measurement-based uncomputation shown in (4.2), (4.3) and proposed in [166, 167]. The pseudo-code is given in Alg. 6.1, while here a high-level description of the procedure is provided.

For each AND node in the XAG, Alg. 6.1 calls the function `compute`, which:

1. finds the parity functions that are input to the AND node;
2. computes the parity functions in-place using CNOT gates;
3. implements inversions using X gates;
4. implements the AND node using the implementation (4.2).

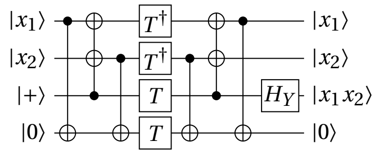
Once the outputs are computed, the iteration is repeated, this time performing the function `compute`† to restore all helper qubits to their initial state. This function will use the implementation (4.3), which performs measurement based uncomputation. Hence, it does not require any T gate.

Please refer to Algorithm 6.3.1 and to Section 4.3 for a more detailed description of the algorithm and the quantum circuits implementing the AND operation, respectively.

XAG-based algorithm to optimize T-depth

The database also contains quantum circuits optimized with respect to their T-depth. These database entries are synthesized using the hierarchical constructive algorithm presented in Section 6.3.2, described by the pseudo-code in Alg. 6.2.

The algorithm generates a circuit with a T-depth equal to the multiplicative depth $\tilde{d}$ of the XAG, which corresponds to the number of levels containing at least one AND node. Each AND node is implemented using the circuit with T-depth = 1 described in Section 4.3 (4.4) and reported here for completeness:



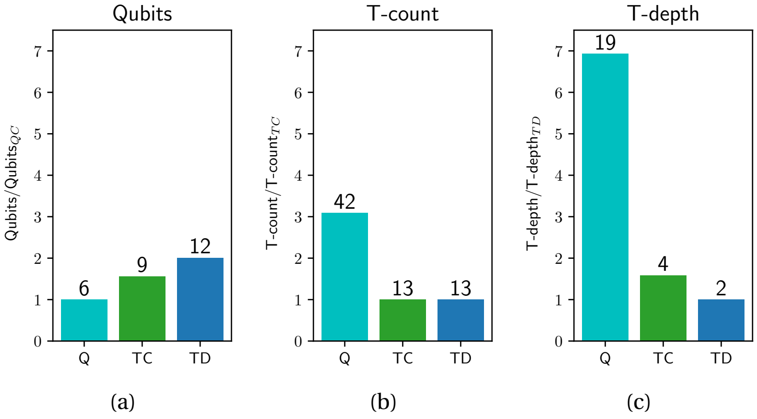


Figure 7.3 – Bar plots showing the cost functions characterizing the three different implementations for the database functions (Q, TC, TD). The y-axes report: (a) average ratio between number of qubits and best number of qubits in the database, (b) average ratio between T-count and best T-count in the database, (c) average ratio between T-depth and best T-depth in the database. Numbers on the bars show average values of the respective cost function for the three implementations.

where $|+\rangle = H|0\rangle$. If compared to the circuit (4.2) used by the previously described algorithm to compute the AND function, this circuit requires one more helper qubit.

The following procedure is performed by Alg. 6.2 for each level of the XAG:

1. finds the parity functions that are input to the AND nodes in the level;
2. computes the parity functions in-place using CNOT gates;
3. if the same parity function is input to more AND nodes, uses a CNOT gate to copy it on a helper qubit initialized to $|0\rangle$;
4. implements inversions using X gates;
5. implements all AND nodes in the level using (4.4).

By copying some of the qubits, this procedure enables parallel computation of all the AND nodes in one level. Please refer to Alg. 6.3.2 for a more detailed explanation of the algorithm.

7.1.3 Database of optimized quantum circuits

The obtained database is available online[1]. The optimized circuits can be downloaded in Q# format (Microsoft's quantum programming language). The database has a total of 162 circuit

[1] https://github.com/gmeuli/stg-benchmark

Table 7.1 – Specifications of the quantum circuits in the open-source database.

	T-count-optimized			T-depth-optimized			qubit-optimized		
truth-table	qubits	T-count	T-depth	qubits	T-count	T-depth	qubits	T-count	T-depth
8000	8	12	4	12	12	3	6	24	12
8080	7	8	3	9	8	2	5	16	8
0888	8	12	4	11	12	3	6	31	14
8888	6	4	2	8	4	1	5	7	3
7080	7	8	3	10	8	2	5	19	8
7880	9	12	4	12	12	3	6	36	13
7888	8	8	2	9	8	1	5	12	3
6ac06ac0	9	8	2	10	8	1	6	12	3
6ac8e000	11	16	5	16	16	2	6	63	28
80008000	9	12	4	13	12	3	6	24	12
80808080	8	8	3	10	8	2	6	16	8
88808000	10	12	4	12	12	3	6	26	11
88808080	10	16	4	13	16	3	7	48	24
88808880	9	12	4	12	12	3	6	31	14
88888888	7	4	2	9	4	1	6	7	3
a8808000	11	16	5	14	16	3	7	56	24
a8808080	9	12	4	12	12	3	6	40	20
a8808880	11	16	5	13	16	3	7	55	26
a880a880	8	8	3	11	8	2	6	15	6
a8888880	9	12	4	12	12	3	6	27	12
a888a080	9	12	4	12	12	3	6	55	24
a8e0c800	11	16	5	13	16	3	7	87	38
aa808080	10	16	4	13	16	3	7	63	30
b884a880	10	12	4	12	12	3	6	32	13
bc88a080	11	16	5	14	16	3	6	55	21
e0a8c880	11	16	5	14	16	2	6	27	13
e1808880	10	12	4	13	12	3	6	70	33
e8808000	9	12	4	13	12	3	6	42	18
e8808002	11	16	5	13	16	3	7	83	37
e8808080	11	16	4	14	16	3	7	64	32
e8808880	10	12	4	13	12	3	6	55	26
e880a880	11	16	5	13	16	3	7	49	20
e880e880	10	12	4	13	12	3	6	36	14
e8818880	11	16	4	13	16	3	7	65	31
e881e880	11	16	5	14	16	3	7	40	15
e8888880	11	16	5	14	16	3	7	63	28
e8a08880	11	16	5	14	16	3	7	87	40
e8c0a880	11	16	5	13	16	3	6	47	21
e9a0c088	11	16	5	14	16	3	7	63	26
e9c0a880	11	16	5	14	16	3	7	79	36
ea808080	10	12	3	13	12	2	6	42	18
eca08880	11	16	5	14	16	3	6	47	22
f8808880	11	16	4	14	16	3	7	79	36
f8888880	10	12	4	13	12	3	6	29	11
fca08880	11	16	5	14	16	3	7	70	29
2888a000	9	12	3	12	12	2	6	32	16
6ac8e240	11	16	5	13	16	3	6	44	17
78888888	10	12	4	14	12	2	6	19	8
80000000	10	16	4	12	16	3	7	32	16
80808000	10	16	4	12	16	3	7	56	28
88888880	11	16	4	12	16	3	7	39	16
e9808080	9	12	4	14	12	2	6	27	13
eac86240	10	12	4	12	12	3	6	19	8
ee84a060	11	16	5	16	16	2	6	43	20

implementations, three for each representative of the spectral equivalent classes of Boolean functions with 4 and 5 inputs.

Table 7.1 details the obtained number of qubits, T-count and T-depth for each different implementation and for each spectral class.

The characteristics of the original benchmark are compressed in Fig. 7.3. The three implementations are named: **Q**, the one targeting a small number of qubits; **TC** targeting a low-T-count; **TD**, targeting a low-T-depth. For each cost function c, a plot reports on the y-axis c/c_{BEST} averaged on the entire benchmark, where c_{BEST} is the best value of c among all three implementations. For example, the *T-count* plot reports for Q, TC, and TD the average T-count/T-count$_{TC}$ over all the 54 entries. In addition, each bar in the plots reports the average value for the respective cost function and implementation.

Any improvement of a single entry in the benchmark would be transferred to all the STGs controlled by functions that are spectral-equivalent to the entry's function. In fact, a quantum circuit for any Boolean function can be built from an instance in the benchmark. Note that the limited size of the benchmark entries does not reduce the impact of optimization results on bigger combinational designs, as these can be compiled from the circuits in the benchmark using decomposition-based techniques.

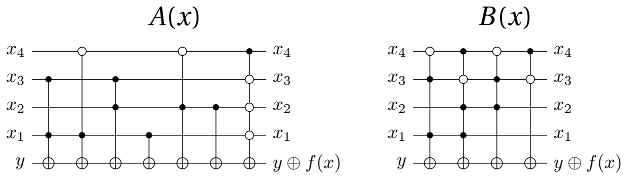
7.2 ESOP-based decomposition

In this section, I consider the problem of synthesizing a quantum circuit for Boolean functions with a number of input variables that can be bigger than 5. Section 4.2.1 already described how ESOP expressions are used to decompose a single-target gate into a cascade of multiple-controlled Toffoli gates. ESOP expressions are not canonical representations, meaning that there are several ESOPs representing the same Boolean function. For this reason, there are many exact and heuristic methods that aim to minimize such expressions, typically targeting the reduction of the number of product terms, or cubes. This section proposes a study of how advanced ESOP synthesis and optimization techniques can impact quantum compilation processes, which we publised in [186].

7.2.1 Optimal ESOP for quantum compilation

Traditionally, ESOP synthesis techniques aim to minimize the number of product terms, i.e., cubes, in the expression. A question arises: *Is the number of cubes the right cost function for the ESOP expressions used in quantum compilation?*

In the following, I introduce the problem of finding the right cost function to synthesize ESOP expressions used in quantum compilation, which leads to optimal quantum circuit characteristics, e.g., reduced number of T gates. The following example illustrates how the



ESOP	Gate types			
	#H	#NOT	#CNOT	#T
$A(x)$	18	6	46	52
$B(x)$	24	6	54	63

Figure 7.4 – Synthesis results for two different ESOPs representing the same function f.

classical cost function may not be suited for this application.

Example 7.2.1 *Given the Boolean function* $f(x) = \overline{x}_1\overline{x}_3x_4 \vee \overline{x}_2\overline{x}_3x_4 \vee \overline{x}_1x_2x_3\overline{x}_4 \vee x_1\overline{x}_2x_3\overline{x}_4$ *with* $x = x_1, \ldots, x_4$*, two possible ESOP expressions for* f *are:*

$$A(x) = \quad x_3x_1 \oplus \overline{x}_4x_1 \oplus x_3x_2 \oplus x_1 \oplus \overline{x}_4x_2 \oplus x_2 \oplus x_4\overline{x}_3\overline{x}_2\overline{x}_1$$
$$B(x) = \quad \overline{x}_4x_3x_1 \oplus x_4\overline{x}_3x_2x_1 \oplus \overline{x}_4x_3x_2 \oplus x_4\overline{x}_3$$

The first expression $A(x)$ consists of 7 product terms while the second expression, $B(x)$, is smaller and has size 4. It is possible to use these ESOPs to synthesize a reversible network for f and then compile each reversible gate into quantum gates using the algorithm described in [90]. The resulting networks and the composition of the quantum circuits are reported in Fig 7.4: #H is the number of Hadamard gates, #NOT and #CNOT are respectively the number of X and the number of controlled-CNOT gates, #T is the number of T gates. It is clearly shown how the second ESOP, independently from the smaller size, generates a quantum circuit with more gates. Differently, the first ESOP, that has larger size, shows characteristics allowing the compiler to create a circuit with reduced T gates, and fewer gates in general.

This analysis aims to identify the characteristics that lead to a better quantum circuit. With this in mind, it is possible to notice that the first ESOP has cubes with fewer literals, with respect to the second ESOP. Thus $A(x)$ generates a reversible circuit with multiple-controlled Toffoli gates with fewer controls and consequently a quantum circuit with fewer T gates. This suggests how the number of literals could be a better cost function for ESOPs used for quantum compilation. In general, evaluating the impact of different cost functions is necessary to use an ESOP synthesis method that accepts an arbitrary cost function. The next section describes how such a method can be implemented.

7.2.2 Constraint-based ESOP synthesis

The problem of finding an ESOP expression that realizes a Boolean function is known as *ESOP synthesis.*

The seminal work of Perkowski and Chrzanowska-Jeske [187] introduces the *Helliwell decision* *function* to characterize the solution space of ESOP synthesis for a given Boolean function. The Helliwell decision function $H_f(g_1,\ldots,g_K)$, $K \le 3^n$, for a given Boolean function $f(x_1,\ldots,x_n)$ describes synthesis as an odd-even covering problem in terms of the minterms of f. For each possible product term in n Boolean variables, a decision variable g_i, $1 \le i \le K$, is introduced. The Helliwell decision function is then defined by the logic equation

$$\bigwedge_{m \in f} \left(\left(\bigoplus_{g \in I(m)} g \right) \oplus f(m) \oplus 1 \right), \tag{7.1}$$

where $m \in f$ denotes that m is a minterm of f and I maps each minterm to the decision variables $g_{i_1},\ldots,g_{i_l}$ whose product terms are covered by m.

The logic equation (7.1) is constructed in such a way that every satisfying assignment $\hat{g}$ for $g = g_1,\ldots,g_K$ for $H(g)$ directly corresponds to an ESOP expression functionally equivalent to f.

Example 7.2.2 *Given the Boolean function $f(x_1,x_2) = x_1 \vee x_2$ with Boolean variables x_1 and x_2, the Helliwell decision function uses 9 Boolean variables $g_1,\ldots,g_9$, which are:*

$$\begin{aligned} &g_1 = \overline{x}_1\overline{x}_2 \quad g_2 = \overline{x}_1 x_2 \quad g_3 = x_1\overline{x}_2 \quad g_4 = x_1 x_2 \\ &g_5 = x_1 \quad g_6 = \overline{x}_1 \quad g_7 = x_2 \quad g_8 = \overline{x}_2 \\ &g_9 = 1 \end{aligned}$$

A SAT solver can find a selection of the cubes such that minterms for which f evaluates to one are covered an odd number of times, whether minterms for which f evaluates to false are covered an even number of times. Constraints must be added to the problem in order for the SAT solver to find a valid solution. The overall Helliwell decision function for f is:

$$\begin{aligned} H(g) = &(g_1 \oplus g_6 \oplus g_8 \oplus g_9 \oplus 0 \oplus 1) \;\wedge\; (g_2 \oplus g_7 \oplus g_6 \oplus g_9 \oplus 1 \oplus 1) \;\wedge \\ &(g_3 \oplus g_5 \oplus g_8 \oplus g_9 \oplus 1 \oplus 1) \;\wedge\; (g_4 \oplus g_5 \oplus g_7 \oplus g_9 \oplus 1 \oplus 1) \end{aligned}$$

Fig. 7.5 shows three possible ESOP covers on the Karnaugh map: g_4, g_6, g_8 and g_4, g_5, g_7 and g_1, g_9.

Size-minimal ESOP synthesis

Size-minimal ESOP synthesis is the problem of finding an ESOP expression for a given Boolean function f with a minimum number of product terms. Utilizing equation (7.1), the problem can be solved by computing minimum satisfying assignments for $H_f(g)$. An assignment $\hat{g}$ is minimum satisfying if the following conditions hold:

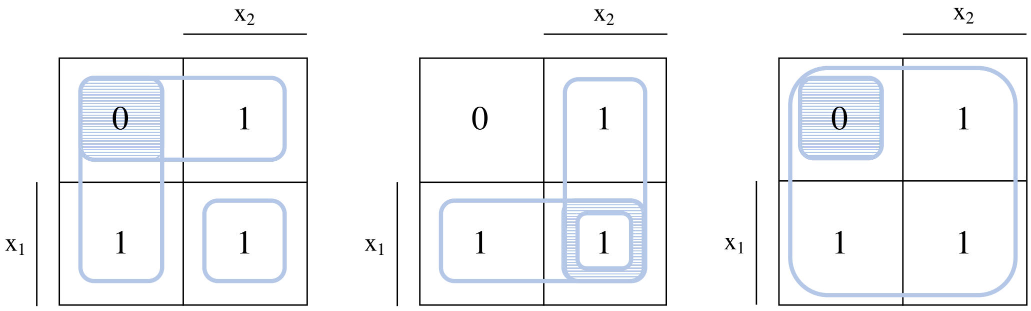


Figure 7.5 – Three possible ESOP covering for the function $f = x_1 \vee x_2$.

1. $\hat{g}$ satisfies H_f

$$H_f(\hat{g})$$

2. any other assignment g different from $\hat{g}$ that satisfies H_f does not imply $\hat{g}$

$$\forall g : \bigl(g \not\leftrightarrow \hat{g} \wedge H_f(g)\bigr) \implies g \not\rightarrow \hat{g}$$

Please note that the symbols $\not\leftrightarrow$ and $\not\rightarrow$ are used to indicate implication relations between assignments.

In the following, the idea of utilizing the Helliwell decision function for synthesizing size-minimum ESOP expression is generalized to synthesizing cost-minimal ESOP expressions, where the cost function is provided as a part of the input.

Cost-minimal ESOP synthesis

Given a Boolean function f over n Boolean variables and a cost function $\kappa : \{0,1,-\}^n \to \mathbb{N}_{>0}$, that maps product terms to positive integer values (costs), cost-minimal ESOP synthesis is the problem of finding an ESOP expression $t_1 \oplus \cdots \oplus t_k$ that realizes f such that $\bigwedge_{i=1}^{k} \kappa(t_i)$ is minimal.

Two different cost functions κ_0 and κ_1 are presented to illustrate the idea of cost-minimal ESOP synthesis. In general, the cost function should be picked keeping the usage of the ESOP expression in mind.

The constant function

$$\kappa_0(t) = 1 \tag{7.2}$$

defines unit costs for all product terms. If used, each ESOP expression obtained as solution of cost-minimal ESOP synthesis has a minimum number of product terms. The cost function

$$\kappa_1(t) = |t| + 1, \tag{7.3}$$

where $|t|$ counts the number of literals in t, weights each product term by the number of appearing literals. The additional 1 ensures that all costs—including the costs of the empty product term—are greater than 0.

Example 7.2.3 *Consider the Boolean function*

$$f_1(x) = \bar{x}_1\bar{x}_2x_3x_4 \vee \bar{x}_1x_2\bar{x}_3x_4 \vee \bar{x}_1x_2x_3\bar{x}_4 \vee x_1\bar{x}_2\bar{x}_3x_4 \vee x_1\bar{x}_2x_3\bar{x}_4 \vee x_1x_2\bar{x}_3\bar{x}_4$$

with $x = x_1,\ldots,x_4$. A cost-minimal ESOP expression that realizes f_1 with respect to the cost function κ_0 is

$$\bar{x}_1x_2\bar{x}_4 \oplus x_2\bar{x}_3 \oplus \bar{x}_2x_3\bar{x}_4 \oplus \bar{x}_1\bar{x}_2x_3 \oplus x_1\bar{x}_3x_4,$$

whereas a cost-minimal ESOP expression for the same Boolean function with respect to the cost function κ_1 is

$$x_1 \oplus x_2 \oplus \bar{x}_3 \oplus x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \oplus x_1x_2x_3x_4.$$

Computing cost-minimal ESOPs

The proposed SAT-based procedure computes cost-minimal ESOP expressions using (weighted) maximum satisfiability (MAX-SAT) [188].

MAX-SAT deals with solving over-constrained constraint satisfaction problems modulo Boolean logic. The problems consist of hard and soft clauses, where each soft clause is associated with an integer weight greater than 0. The constraint satisfaction problem initially is unsatisfiable and the task of a MAX-SAT oracle is to find a minimal-cost relaxation of the soft clauses, i.e., the oracle has to remove a subset of the soft clauses, such that the problem becomes satisfiable while a given cost function is minimized.

Given a Boolean function f over n Boolean variables and a cost function $\kappa : \{0,1,-\}^n \to \mathbb{N}_{>0}$, cost-minimal ESOP synthesis is solved in three steps:

1. Formulate the Helliwell decision function $H(g)$ as described in (7.1).
2. Invoke a MAX-SAT oracle to find a satisfying assignment $\hat{g} = \hat{g}_1,\ldots,\hat{g}_K$ that minimizes $\sum_{i=1}^{K} \kappa(g_i)$ subject to $\mathrm{CNF}[H(g)] \wedge (\bigwedge_{i=1}^{K} \bar{g}_i)$, where CNF translates the XOR-clauses to *conjunctive normal form* (CNF).
3. Construct the ESOP from the satisfying assignment $\hat{g}$.

The described approach is independent from the choice of the MAX-SAT oracle and the translation to CNF, but uses them as black-boxes.

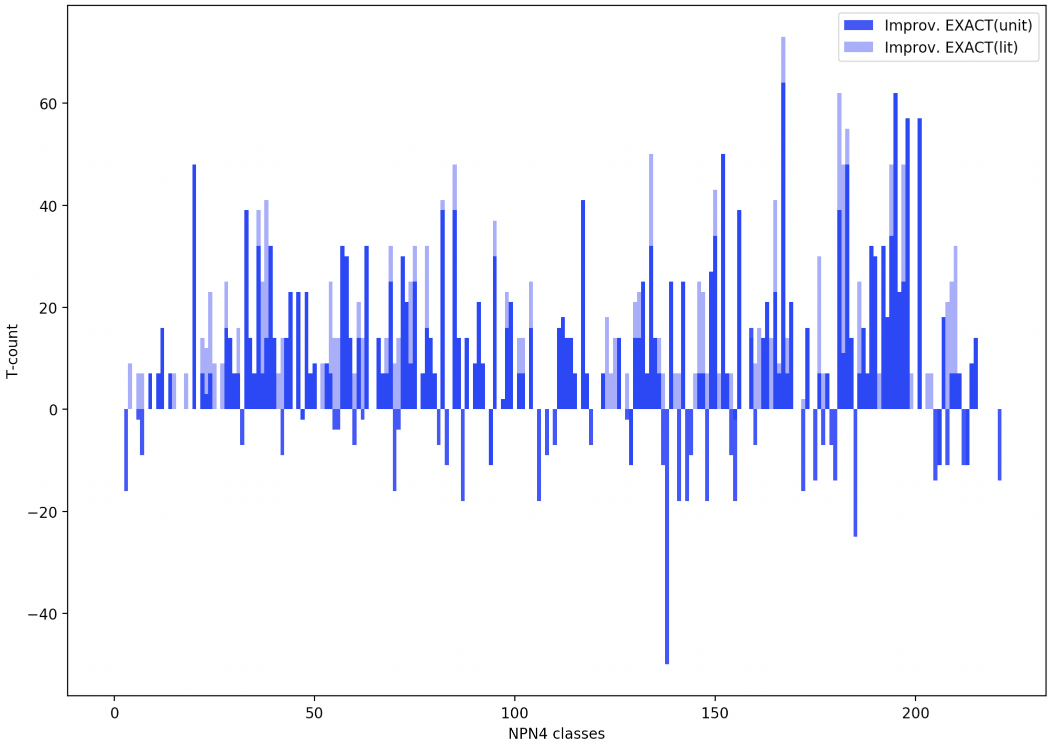


Figure 7.6 – Histogram showing the improvement of exact methods over *PKRM* with respect to two different cost functions: number of terms (*EXACT(unit)*) and number of literals (*EXACT(lit)*).

7.2.3 Results

The proposed SAT-based exact synthesis method is implemented in the open-source C++ library *easy*[2] [181, 189] using its own C++ implementation of RC2 [190] as MAX-SAT oracle. The *easy* library provides implementations of various verification and synthesis algorithms for ESOP expressions.

NPN4 equivalence classes

In this section, the effect of different ESOP optimization methods is evaluated on the 222 representatives of the NPN4 equivalence classes. The results report the number of product terms in the ESOP, as well as, the number of T gates in the generated quantum circuits, comparing different state-of-the-art ESOP synthesis methods and the proposed constraint-based approach:

1. *Positive Polarity Reed Muller* (*PPRM*) [191],

[2] https://github.com/hriener/easy

Cost function	ESOP Synthesis Method				
	PPRM	PKRM	EXORCISM	EXACT(unit)	EXACT(lit)
avg. ESOP size	7.77	4.69	3.41	3.41	3.42
avg. num. T gates	87.35	82.32	59.05	67.50	58.19

Table 7.2 – Comparison of different ESOP synthesis methods.

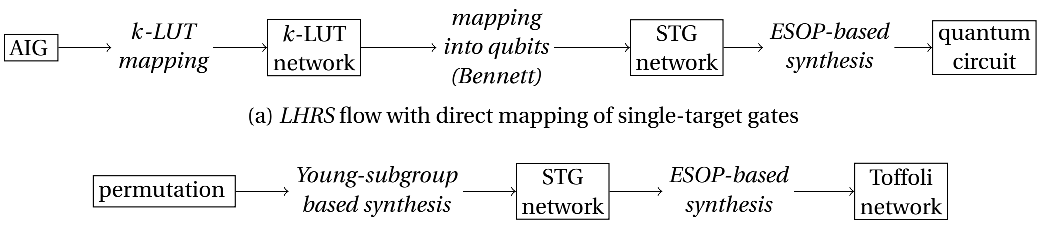
2. *Pseudo-Kronecker Reed Muller* (*PKRM*) [192],
3. *EXORCISM* [116] and
4. *EXACT(unit)* and *EXACT(lit)* minimizing respectively κ_0 and κ_1

Table 7.2 reports the average number of product terms (size) and the average number of T gates for each of the ESOP synthesis methods. *PPRM* and *PKRM* are special cases of general ESOP expressions, that can be easily derived from a given Boolean function but are sub-optimal when considering the number of product terms. They are often used as starting covers for ESOP optimization approaches. They are reported to enable better comparability of the achieved reduction. *EXORCISM* is a fast cube transformation heuristic, capable of finding close to optimal ESOP expressions, starting from a PKRM cover of the Boolean function. Nevertheless, *EXORCISM* is an heuristic method and does not guarantee the minimality of the solution. In many cases, reducing the size of an ESOP also leads to a reduction of the number of T gates. Consequently, *EXORCISM*, *EXACT(unit)*, and *EXACT(lit)* improve over *PPRM* and *PKRM*. Reducing the number of literals also has a positive effect on the T gates, i.e., *EXACT(lit)* achieves a better reduction than *EXACT(unit)*. Moreover, *EXORCISM* also improves over the *EXACT(unit)* method because its heuristic prefers don't cares over concrete values and reduces the overall number of literals in an ESOP expression. The histogram in Fig. 7.6 gives a more detailed overview of the improvement in T-count of *EXACT(lit)* and *EXACT(unit)* over *PKRM*, respectively, for all the 222 representatives in NPN4 equivalent classes.

Optimizing size and literals, however, does not minimize the number of T gates, as illustrated by the following example: consider the two equivalent ESOPs

$$C(x_1, x_2, x_3) = 1 \oplus \bar{x}_1 x_2 \oplus x_1 x_2 x_3 \text{ and } D(x_1, x_2, x_3) = x_1 x_2 \bar{x}_3 \oplus \bar{x}_2 \oplus \bar{x}_1. \tag{7.4}$$

Both ESOPs have the same number of product terms and the same number of literals. To realize $C(x_1, x_2, x_3)$ as quantum circuit, however, 23 T gates are required, whereas for realizing $D(x_1, x_2, x_3)$ 16 T gates are needed. This results suggest that in future work it would be valuable to identify more fitting cost functions than the number of literals. In addition, future technology developments could themselves require different cost functions. The proposed constraint-based method could provide the flexibility to enable future research in this direction.



(a) *LHRS* flow with direct mapping of single-target gates

(b) *DBS* flow

Figure 7.7 – Two different state-of-the-art compilation flows for Boolean functions that use ESOP-based reversible synthesis.

Algorithm 7.1 Pseudo-exact optimal ESOP

Input: control function $f : \mathbb{B}^n \to \mathbb{B}$
Output: optimized ESOP expression of f

```
function GET_ESOP
    if $f \in cache$ then return $cache[f]$
    if $f$ is symmetric then
        $esop \leftarrow PKRM(f)$
    else if $n \leq 4$ then
        $esop \leftarrow EXACT_{LIT}(f)$
    else
        $esop \leftarrow EXORCISM(f)$
    $cache.insert(f, esop)$
    return $esop$
```

Integration into quantum compilation flows

This section reports the result of integrating the advanced ESOP optimization methods into quantum compilation flows. In addition to the hierarchical reversible synthesis flow, presented in Chapter 5, the Decomposition-Based Synthesis (DBS) flow, which compiles permutations, is considered. The two flows are compared in Fig. 7.7.

The pseudo-optimal portfolio approach described in Alg. 7.1 is used to integrate different ESOP synthesis methods. For each symmetric control function, the ESOP expression *esop* is computed using the *PKRM* method, that is optimum in this case. If the number of inputs is smaller than or equal to 4, then one of the exact methods is used. For larger functions, the approach uses the heuristic *EXORCISM* (command *&exorcism -q* of abc [119]).

The first experiment evaluates the improvement of the proposed method integrated into DBS(Fig. 7.7(b)). Table 7.3 shows the synthesis results for reversible permutations from

Table 7.3 – Comparison between exact method and heuristic for small reversible functions

Permutation		PKRM		EXACT(lit)		EXACT(unit)	
	Q	T	t[s]	T	t[s]	T	t[s]
hwb4	4	123	0.0	109	0.1	116	0.0
hwb5	5	514	0.0	337	59.9	447	0.3
hwb6	6	1361	0.0	993	0.9	993	0.9
hwb7	7	5331	0.0	3066	1.0	3066	1.1
hwb8	8	13562	0.0	7654	1.2	7654	1.2
mod5_11	5	453	0.0	350	36.5	368	0.2
mod5_12	5	453	0.0	361	59.1	400	0.2
mod5_13	5	428	0.0	329	38.2	343	0.1
mod5_17	5	478	0.0	382	64.3	414	0.3
mod5_21	5	433	0.0	352	34.9	482	0.1
mod5_22	5	469	0.0	354	25.0	391	0.1
mod5_24	5	503	0.0	405	61.4	448	0.3
mod5_3	5	494	0.0	386	34.8	411	0.2
mod7_14	7	5201	0.0	2936	1.0	2936	1.0
mod7_3	7	4945	0.0	2957	1.0	2957	1.0
mod7_7	7	4859	0.0	3039	1.0	3039	1.0
prime4	4	102	0.0	95	0.0	106	0.0
prime5	5	367	0.0	271	28.5	289	0.1
prime6	6	1054	0.0	786	0.8	786	0.7
prime7	7	3600	0.0	2283	1.0	2283	0.9
prime8	8	8302	0.0	4420	1.1	4420	1.0
avg. reduction *EXACT(lit) = 28.23%*							
avg. reduction *EXACT(unit) = 22.66%*							

Maslov's reversible benchmark[3] and for the reversible functions $MOD_{n/g} : \mathbb{B}^n \rightarrow \mathbb{B}^n$, where:

$$MOD_{n/g} = \begin{cases} 0 & \text{if} \quad x = 0 \\ g^x mod(2^n - 1) & \text{if} \quad 1 \leq x \leq 2^n - 2 \\ 2^n - 1 & \quad \text{otherwise} \end{cases}$$

The data show a reduction in the number of T gates, with respect to the *PKRM* method, for both the *EXACT* approaches. In addition, if the synthesis is performed to minimize the number of literals in each cube, the T-count can be further improved. Indeed, the *unit* approach gets to 22.66% improvement, while *lit* gives 28.23% improvement.

The second experiment aims to evaluate the integration into the hierarchical synthesis framework. Table 7.4 shows the results obtained by synthesizing quantum circuits for the arithmetic designs of the EPFL benchmark[4]. The first steps of the flow generate a reversible circuit made of single-target gates, each one with a control function of maximum k inputs, where k is the LUT size used to build the k-LUT network. An ESOP expression is synthesized for each control function and translated into quantum circuits as described in [90]. The flow integrating the pseudo-exact approach is compared against the flow using *PKRM* for the mapping of single-target gates. Synthesis results are provided for LUT sizes (k) from 4 to 10. The maximum reduction of number of T gates is obtained in the case of k = 10 equal to 36.32% and the

[3] http://webhome.cs.uvic.ca/~dmaslov

[4] https://github.com/lsils/benchmarks

Table 7.4 – Comparison between PKRM and the pseudo-exact optimal ESOP synthesis integrated into LHRS to synthesize the EPFL arithmetic benchmark

			PKRM		**Opt.**				**PKRM**		**Opt.**	
k		Q	T	t[s]	T	t[s]		Q	T	t[s]	T	t[s]
4	adder	511	5398	0.0	5356	0.4	bar	1415	76816	0.2	56320	1.8
5		448	16061	0.1	15151	0.5		1031	95576	0.3	63694	2.9
6		448	16271	0.1	15279	0.6		647	52750	0.2	50944	1.8
7		427	37259	0.1	36110	0.7		647	52750	0.3	50944	1.9
8		427	37963	0.1	36654	0.7		647	52750	0.3	50944	1.9
9		416	84076	0.2	72338	0.8		647	52750	0.3	50944	1.9
10		416	85509	0.2	72985	0.9		647	52750	0.3	50944	1.9
4	div	26467	757193	5.8	635999	12.4	hyp	64630	2448872	25.3	2208000	37.5
5		24474	851035	6.8	690622	15.1		56568	2647894	26.1	2156087	40.5
6		24083	876636	8.0	709586	19.0		50118	2860466	28.2	2145634	46.6
7		23944	939887	9.6	742327	23.8		48399	3501767	31.0	2817812	51.8
8		23808	1034583	11.2	773058	26.6		47581	4540244	36.9	3546120	66.7
9		23711	1204407	13.0	831482	30.5		46992	5379295	43.0	4158260	79.1
10		23633	1710038	15.4	875766	34.3		46933	6238649	50.0	4596940	94.4
4	log	10420	458335	2.4	380787	12.8	max	1484	54422	0.2	42684	5.4
5		9661	623957	3.2	492501	24.1		1346	76507	0.2	60597	6.4
6		8156	1033225	4.3	768429	49.4		1256	104109	0.3	79853	6.4
7		8141	1507690	5.1	883462	103.7		1149	148355	0.4	102310	6.0
8		4658	2196359	6.2	1228593	48.1		1067	209851	0.6	140106	6.9
9		4456	3393095	8.4	1912337	65.8		977	323027	0.8	200270	5.9
10		3697	5786642	10.8	3268408	74.8		929	355341	1.1	230118	5.6
4	mult	8194	359422	1.8	270268	6.2	sin	1962	71409	0.4	64103	14.5
5		8100	479930	2.2	368062	8.8		1818	82386	0.5	71471	19.4
6		6706	1034190	2.8	579420	11.6		1608	115107	0.7	92659	25.7
7		7050	1448336	3.7	847558	15.2		1553	137989	0.9	104092	27.3
8		5101	1371054	3.7	818914	16.6		1449	249964	1.2	157332	32.5
9		5165	2115333	5.2	1410009	18.3		915	794521	1.5	362082	33.2
10		4006	3657831	8.0	2417393	23.9		878	1241237	2.2	542136	37.2
4	sqrt	8686	317522	1.7	255275	6.4	square	6909	354552	1.5	240636	11.5
5		8351	344049	2.3	265948	6.8		6092	553311	1.9	308262	18.0
6		8332	391900	2.9	285310	7.8		4195	299574	1.8	206683	16.1
7		8152	448518	3.4	301246	8.4		4213	368160	2.0	261272	22.8
8		7986	709282	4.4	358215	9.5		3764	477446	2.3	341092	24.3
9		7976	720144	5.3	359296	10.6		3724	658343	2.9	445505	32.5
10		7966	1413589	7.0	540586	12.4		3792	876884	3.7	532124	41.4
min avg. improvement k=4 : 17.86 %												
max avg. improvement k=10 : 36.32 %												
avg. improvement: 26.36 %												

minimum reduction in the case of $k = 4$ equal to 17.86%.

7.3 LUT-based decomposition

This section describes how to exploit LUT mapping to translate a single-target gate into a network of multiple-controlled Toffoli gates. An LUT-based mapping method performs k-LUT mapping, dividing the network into subnetworks, each one having maximum k inputs. If the mapped LUT network is composed of ℓ LUTs, then it is possible to map the single-target gate into a network of single-target gates with at most k inputs, by using $\ell - 1$ helper qubits.

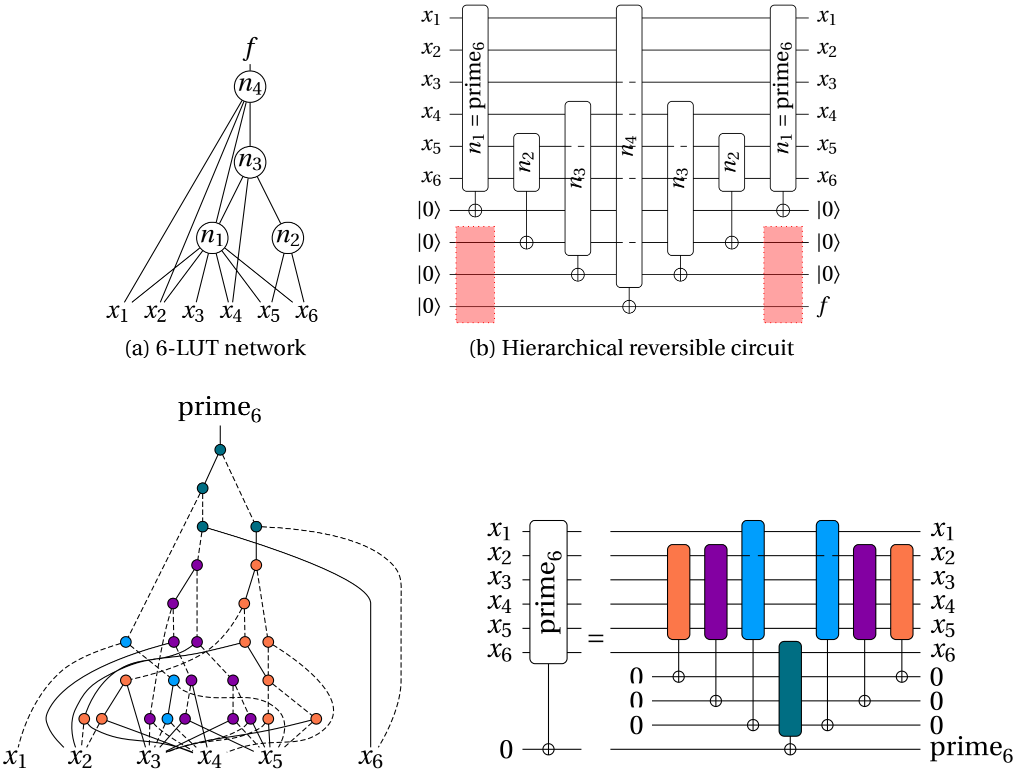
Example 7.3.1 *Consider the 6-LUT network in Fig. 7.8 (a). This network is characterized by 6 primary inputs $x_1, \dots, x_6$ and four nodes, implementing the Boolean functions n_1, n_2, n_3 and n_4. Such network can be translated using the hierarchical flow and the Bennett uncomputation strategy into the reversible network in Fig. 7.8 (b). Please note how the first single-target gate, which computes the function $n_1 = \mathrm{prime}_6$, has three helper qubits available for its decomposition, as highlighted by the boxes in red. Fig. 7.8 (c) shows a 4-LUT mapping of the multi-level logic network, i.e., an AIG, for the control function* prime_6. *Finally, Fig. 7.8 (d) shows how the single target gate computing the* prime_6 *function can be mapped into all the available qubits according to the 4-LUT mapping in Fig. 7.8 (c).*

This method requires extra qubits to store intermediate results, differently from the ESOP-based decomposition strategy. Nevertheless, consider the case in which a hierarchical synthesis method is applied on a large combinational design, generating a network of STGs, as in Fig. 7.8 (b). Each STG acts on a limited number of qubits, but has at its disposal many other qubits, i.e., other extra qubits used as target for the other STGs. The LUT-based decomposition methods are designed to take full advantage of all the qubits available in the network. Once the control function of the STG has been decomposed into smaller functions, it is possible to use precomputed optimal implementations, as shown in Section 7.1. Otherwise, if a function obtained by the decomposition exceeds the number of controls of the available database, the ESOP method is used.

It is possible to distinguish different strategies to perform such LUT decomposition of single-target gates.

Hybrid LUT-based decomposition

The decomposition method takes as input the control function of the STG represented as a logic network. It performs a k-LUT mapping, setting k in order to generate sub-networks with a number of input matching the size of the optimal functions in the database. However, it can happen that there are not enough helper qubits available to store all the intermediate values of the mapping. In this case, the hybrid method merges two LUTs into a larger one, thereby requiring one fewer helper qubit. This procedure is repeated until the number of available helper qubits suffices. This procedure may lead to a very large LUT, to the point that this LUT can have more inputs than the number of primary inputs.


f
n_4
n_3
n_1
n_2
x_1 x_2 x_3 x_4 x_5 x_6
x_1 x_2 x_3 x_4 x_5 x_6 $|0\rangle$ $|0\rangle$ $|0\rangle$ $|0\rangle$
$n_1 = \text{prime}_6$
n_2
n_3
n_4
x_1 x_2 x_3 x_4 x_5 x_6 $|0\rangle$ $|0\rangle$ $|0\rangle$ f
prime$_6$
x_1 x_2 x_3 x_4 x_5 x_6
prime$_6$
0
=
0 0 0
x_1 x_2 x_3 x_4 x_5 x_6 0 0 0 prime$_6$

(a) 6-LUT network

(b) Hierarchical reversible circuit

(c) 4-LUT mapping of the control function of n_1 = prime$_6$

(d) LUT-based mapping of the STG controlled by n_1

Figure 7.8 – Example of applying a LUT-based mapping method to single-target gates obtained from the hierarchical reversible synthesis flow.

Best-fit decomposition

The best-fit mapping [193] is a LUT-based decomposition method that fully exploits the available helper qubits. It also leverages near-optimal precomputed networks, and does not generate large networks to be synthesized using the ESOP method. The idea is to find a suitable value for k. This value is chosen to be the smallest for which the LUT size fits the available number of helper qubits. Starting from $k = 4$, k is incremented until the above mentioned condition is satisfied. Once the mapping is obtained, each k-LUT needs to be mapped in Clifford+T gates. If the number of a LUT's inputs is small enough, the LUT can be replaced by a precomputed Clifford+T network, otherwise ESOP-based decomposition is applied. Using the best-fit mapping method, each STG in a reversible network is decomposed using a different k parameter for the LUT decomposition, which indeed *fits* the number of helper qubits available for the specific single-target gate.

7.4 Experimental results

This section presents an experiment carried out using the open-source tool RevKit [194]. The experiment consists of synthesizing quantum circuits computing the logic functions of the EPFL combinational benchmark suite[5]. The state-of-the-art LUT-based hierarchical reversible flow (*LHRS*) is applied, setting the k parameter to 6, 10 and 16.

Originally, RevKit exploited a database of precomputed optimized circuits compressed using affine classification, as described in [195], which enabled storing entries up to 4 variables. For this experimental evaluation, the database is changed to exploit spectral classification, with stored functions up to 5 inputs. This database, integrated into RevKit, is obtained using a procedure similar to the ESOP-based one presented in Section 7.1.2. The difference is that, to further reduce the T-count, more than one helper qubit is allowed. The reversible circuits representing all the entries of this database are reported in Appendix A.

The experiment include four different configurations: *hybrid, spectral* where the hybrid LUT mapping is coupled with a database of spectral equivalent classes; *hybrid, affine* where the hybrid LUT mapping is coupled with a database of affine equivalent classes; *best-fit, spectral* where the best-fit LUT mapping is coupled with a database of spectral equivalent classes; and *best-fit, affine* where the best-fit LUT mapping is coupled with a database of affine equivalent classes.

Table 7.5 shows the experimental results. The last rows summarizes the improvement obtained using the database based on spectral equivalence with respect to the one based on affine equivalence, by computing the normalized geomeans, as well as the average and maximum improvement in T-count, independently for the hybrid and best-fit mapping methods. The improvement is stronger in the hybrid mapping case. In addition, the experiment suggests that the best-fit method, combined with the spectral-based database, enables a significant reduction of the T-count. Finally, it can easily be seen that the runtime improved after using the spectral classification technique. This is a consequence of having fewer single-target gates synthesized using the ESOP decomposition method, and of the reduced size of the database.

7.5 Summary

In this chapter, I analyzed the problem of decomposing single-target gates into quantum circuits. In particular, I focused on the specific application of this problem in the hierarchical reversible synthesis of oracle circuits. In Section 7.1, I considered the case in which the STG to be decomposed is controlled by a Boolean function with a number of inputs smaller than or equal to 5. In this case, the size of the control function enables us to generate a database of precomputed quantum circuits optimized for specific cost functions. As I researched compilation methods for fault-tolerant quantum computing, the database contains entries

[5] https://github.com/lsils/benchmarks

minimized for the T-count, the T-depth, and the number of qubits. In Section 7.1.2, I gave details on the algorithms used to generate the database, which is publicly available at https://github.com/gmeuli/stg-benchmark. Future work should target the integration of this database into the hierarchical reversible synthesis method (*ROS*) described in Chapter 5.

In the second part of the chapter, I moved to the case in which the control function of the single-target gate is arbitrarily large. One option is to use ESOP-base decomposition. Section 7.2 presented a study on the use of advanced SAT-based ESOP synthesis methods in quantum compilation. The study concludes that ESOP synthesis methods for this application should take into account different cost functions than the standard number of terms. For this reason, I proposed an exact synthesis method capable of minimizing arbitrary cost functions. Future work should focus on the development of heuristics to synthesize large expressions according to more fitting cost functions.

A second option for large control functions is to decompose them using k-LUT-decomposition methods, presented in Section 7.3. This method is particularly useful when several helper qubits are available. Section 7.4 reported some experimental results on the use of a spectral-equivalence-based database combined with k-LUT-decomposition methods in the compilation of large combinational logic designs.

Table 7.5 – Experimental evaluation of different single-target gate decomposition methods in LHRS on the EPFL arithmetic benchmarks.

	LUT size		*hybrid, spectral*		*hybrid, affine*		*best fit, spectral*		*best fit, affine*	
			T-count	runtime [s]	T-count	runtime [s]	T-count	runtime [s]	T-count	runtime [s]
adder	6	Best-LUT	2,734	0.00	21,023	0.43	2,515	0.00	12,623	0.65
		Original	1,893	0.01	1,988	0.05	1,899	0.01	2,066	0.20
	10	Best-LUT	4,632	0.02	21,953	0.46	3,609	0.01	13,679	0.81
		Original	2,688	0.01	2,730	0.05	2,698	0.01	2,860	0.21
	16	Best-LUT	6,584	0.02	23,273	0.64	4,582	0.01	14,802	1.16
		Original	4,865	0.02	4,914	0.10	4,881	0.01	5,122	0.24
bar	6	Best-LUT	17,024	0.04	17,024	0.05	17,024	0.05	17,024	0.05
		Original	114,374	0.23	108,917	0.95	52,327	0.25	76,883	1.75
	10	Best-LUT	24,908	0.07	25,676	0.08	31,272	0.18	42,656	1.23
		Original	114,374	0.26	108,917	0.96	52,327	0.11	76,883	1.72
	16	Best-LUT	46,838	0.11	47,390	0.25	58,898	0.08	79,530	1.12
		Original	116,202	0.13	110,669	0.82	53,543	0.12	78,671	1.96
div	6	Best-LUT	380,467	0.77	405,896	3.31	296,404	0.82	435,554	13.99
		Original	663,464	4.18	729,940	11.40	622,086	4.52	819,918	9.45
	10	Best-LUT	457,979	0.86	466,635	2.67	364,483	0.95	510,188	15.08
		Original	829,094	4.91	874,476	6.72	737,552	4.51	1,073,427	12.58
	16	Best-LUT	880,482	1.32	882,893	2.80	669,825	1.51	956,102	22.42
		Original	1,005,870	4.72	1,044,271	6.94	896,943	4.56	1,296,742	13.10
hyp	6	Best-LUT	6,052,714	71.92	6,495,823	371.59	2,783,871	72.25	5,050,444	75.71
		Original	2,984,620	38.05	3,021,920	174.50	2,411,474	36.65	3,571,054	36.04
	10	Best-LUT	8,618,483	73.21	8,789,664	250.10	3,832,579	73.53	7,880,861	81.21
		Original	4,405,839	33.90	4,330,883	217.42	3,606,044	53.34	5,430,905	40.73
	16	Best-LUT	11,734,937	49.77	11,939,196	366.14	5,148,762	47.82	11,580,940	87.36
		Original	6,299,001	64.85	6,194,774	325.07	5,322,818	41.06	8,146,674	47.58
log2	6	Best-LUT	714,982	2.05	860,028	20.35	587,663	1.90	1,363,890	50.64
		Original	487,876	1.62	628,777	5.04	471,786	1.69	996,739	9.66
	10	Best-LUT	13,389,050	17.87	14,116,708	74.62	13,516,750	22.95	20,415,921	75.39
		Original	15,025,683	33.26	15,188,782	91.37	16,591,838	32.20	25,763,551	43.03
	16	Best-LUT	28,417,955	200.37	33,794,883	648.48	33,700,030	118.90	46,996,388	197.99
		Original	40,245,230	93.52	41,207,475	587.85	43,916,658	111.81	66,794,792	123.33
max	6	Best-LUT	14,440	0.02	18,369	1.30	14,106	0.02	19,849	1.02
		Original	41,333	0.06	48,490	1.60	41,008	0.06	64,986	3.61
	10	Best-LUT	22,932	0.07	23,460	0.36	22,227	0.09	25,288	1.09
		Original	55,967	0.10	55,993	0.63	61,967	0.20	88,321	3.28
	16	Best-LUT	29,760	0.11	32,380	0.49	29,658	0.21	33,064	1.28
		Original	63,726	0.12	63,993	1.31	68,362	0.18	93,762	4.73
mult	6	Best-LUT	852,784	1.77	929,110	5.81	412,161	1.65	868,907	17.08
		Original	330,101	0.97	399,632	7.87	287,570	1.23	733,580	4.87
	10	Best-LUT	1,247,591	2.08	1,288,912	10.40	757,357	2.36	1,368,712	19.64
		Original	1,065,056	2.90	1,035,633	10.48	1,133,820	3.85	2,518,857	8.79
	16	Best-LUT	2,585,906	8.21	2,581,486	78.11	1,624,371	5.20	3,180,890	22.32
		Original	1,316,854	9.28	1,299,740	11.11	1,375,811	4.76	3,123,922	10.46
sin	6	Best-LUT	166,548	0.25	195,203	7.84	131,431	0.25	211,222	31.35
		Original	72,337	0.10	109,248	5.25	70,613	0.13	134,658	8.61
	10	Best-LUT	600,374	1.67	603,866	26.14	682,426	2.65	875,196	36.85
		Original	655,022	1.79	661,028	6.51	867,436	3.64	1,205,074	15.91
	16	Best-LUT	1,416,441	23.22	1,581,160	93.50	1,217,234	6.66	1,523,135	45.03
		Original	1,888,349	4.22	1,938,452	11.96	2,281,121	7.75	3,234,887	22.96
sqrt	6	Best-LUT	387,266	0.75	407,081	6.54	313,607	0.82	447,876	15.72
		Original	266,611	0.78	712,757	2.00	261,813	0.73	749,687	3.95
	10	Best-LUT	496,633	0.88	508,105	3.67	398,264	1.01	564,062	21.15
		Original	322,617	1.05	758,762	1.58	316,977	0.96	833,748	5.84
	16	Best-LUT	1,148,079	7.97	1,251,139	28.74	784,685	2.01	1,124,877	22.07
		Original	432,535	1.25	861,567	2.04	416,513	1.22	1,028,377	6.44
square	6	Best-LUT	617,451	1.16	651,639	5.53	241,146	0.82	490,182	18.60
		Original	492,341	0.96	504,369	3.58	481,628	1.02	768,541	6.24
	10	Best-LUT	903,446	1.14	911,949	8.14	371,268	1.07	828,392	20.33
		Original	804,561	1.46	781,512	4.39	788,937	1.62	1,242,062	10.75
	16	Best-LUT	1,938,405	117.20	1,950,409	232.57	694,144	3.67	1,501,536	25.56
		Original	1,933,222	135.06	1,898,077	219.48	1,384,694	6.72	2,101,950	19.70
Normalized geomean			0.84		1.00		0.60		1.00	
Average improvement			10.94%				37.54%			
Maximum improvement			87.00%	Adder, $k = 6$, Best-LUT			80.08%	Adder, $k = 6$, Best-LUT		
Normalized geomean			1				0.82			

8 Quantum circuit optimization

This chapter focuses on optimization techniques to reduce the resources required to compute quantum circuits in a fault-tolerant setting. Two optimization techniques are presented: one to be applied on reversible circuits that aims to reduce the number of T gates, and one to be applied on Clifford+T circuits that aims to minimize the number of CNOT gates.

8.1 T-count optimization using graph matching

Some optimization techniques aiming to reduce the implementation cost of quantum circuits generated from reversible networks can be found in the literature [196, 197].

This section presents a technique to reduce the T-count of quantum circuits obtained from reversible networks consisting of multiple-controlled Toffoli gates. The optimization is effective for reversible circuits obtained with the ESOP-decomposition method, since these circuits consist of a set of multiple-controlled Toffoli gates all acting on the same target line. Similar networks can be optimized by using some properties that apply to pairs of gates. Such properties were originally proposed in [3] and are described in the next section. The presented optimization method solves a graph matching problem to select the best strategy to apply such optimization properties in order to maximize the gain, i.e., T-count reduction in the corresponding quantum circuit.

8.1.1 Optimization properties

The following rules describe ways to combine two multiple-controlled Toffoli gates according to their control lines. All the rules apply on two gates which share the same target line.

Rule 8.1.1 *Let* $g_1 = g_2 = \mathrm{T}(C, t)$. *Then* $\mathrm{costs}(g_1 \circ g_2) = 0$.

Please note that the operator $\circ$ performs the composition between gates and $\mathrm{costs}(g)$ indicates the implementation cost of the multiple-controlled Toffoli gate g in terms of T gates. If two

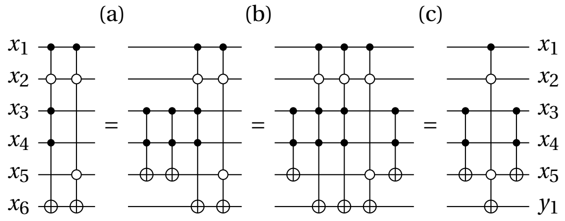


Figure 8.1 – Rule 2 example. Equivalence rules from [3]: (a) D1, (b) D7, (c) D1.

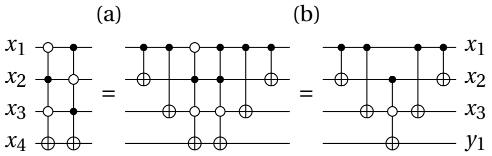


Figure 8.2 – Rule 3 example. Equivalence rules from [3]: (a) D2, (b) D3.

gates are identical, they can both be removed from the network. This property is called *deletion rule* in [3].

Rule 8.1.2 *Let* $g_1 = \mathrm{T}(C_1, t)$ *and* $g_2 = \mathrm{T}(C_2, t)$ *with* $A = C_1 \cap \overline{C}_2$ *and* $B = C_2 \cap \overline{C}_1$.
If the following conditions are verified:

$$\begin{aligned} &\textit{if} \quad l \in C_1 \quad \textit{then} \quad \bar{l} \notin C_2, \\ &\textit{if} \quad l \in C_2 \quad \textit{then} \quad \bar{l} \notin C_1, \\ &\#B = 1, \quad B = \{c\} \end{aligned}$$

where $\#B$ *is the cardinality of the set of controls* B*. Then* $g_1 \circ g_2 = \mathrm{T}(A, |c|) \circ g_2 \circ \mathrm{T}(A, |c|)$.

Rule 8.1.2 is a generalization of some rules presented in [198]. An example is shown in Fig. 8.1, which explains the rule using identities from [3]. Given two Toffoli gates, the rule applies if one gate has a single control on a qubit that is not a control of the other one (x_5 in the example). It is possible to substitute the second gate with two identical gates applied before and after the remaining one. These gates are controlled by the control lines that are not shared with the first gate. This rule leads to cost reduction only if the two initial gates have some identical controls: x_1 and x_2 in the example.

Rule 8.1.3 ([199]) *Let* $g_1 = \mathrm{T}(C_1, t)$ *and* $g_2 = \mathrm{T}(C_2, t)$ *with* $|C_1| = |C_2|$*, i.e.,* g_1 *and* g_2 *share the same set of controls. Let* $D = C_1 \cap \overline{C}_2 = \{c_1, \dots, c_k\}$ *be the set of controls that occur in different polarities in* g_1 *and* g_2*, and let* $\#D > 0$*. Then*

$$g_1 \circ g_2 = \bigcirc_{i=2}^{k} \mathrm{T}(c_1, |c_i|) \circ \mathrm{T}(C_1 \cap C_2, t) \circ \bigcirc_{i=2}^{k} \mathrm{T}(c_1, |c_i|).$$

An example is shown in Fig. 8.2. This rule applies when the first and the second gates have controls on the same lines but with different polarities. It uses two identical CNOT gates before and after the initial gates to complement the polarity of one control (see rule D2 in [3]). This is done until only one control with different polarity remains. Then the pair is equivalent to a single gate with this control removed and all the identical controls maintained (see rule D3 in [3]).

8.1.2 Graph matching problem

The ESOP decomposition of single-target gates produces reversible networks with multiple-controlled Toffoli gates acting on the same target line. There are many pairs of gates on which the described properties could be applied. Different optimization strategies are available depending on which gates are combined.

This section describes an algorithm that selects an optimization strategy that maximizes the T-count reduction. The idea is to derive an optimization graph from the circuit and perform maximum weight matching, similar to how it was done in [200].

Definition 8.1.1 (Optimization graph) *Given a set of generalized Toffoli gates* $g_1 = \mathrm{T}(C_1, t), g_2 = \mathrm{T}(C_2, t), \dots, g_m = \mathrm{T}(C_m, t)$, *the* optimization graph *is an undirected graph* $G = (V, E)$ *with edge weights* $q : E \to \mathbb{N}_0$ *defined as follows:*

$$\begin{aligned} V &= \{g_1, \dots, g_m\} \\ E &= \{\{v, w\} \mid v, w \in V \wedge \\ &\qquad \mathrm{costs}(v \circ w) < \mathrm{costs}(v) + \mathrm{costs}(w)\} \\ q(e) &= \mathrm{costs}(v) + \mathrm{costs}(w) - \mathrm{costs}(v \circ w) \\ &\qquad \textit{where } e = \{v, w\} \end{aligned}$$

In other words, the vertices in G represent the reversible gates. Two vertices are connected if the cost of the corresponding gates combined together is smaller than their accumulated individual cost. The weight on an edge $e = (v, w)$ corresponds to the cost savings that can be achieved when composing the gates corresponding to v and w together.

It is possible to solve the maximum weight matching problem to find the set of graph edges corresponding to the set of combined pairs that leads to the largest gain in terms of cost.

Problem 8.1.1 (Maximum weight matching) *Given a graph* $G = (V, E)$ *with edge weights* $q : E \mapsto \mathbb{N}_0$, *find the graph match* $M \in E$, *i.e., a set of non-adjacent edges, such that* $\sum_{e \in M} q(e)$ *is maximized.*

It follows that, given an optimization graph as defined above $G = (V, E)$ and a graph matching

$M = \{e_1, \dots, e_j\}$, it is possible to realize all generalized Toffoli gates in a circuit $g_1 \circ \cdots \circ g_m$ with

$$\operatorname{costs}(g_1 \circ \cdots \circ g_m) = \sum_{(v,w)\in M} \operatorname{costs}(v \circ w) + \sum_{v \in V_\mathrm{r}} \operatorname{costs}(v)$$

where $V_\mathrm{r} = V \setminus (e_1 \cup \cdots \cup e_j)$.

There is an exact method to solve the maximum weight matching problem in $O(V^2E)$, proposed by Edmonds in 1965 [201]. More recently, an approximate algorithm [202] has been proposed, which runs in linear time.

The described post-synthesis optimization approach has been integrated in RevKit [194]. The efficiency of the method has been evaluated in the context of the k-LUT-based hierarchical synthesis of logic designs, in particular, when single-target gates with large control functions are decomposed into a cascade of multiple-controlled Toffoli gates using 16-LUT decomposition. We reported in [193] a maximum reduction of T gates of 53% for the arithmetic EPFL benchmark using a greedy algorithm with complexity $\Theta(E)$ to solve the maximum weight graph matching problem.

8.2 SAT based {CNOT, T} optimization

This section presents a SAT-based optimization technique that targets Clifford+T circuits, in particular the ones consisting of the Hadamard, the CNOT, and the T gate. In this library, the T gate has been proven to be the most expensive to implement in fault-tolerant circuits [164]. This is the reason why many research works focus on minimizing the T-count. Nevertheless, the CNOT gate is the hardest one to execute on the physical level because it requires to establish an interaction between two adjacent qubits [203].

While there are some methods to efficiently synthesize CNOT circuits [204], in [205] we proposed a SAT-based algorithm to *exactly* synthesize {CNOT, T} circuits from a phase polynomial representation with the minimum number of CNOT gates. In [98] the authors presented an approach that solves the same problem *heuristically*. We then proposed to use this technique to optimize the number of CNOT gates in Clifford+T circuits without increasing the T-count, i.e., the number of T gates in the quantum circuit.

8.2.1 Linear reversible functions

As stated in Section 2.1.1, a multi-output linear Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ can be represented using a $m \times n$ matrix, in which each row is the row vector representing a component linear function f_i. If the multi-output function is linear and reversible the representative matrix is a non-singular matrix $n \times n$.

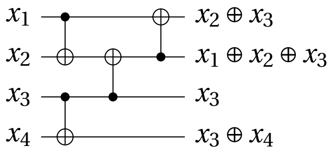


Figure 8.3 – Example of a linear reversible CNOT circuit.

Example 8.2.1 *The controlled-NOT (CNOT) gate, implementing the function*

$$\text{CNOT} : |x_1\rangle|x_2\rangle \mapsto |x_1\rangle|x_1 \oplus x_2\rangle,$$

is both linear and reversible, while the Toffoli gate implements a function that is reversible but not linear:

$$\text{Tof} : |x_1\rangle|x_2\rangle|x_3\rangle \mapsto |x_1\rangle|x_2\rangle|x_3 \oplus (x_1 \wedge x_2)\rangle.$$

By only using CNOT gates it is possible to build a linear reversible circuit with n inputs, implementing a multi-output reversible linear function $f : \mathbb{B}^n \to \mathbb{B}^n$, with n linear reversible Boolean functions as components f_i.

Example 8.2.2 *The linear reversible circuit shown in Fig. 8.3 computes four different linear functions $f_i : \mathbb{B}^4 \to \mathbb{B}$:*

$$f_1 = x_2 \oplus x_3, f_2 = x_1 \oplus x_2 \oplus x_3, f_3 = x_3, f_4 = x_3 \oplus x_4.$$

They can be represented using the row vectors of a $n \times n$ matrix, obtaining a matrix G representing a multi-output linear reversible Boolean function:

$$G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

This representation is used for encoding the SAT problem described in the next section.

8.2.2 SAT-based algorithm for the synthesis of {CNOT, T} circuits

This section describes how to solve the problem of synthesizing quantum circuits specified using the phase polynomial representation, introduced in Section 3.2, using the minimum number of CNOT gates.

Problem 8.2.1 *Given a phase polynomial description of a unitary transformation (g, f_i, c_i) and an integer K, determine if there exists a {CNOT, T} quantum circuit implementing it with*

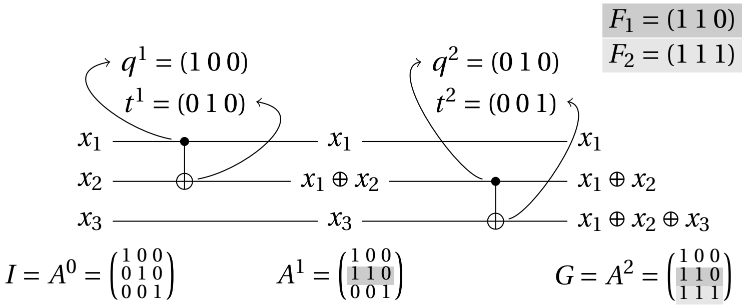
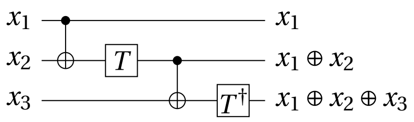

$F_1 = (1\,1\,0)$
$F_2 = (1\,1\,1)$
$q^1 = (1\,0\,0)$
$t^1 = (0\,1\,0)$
$q^2 = (0\,1\,0)$
$t^2 = (0\,0\,1)$
x_1
x_2
x_3
x_1
$x_1 \oplus x_2$
x_3
x_1
$x_1 \oplus x_2$
$x_1 \oplus x_2 \oplus x_3$
$I = A^0 = \begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$
$A^1 = \begin{pmatrix}1&0&0\\1&1&0\\0&0&1\end{pmatrix}$
$G = A^2 = \begin{pmatrix}1&0&0\\1&1&0\\1&1&1\end{pmatrix}$

Figure 8.4 – Illustration of SAT encoding for sample circuit in Example 8.2.3.

K CNOT gates. We denote an instance of this problem HasCNOT(g, f_i, c_i, K).

The linear reversible function g is represented using a $n \times n$ matrix G with entries $G_{i,j}$. F_i is the row vector representation of f_i with entries $F_{i,j}$.

Example 8.2.3 *The phase polynomial representation of the circuit*



is represented by:

$$G = \begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 1 & 1 & 1\end{pmatrix}, \{c_1 = 1, F_1 = (1\,1\,0)\}, \{c_2 = 7, F_2 = (1\,1\,1)\}$$

Encoding

Example 8.2.4 *Fig. 8.4 shows the encoding of the problem of synthesizing a circuit for the phase polynomial representation in Example 8.2.3 using two CNOT gates.*

If the specified transformation is performed using K CNOT gates, there must be K gate transformations that map $A^{k-1} \to A^k$, for $1 \le k \le K$, where A^0 is the identity matrix satisfying:

$$A^K = G \text{ and } \forall_j \exists A_i^k.(A_i^k = F_j) \tag{8.1}$$

The latter means that at least one row A_i^k is equal to the specified linear Boolean functions.

Example 8.2.5 *Considering the example in Fig. 8.4:*

$$A^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, A^1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, A^2 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} = G, \textit{ and}$$

$$A_1^1 = A_1^2 = (1\,1\,0) = F_1, A_2^2 = (1\,1\,1) = F_2$$

Each CNOT gate is represented by two vectors: $q^k = (q_1^k \dots q_n^k)$ describing the gate control and $t^k = (t_1^k \dots t_n^k)$ describing the gate target, where $q_i^k(t_i^k) = 1$ only if the i^{th} line is a control (target) of the gate (see Fig. 8.4).

First, the encoding must ensure that these variables are describing valid CNOT gates, characterized by one control and one target line. The following one-hot clauses are defined:

$$\forall_{1\leq k\leq K}[(q_1^k \vee \dots \vee q_n^k) \wedge \bigwedge_{1\leq i<j\leq n} (\bar{q}_i^k \vee \bar{q}_j^k)] \tag{8.2}$$

and

$$\forall_{1\leq k\leq K}[(t_1^k \vee \dots \vee t_n^k) \wedge \bigwedge_{1\leq i<j\leq n} (\bar{t}_i^k \vee \bar{t}_j^k)] \tag{8.3}$$

In addition, the control and target of each gate need to be acting on different lines to represent a valid CNOT:

$$\forall_{1\leq k\leq K} \bigwedge_{i=1}^{n} (q_i^k \neq t_i^k) \tag{8.4}$$

Once the matrices $A_0, \dots, A_K$ (8.1) and the conditions over the gate vectors q^k and t^k describing the mapping $A^{k-1} \mapsto A^k$ (8.2),(8.3),(8.4) are defined, the functionality of this mapping must be encoded. Some intermediate expressions h_j^k for each row of a matrix A^{k-1} are used to determine whether the row intersects with the control vector q^k:

$$\forall_{1\leq k\leq K}, \forall_{j=1}^{n} h_j^k = \bigoplus_{i=1}^{n} A_{ij}^{k-1} \wedge q_i^k$$

By means of such variables, it is possible to define in which cases the application of a CNOT gate causes an element of the matrix A to switch.

$$\forall_{1\leq k\leq K}, \forall_{i=1}^{n}, \forall_{j=1}^{n} A_{i,j}^k = A_{i,j}^{k-1} \oplus (t_i^k \wedge h_j^k) \tag{8.5}$$

Example 8.2.6 *Consider the first CNOT gate in Fig 8.4, it can be represented by the vectors:*

$$q^1 = (1\,0\,0) \textit{ and } t^1 = (0\,1\,0)$$

verifying conditions (8.2),(8.3),(8.4). One can compute the intermediate variables h_j^1 checking *whether one or more elements in the matrix A^0 should switch.*

$$h_1^1 = A_{1,1}^0 \wedge q_1^1 \oplus A_{2,1}^0 \wedge q_2^1 \oplus A_{3,1}^0 \wedge q_3^1 = 1$$
$$h_2^1 = A_{1,2}^0 \wedge q_1^1 \oplus A_{2,2}^0 \wedge q_2^1 \oplus A_{3,2}^0 \wedge q_3^1 = 0$$
$$h_3^1 = A_{1,3}^0 \wedge q_1^1 \oplus A_{2,3}^0 \wedge q_2^1 \oplus A_{3,3}^0 \wedge q_3^1 = 0$$

Then, the only element in the matrix A that switches is $A_{2,1}^1 = A_{2,1}^0 \oplus (t_2^1 \wedge h_1^1)$ and:

$$A^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mapsto A^1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SAT problem encoding: summary

The specification of the SAT problem is a phase polynomial representation: a matrix G representing a linear reversible Boolean function g, the set of coefficients c_i and the linear Boolean functions f_i.

SAT variables The following variables are declared for each gate: (i) control variables q_i^k, defining which line i is the control of gate k, (ii) target variables t_i^k, defining which line i is the target of gate k.

In order to evaluate the evolution of the implemented function gate by gate, matrices representing intermediate synthesized linear transformations are defined: A^k for $1 \leq k \leq K$. In addition, some auxiliary variables h_j^k are used to describe the mapping $A^{k-1} \mapsto A^k$

SAT clauses The following clauses define the SAT problem:

- Initial clauses: $A^0 = I$ where I is the identity matrix.
- Final clauses: $A^K = G$ and $\forall_j \exists A_i^k.(A_i^k = F_j)$. The final linear transformation implements the desired G and all the functions f_i are computed at some time by the circuit so that c_i $\pi/4$ phase shifts can be applied.
- Validity clauses: only one variable q_i^k (t_i^k) is equal to one for each gate k and $q^k \neq t^k$. Every CNOT gate has one target and one control, on different lines.
- Dependency clauses: $A_{i,j}^k = A_{i,j}^{k-1} \oplus (t_i^k \wedge h_j^k)$. They define the relation between the gate variables q_i^k, t_i^k and the intermediate matrix variables. These clauses define the mapping $A^{k-1} \mapsto A^k$. Considering all the K gates, $I = A^0 \mapsto A^1 \mapsto \ldots \mapsto A^{K-1} \mapsto A^K = G$.

8.2.3 SAT-based rewriting algorithm

This section describes the rewriting algorithm proposed to minimize the number of CNOT gates without increasing the number of T gates. The starting point is a Clifford+T circuit with an optimized number of T gates, obtained using the optimization algorithm T-par [108]. The overall procedure is described by Alg. 8.1.

Algorithm 8.1 SAT-based rewriting algorithm

```
$c \leftarrow$ T-optimized quantum circuit
ct_extract($c$)    (extract {CNOT, T} sub-circuits)
for all sub-circuit $c'$ do
    $(g, f_i, c_i) \leftarrow$ phase($c'$)    (set the phase polynomial representation from $c'$)
    $K \leftarrow 0$    (number of gates)
    repeat
        Solve(HasCNOT($g, f_i, c_i, K$))
        if SAT then
            $c''$ ←Extract {CNOT, $T$} circuit
            Replace $c'$ by $c''$ in $c$
        else if  UNSAT then
            $K \leftarrow K + 1$
    until a solution is found
```

The first step of the algorithm is to extract from the input {CNOT, T, H} circuit, some {CNOT, T} sub-circuits. This partition is required because the algorithm aims at minimizing quantum circuits defined over the Clifford+T universal library, using an exact method for reducing {CNOT, T} circuits. The extraction method *ct_extract* is performed in such a way that after the rewriting of the sub-circuits, they can be recombined to restore the initial functionality.

The next step is to re-synthesize each sub-circuit. First, the phase polynomial representation (g, f_i, c_i) is extracted by the procedure *phase.* Then, the SAT solver is used iteratively to solve the encoded problem *HasCNOT.* It tries to find a satisfying solution with K gates and adds an additional gate if the problem is unsatisfiable. This procedure is repeated until a valid solution is found. This solution will describe a circuit with the minimum number of CNOT gates, given the phase polynomial representation.

8.2.4 Results

Algorithm 8.1 is implemented in C++ on top of RevKit [194] and the Z3 prover [148] is used to encode and solve the SAT problem. The selected benchmark consists of Clifford+T circuits for the 48 spectral equivalent representatives of the 5-input Boolean function. The proposed optimization algorithm can reduce the number of CNOT gates by keeping the T-count unchanged.

Results are shown in Table 8.1, which reports for each equivalent class: (i) the hexadecimal encoding of the representative function, (ii) the number of T gates in the initial circuit, (iii) the number of CNOT in the initial circuit, (iv) the reduced number of CNOT after SAT-based rewriting, (v) the percentage reduction and (vi) the runtime in seconds. The table also reports the average percentage reduction and the maximum measured reduction, 26.84% and 45.45%, respectively. The method used to decompose the initial Clifford+T circuit into {CNOT, T} sub-circuits, namely *ct_extract* impacts the final optimized results and the runtime. As it was expected, the presence of larger sub-circuits leaves larger space for optimization, but slows down the operation of the SAT solver.

8.3 Summary

This chapter discussed two different optimization methods. The first one, which aims to reduce the T-count of quantum circuits derived from the decomposition of reversible networks, is based on solving instances of the maximum weight graph matching problem. This method is described in Section 8.1. Section 8.2 presented a second technique dedicated to the optimization of the number of CNOT gates in Clifford+T circuits. This technique rewrites parts of the original network exploiting an exact SAT-based method for the synthesis of {CNOT, T} circuits. Targeting the reduction of CNOT gates, this optimization technique is also compliant with the requirements of NISQ systems. For this reason, future works should extend the algorithm to support the optimization of circuits composed of the X gate, the CNOT gate, and the phase gate with continuous rotation angles. In addition, the SAT-encoding can be extended to take into account qubit connectivity.

Table 8.1 – Results of the SAT-based rewriting algorithm.

Class	T-count	CNOT initial	CNOT final	%CNOT	Runtime
00000000	0	0	0	0.00	0.00
80000000	31	61	55	9.84	137.24
80008000	24	45	38	15.56	8.68
00808080	51	112	97	13.39	24.93
80808080	16	47	30	36.17	7.82
88800800	48	94	79	15.96	13.81
88088020	75	175	143	18.29	574.99
88808080	47	100	79	21.00	133.77
2a808080	32	89	54	39.33	202.16
70080088	56	127	96	24.41	814.43
f3c0dd00	48	118	79	33.05	257.79
c0c8c0c8	29	65	54	16.92	9.46
734470c8	111	221	176	20.36	2148.00
e0a0c000	63	156	111	28.85	135.59
e8080808	71	178	128	28.09	293.18
8808a808	63	136	102	25.00	17.44
08888888	36	82	73	10.98	445.86
88888888	7	11	6	45.45	0.92
d5808080	32	89	58	34.83	215.22
70807080	15	40	23	42.50	3.08
e1808880	88	194	130	32.99	16.48
ea808080	56	140	84	40.00	10.14
cc808880	55	117	88	24.79	10.53
e4404440	55	118	83	29.66	5.71
7f008000	23	39	32	17.95	4.22
e0a8c800	91	219	161	26.48	821.58
e8818880	115	291	210	27.84	832.18
e8a08880	80	195	168	13.85	368.34
f8808880	80	182	143	21.43	57.50
e2222220	56	123	85	30.89	1771.19
a0c8a088	63	192	149	22.40	385.03
e6804c80	39	96	58	39.58	5.94
7f808080	19	60	44	26.67	388.19
0231da51	79	192	138	28.12	8.77
a008bc88	95	204	151	25.98	50.76
d8887888	43	112	70	37.50	360.20
eca08088	80	180	118	34.44	10.77
f0888888	56	144	97	32.64	860.57
8a80cac0	47	108	82	24.07	523.50
78807880	36	74	56	24.32	1668.62
bca08488	79	208	154	25.96	863.73
fca08880	96	225	153	32.00	53.71
dac08a80	76	190	157	17.37	356.73
f8887888	43	107	91	14.95	363.48
78887888	12	24	15	37.50	869.91
a6cc60a0	47	126	83	34.13	17.35
62c8ea40	35	95	63	33.68	25.30
6ac8e240	48	107	81	24.30	11.43
Average achieved CNOT minimization: 26.84%					
Max achieved CNOT minimization: 45.45%					

Part III

Compiler-integrated accuracy management

9 Automatic accuracy management

Concrete resource estimates for problems and corresponding problem sizes at which quantum computers are expected to outperform their classical counterparts are scarce. To carry out such resource estimates, several quantum programming languages and toolchains have been developed such as Q# [55], Quipper [110], Scaffold/ScaffCC [60], Qiskit [56], ProjectQ [61], and QuRE [206]. Despite the availability of these languages, there is still a significant amount of manual work involved in resource estimation [111, 112]—one reason being the lack of built-in support for handling approximation errors.

Approximation errors must be taken into account when compiling quantum programs into a low-level gate set. With the existing languages, programmers must manually keep track of all the introduced errors. Furthermore, programmers must tune the parameters of their implementation to keep the total error within a target budget. To guarantee this, they must derive the resulting error bounds manually—a task that is tedious and error prone.

To address this issue, I present a methodology to introduce language support into existing quantum programming languages, allowing programmers to deal independently with the approximation errors in each subroutine. The job of inferring how all introduced approximation errors interact is thus transferred from the programmer to the compiler. The proposed methodology automatically infers an error bound for the overall quantum program and then selects appropriate values for each of the program's accuracy parameters to simultaneously (1) satisfy a user-specified overall tolerance and (2) reduce the required quantum resources.

More specifically, the methodology supports:

- given the desired approximation error, determining the assignment of accuracy parameters that guarantees the given approximation error while aiming to minimize the number of operations;
- given a maximal operation count, determining the assignment of accuracy parameters that yield at most the given operation count while aiming to reduce the total approximation error.

The automatic optimization of accuracy parameters is carried out by solving an optimization problem before the quantum circuit is generated. The constraint and cost functions describing the optimization problem are extracted by the compiler directly from the source code of the quantum program. Then, before execution, the optimized accuracy parameters are fed into the main program.

Finding a suitable assignment of accuracy parameters using, e.g., simulated annealing [207, 208], requires hundreds of evaluations of both constraint and cost functions. Hence, the methodology has to lean on a fast method to estimate resource requirements and the total approximation error. Available methods, e.g., in Q#, estimate resources by actually generating quantum circuits from completely specified programs and then counting the generated gates. Hence, their runtime will increase with increasing problem size, making these methods ill-suited to evaluate cost functions in an optimization procedure, as shown in Section 9.8.

Instead, a fast symbolic method is proposed, which extract a symbolic expression for the desired cost or constraint function (total approximation error or number of gates) directly from the source code of the quantum program. The resulting expressions feature variables that correspond to the various parameters of the program, including accuracy parameters. The symbolic approach does not need to execute the complete control flow of the quantum program to get an estimate, hence it provides a much faster solution that is viable even for application-scale programs.

The presented methodology can be integrated into any quantum programming language compiler. Two different prototypes are developed: one integrated in the *Low-Level Virtual Machine (LLVM)* project and one in Q#. Both prototypes implement a symbolic approach for resource and error estimation. Since the resulting expressions may theoretically still contain some residual code that must be executed (e.g., certain if-else statements), they are referred to as *(near-)symbolic*. The parentheses indicate that, for most applications, the resulting expressions would be fully symbolic. However, there are examples where this is not the case, e.g., a quantum program that executes one of two different algorithms depending on a runtime parameter. However, both prototypes generate fully symbolic expressions for all the examples in this thesis.

The chapter is structured as follows: Section 9.1 introduces common sources of errors in quantum compilation; Section 9.2 describes how such approximation errors compose in quantum circuits; Section 9.3 describes some famous quantum algorithms that will be later used for the experimental evaluations; Section 9.4 introduces language support to define approximation parameters in a quantum program; Section 9.5 describes how to automate the process of optimizing such approximation parameters; Section 9.6 lists all the features that a compiler of a quantum programming language should support to integrate the proposed methodology and describes how such features are implemented in the developed prototypes; finally Section 9.7 and Section 9.8 provide experimental evaluations of the LLVM prototype.

9.1 Errors in quantum compilation

Why do approximation errors occur in quantum programs in the first place? This section discusses three main sources of errors, however, the framework is extensible to other sources of errors.

9.1.1 Synthesis errors

Due to the discrete nature of fault-tolerant instruction sets (and indeed, it is known that any universal fault-tolerant instruction set necessarily must be discrete [64]), it cannot be avoided to introduce approximation errors for general target operations. For instance, consider a rotation around the Z-axis such as

$$R_Z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix},$$

which can be defined for any $\theta \in [0, 4\pi)$. These rotations can only be implemented exactly for a discrete subset of the interval $[0, 4\pi)$, as gates have to be expressed as words of finite length over any universal set of generators. It should be noted that there is a mathematical function that expresses the length of the approximating word in terms of an approximation error, which is called ε_R in this thesis. This mathematical function depends on the concrete synthesis algorithm used to perform the factorization into fault-tolerant instructions. State-of-the-art synthesis algorithms lead to a cost (e.g., number of T gates, where $T = e^{i\pi/8}R_Z(\pi/4)$) that is proportional to $\log_2(\varepsilon_R^{-1})$ [209, 210].

9.1.2 Phase estimation errors

An important technique in quantum computing, used by many quantum algorithms [18, 211], is to extract estimates of an eigenvalue λ of an operator U to k bits of precision. A common method to achieve this is to prepare an eigenstate $|\psi_\lambda\rangle$ of U and to then apply powers U^{2^i}, for $i = 0, \ldots, k-1$, to the eigenstate $|\psi_\lambda\rangle$. This application is done conditionally on the value of a reference system and allows us to extract the k most significant bits of the eigenvalue. As λ can in principle be any complex number of the form $\lambda = e^{i\alpha}$, where $\alpha \in [0, 2\pi)$, the particular choice of k introduces an approximation error and limits how precisely λ can be estimated. The resulting approximation error is called ε_{QPE} in this thesis.

9.1.3 Algorithmic errors

Some quantum programs are part of a parametric family of programs that gracefully degenerate with a reduction of the parameters. A concrete example for such a family of programs is the quantum Fourier transform [18], or QFT for short. While the transformation itself can be implemented exactly and with no approximation error over a gate set that includes continuous rotations such as $R_Z(\theta)$ for arbitrary $\theta \in [0, 4\pi)$, it is possible to approximate the transformation by selectively dropping some of the rotations that occur, in particular by "pruning" the values of θ that are very close to 0. One such pruning method is well known [212] and allows us to drop many of the $O(n^2)$ rotations that a simple implementation of the Fourier transform requires and just retain $O(n \log n)$ rotations, while still maintaining a sufficient approximation. The resulting approximation error is called ε_{QFT} in this thesis. Another example of algorithmic error comes from formulas that are known to converge to a target program when taking a suitable limit, e.g., of alternations of other, typically smaller and simpler, programs. An example for the latter is the so-called Trotter formula, a well-known identity to implement an approximation to $e^{i(A+B)}$ for Hermitian matrices A and B, from the knowledge of implementations for e^{iA} and e^{iB}. The resulting approximation error (the "Trotter error") is called ε_{TE} in this thesis.

9.2 Approximation errors in quantum circuits

As mentioned, the framework addresses approximation errors that may be reduced at an increased implementation cost. For the example of *synthesis errors*, which occur due to the discrete nature of fault-tolerant instruction sets, the trade-off between the number of T gates and the approximation error is logarithmic [209]. That is, to achieve an approximation error ε, $O(\log(1/\varepsilon))$ T gates are sufficient. In a typical quantum program, multiple different sources of such errors are present and thus the question arises: *How do the approximation errors of individual operations compose?*

It is possible to derive an upper bound on the total approximation error by summing up all the individual approximation errors. Consequently, the proposed methodology produces quantum circuits with accuracy guarantees for quantum programs without measurement feedback. Note that this excludes programs that rely on repeat-until-success statements, i.e., loops that iterate until a certain measurement outcome is observed (see, e.g., [213]). Such cases can be handled separately using, e.g., an upper bound on the number of iterations. In the last section, I propose a possible way of enabling accuracy management for such programs as future work. In all other cases, branching on measurement results may be addressed using an expression for the total error of the form $\varepsilon_b = \max(\varepsilon_1, \varepsilon_2)$, where ε_1 and ε_2 denote the errors of each branch. This follows from the *Deferred Measurement Principle* [64, Section 4.4], which says that measurements may be delayed until the end of the computation, transforming all quantum operations that are executed conditionally on the measurement result into quantum-controlled operations.

Quantum circuits corresponding to feedback-free programs consist of a sequence of gates $U_1, \ldots, U_m$, followed by a sequence of measurements $M_1, \ldots, M_k$ that produce a measurement outcome $m = x_i$ for a final state

$$|\psi\rangle = U_m \cdots U_1 |0\rangle^{\otimes N}$$

with probability

$$P(m = x_i) = |\langle x_i|\psi\rangle|^2,$$

where $\langle x_i|\psi\rangle$ denotes the overlap of $|\psi\rangle$ with $|x_i\rangle$.

Therefore, the methodology must ensure that the actual final state $|\tilde{\psi}\rangle$ is close to the desired final state $|\psi\rangle$ after all decompositions have been applied to $U_1,\dots,U_m$.

Let $V_1,\dots,V_n$ be an approximate decomposition of the quantum program in terms of the gates supported by the target hardware, i.e.,

$$\| \underbrace{U_m \cdots U_1}_{U} - \underbrace{V_n \cdots V_1}_{V} \| \le \varepsilon,$$

where $\|\cdot\|$ denotes the spectral norm as defined in [64, Section 2.1.4]. Then, $\| |\psi\rangle - |\tilde{\psi}\rangle \| = \| U|0\rangle^{\otimes N} - V|0\rangle^{\otimes N} \| \le \varepsilon$, which guarantees that, with $|\psi\rangle = \sum_i a_i |x_i\rangle$ and $|\tilde{\psi}\rangle = \sum_i \tilde{a}_i |x_i\rangle$,

$$\begin{aligned} |a_i - \tilde{a}_i| &= \sqrt{|a_i - \tilde{a}_i|^2} \\ &\le \sqrt{\sum_i |a_i - \tilde{a}_i|^2} \\ &\le \varepsilon. \end{aligned}$$

Therefore, it is sufficient that the methodology guarantees

$$\|U - V\| \le \varepsilon.$$

For more details on approximated unitary operators, refer to Box 4.1 *"Approximating quantum circuits"* of [64].

In the process of translating the quantum program to the native gate set, several decompositions are applied that introduce approximation errors. Let U be a quantum operation being approximated by the decomposition into $W_1,\dots,W_t$. The decomposition introduces at most ε_U if $\|U - (W_t \cdots W_1)\| \le \varepsilon_U$, assuming that all W_i are implemented exactly. Now, combining multiple such approximate implementations $\tilde{U}_i$ of U_i such that $\|U_i - \tilde{U}_i\| \le \varepsilon_i$ yields a total error of at most $\sum_i \varepsilon_i$ [214].

Therefore, it is possible to derive recursively-defined expressions for the error $E(U,\varepsilon_U)$ and the total gate count $T(U,\varepsilon_U)$ [208]:

$$\begin{aligned} E(U,\varepsilon_U) &= \varepsilon_U + \sum_{W \in D(U,\varepsilon_U)} E(W,\varepsilon_W) f_W(\varepsilon_U), \\ T(U,\varepsilon_U) &= \sum_{W \in D(U,\varepsilon_U)} T(W,\varepsilon_W) f_W(\varepsilon_U), \end{aligned}$$

where $D(U,\varepsilon_U)$ is the set of gates in the ε_U-approximate decomposition of U and $f_W(\varepsilon_U)$ denotes the number of W operations in the decomposition. Looking at these expressions, it is clear that an upper-bound on the total error can be computed very similarly to counting gates.

In conclusion, by applying this reasoning to the main entry point of a quantum program, one

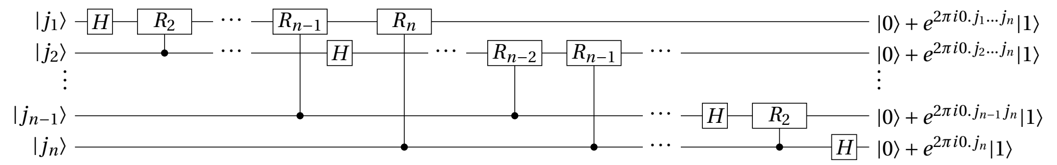

$|j_1\rangle$
$|j_2\rangle$
$|j_{n-1}\rangle$
$|j_n\rangle$
H
R_2
R_{n-1}
R_n
R_{n-2}
$|0\rangle + e^{2\pi i 0.j_1 \dots j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_2 \dots j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_n}|1\rangle$

Figure 9.1 – Efficient circuit computing the quantum Fourier transform.

can choose all $\varepsilon_{(\cdot)}$ in the expression for

$$E(U_{Main}, \varepsilon_{U_{Main}})$$

such that $E(U_{Main}, \varepsilon_{U_{Main}}) \leq \varepsilon$. This ensures that the measurement probability amplitude for a given bit-string changes by at most ε.

9.3 Sample quantum programs

This section describes some famous quantum algorithms, that will be used to test the implemented LLVM prototype.

9.3.1 Quantum Fourier transform

The quantum Fourier transform (QFT) is an algorithm that performs the Fourier transform of quantum mechanical amplitudes. QFT is implemented as a linear operator that applies the following unitary transformation on a basis state $|j\rangle$ [64]:

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i jk/N} |k\rangle$$

The effect of the transform on an arbitrary state can be described using a product representation that maps $|j_1, \dots, j_n\rangle$ to

$$\frac{(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle)\cdots(|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_n}|1\rangle)}{2^{n/2}}$$

where $0.j_l j_{l+1} \dots j_n$ is the binary expansion $j_l/2 + j_{l+1}/4 + \cdots + j_n/2^{n-l+1}$. This representation has a direct correspondence to the circuit implementation of the quantum Fourier transform shown in Fig. 9.1. The circuit consists of n steps, one for each qubit. In each step, for each qubit, a Hadamard gate is applied, followed by a series of rotation gates controlled by all remaining qubits.

In this work, I will largely refer to an approximate version of QFT (AQFT) in which the number of rotations is reduced according to the desired approximation error ε_{QFT}. This is done by pruning the rotations with small angles. In particular, for each qubit j_i with $1 \leq i < n$ a maximum of

$$l = \lceil \log_2(n/\varepsilon_{QFT}) \rceil + 3$$

controlled-rotations is applied [212].

The quantum Fourier transform enables the *quantum phase estimation* algorithm and has a key role in the solution of many relevant problems, e.g., the *integer factorization* problem.

9.3.2 Simulating time-evolution of operators

Being a quantum system, a quantum computer can be programmed to simulate other quantum systems. This, for example, can be used to elucidate chemical reaction mechanisms [111].

Given the quantum-mechanical Hamiltonian $\mathscr{H}$ which describes the system being studied, the time evolution of the system can be simulated by implementing the *time-evolution operator* $U = e^{-i\mathscr{H}t}$. The time-evolved quantum state can then be obtained by applying U to the initial state $|\psi(0)\rangle$:

$$|\psi(t)\rangle = e^{-i\mathscr{H}t}|\psi(0)\rangle$$

In order to implement the time-evolution operator on a quantum computer, it needs to be decomposed into the native gate set, e.g., Clifford+T. Different decomposition methods are available, each one with a different scaling with respect to the targeted precision ε_{TE}: polynomial for the Trotter decomposition method, logarithmic for the linear combination of unitaries (LCU) [72, 215].

Example 9.3.1 *Consider the Hamiltonian of a 1D transverse-field Ising model (TFIM)*

$$\mathscr{H} = \underbrace{-J\sum_{\langle i,j\rangle} Z^i Z^j}_{\mathscr{H}_1} \underbrace{- h\sum_i X^i}_{\mathscr{H}_2},$$

where $\mathscr{H}_1$ defines the interaction of adjacent spins, denoted by $\langle i,j\rangle$, with periodic boundary conditions, $\mathscr{H}_2$ defines the interaction of the system with the external transverse field, and X^i, Z^i denote the application of $X = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ and $Z = \left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$, respectively, to spin i.

Using a second-order Trotter-Suzuki decomposition, the time-evolution operator under this Hamiltonian can be written as

$$e^{-i\mathscr{H}t} \approx (e^{-i\mathscr{H}_1\frac{t}{2M}} e^{i\mathscr{H}_2\frac{t}{M}} e^{-i\mathscr{H}_1\frac{t}{2M}})^M.$$

The number of Trotter steps M will be chosen according to the desired accuracy ε_{TE}. In particular, for this second-order Trotter decomposition, M is proportional to $1/\sqrt{\varepsilon_{TE}}$ [111]. Each Trotter step can be implemented using CNOT and $R_Z(\theta)$ gates. The latter being a gate that applies a rotation equal to the angle θ around the z-axis. Considering Clifford+T as the native gate set, each rotation has to be synthesized or decomposed in terms of these gates. As not every rotation

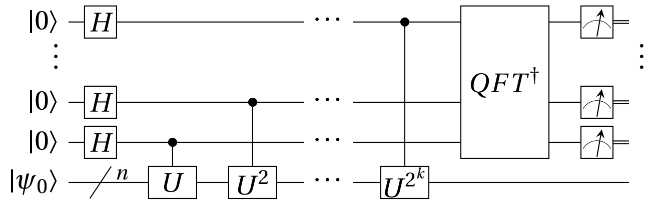


Figure 9.2 – Quantum circuit performing quantum-phase estimation on an n-qubit system with an accuracy of $k+1$ bits.

can be realized exactly using this gate library, rotation synthesis also introduces an error. Given a target approximation error ε_R*, the number of T gates per rotation will be proportional to* $\log_2\left(\frac{1}{\varepsilon_R}\right)$.

Thus, to express the time-evolution operator, two inter-dependent approximation errors must be taken into account, namely ε_{TE} *and* ε_R.

9.3.3 Quantum phase estimation

Once time-evolution under the Hamiltonian $\mathscr{H}$ is implemented, one may perform measurements similar to experiments with the actual system. Quantum computing, however, allows us to achieve a quadratic advantage over repeated measurement and sampling via quantum phase estimation (QPE). Given a state with large overlap with the ground state $|\psi_0\rangle$ of the Hamiltonian, this algorithm determines the ground state energy E_0

$$\mathscr{H}|\psi_0\rangle = E_0|\psi_0\rangle.$$

Fig. 9.2 shows one of several possible implementations of QPE [64]. The measurement outcomes of the top $k+1$ qubits yield a $k+1$-bit approximation to the phase due to time-evolution. More precisely, the number of qubits to choose n_{QPE} depends on the desired accuracy *and* the probability p of a successful measurement as

$$n_{QPE} = n + \left\lceil \log\left(2 + \frac{1}{2(1-p)}\right)\right\rceil,$$

where n is the desired accuracy in number of bits.

The number of controlled time-evolution unitaries required for QPE to succeed with $p = 0.5$ and accuracy ε_{QPE} may thus be bounded by $2^{n_{QPE}} - 1 \le 16\pi/\varepsilon_{QPE}$.

Not only does QPE allow one to infer the ground state energy if the ground state is known, but it also collapses a non-eigenstate input $|\phi\rangle$ to the i-th eigenstate $|\psi_i\rangle$ of the Hamiltonian $\mathscr{H}$ with probability $|\langle\psi_i|\phi\rangle|^2$.

In terms of accuracy, it is important to distinguish between the different applications of QPE. If QPE is used to determine only the energy, i.e., $|E_0 - \tilde{E}_0| \le \varepsilon$ is required, then it is sufficient to implement time-evolution such that $\|U - \tilde{U}\| \leq \varepsilon - \varepsilon_{QPE}$. However, if the goal is to prepare the ground state, i.e., $\| |\psi_0\rangle - |\tilde{\psi}_0\rangle \| \leq \varepsilon$, then $\|U - \tilde{U}\| \leq \frac{\varepsilon - \varepsilon_{QPE}}{2^{n_{QPE}-1}}$ is sufficient (both via triangle inequality). Distinguishing these cases clearly has a great impact on the resulting resource requirements.

9.4 Adding language support for accuracy management

Since large-scale quantum computers are not yet available, resource estimation is a crucial feature of any software framework for quantum computing. Typically, resource estimation is performed by compiling the quantum program into the chosen target gate set and then executing the resulting circuit on a classical simulator that counts native operations (instead of executing them). For this to be possible, however, all the parameters of the program, including accuracy parameters for each subroutine, must be determined.

Existing quantum programming languages do not offer built-in support for accuracy management. Consequently, it is very cumbersome to implement large-scale quantum algorithms in an accuracy-aware fashion. Thus, despite the availability of a wide range of quantum programming languages, resource estimates are still computed (semi-)manually, taking care of accuracy parameters using pen and paper [112, 111].

The main difficulty when selecting appropriate accuracy parameters is that parameters at a higher level of abstraction have an effect on the ones at lower levels, as illustrated by the following example:

Example 9.4.1 *Consider QPE on $U = R_Z(\alpha) := e^{-i\frac{\alpha}{2}Z}$ and the target gate set Clifford+T. The number of phase-estimation qubits n_{QPE} depends on the desired precision of the phase (and the probability of success). At the lowest level, the various $U^{2^i} = R_Z(\alpha_i)$ are decomposed into a sequence of Clifford+T gates featuring $O(\log\frac{1}{\varepsilon_r})$ T gates, where ε_r is the error of a single rotation. To achieve an overall target accuracy ε, ε_r must be chosen such that $\varepsilon_{QPE} + \varepsilon_R \leq \varepsilon$, where ε_R denotes the error introduced by all rotations in the quantum circuit. If the same accuracy is chosen for all the rotations, then $\epsilon_R = N_{rot}\epsilon_r$, where N_{rot} is the number of rotations. Since ε_{QPE} affects the number of rotations in the circuit, ε_r must be chosen as a function of ε_{QPE}.*

In general, it would be possible to adapt all values in the code manually on a case-by-case basis—however, this defeats the purpose of having a high-level programming language. The lack of language support forces programmers to manually handle accuracy parameters by passing all such parameters to the main routine, which forwards them to each subroutine. In case of the QPE algorithm, this might be result in code as follows:

```
**function** QPE($\varepsilon_{QPE}$, $\varepsilon_{TE}$,$\varepsilon_{QFT}$, $\varepsilon_{R_{TE}}$, $\varepsilon_{R_{QFT}}$, U)
    $reg\_size \leftarrow f(\varepsilon_{QPE})$
    **for** $i \leftarrow 0$ to $reg\_size$ **do**
        **for** $j \leftarrow 0$ to $n\_iter(i)$ **do**
            ${}^c$U($\varepsilon_{TE}$, $\varepsilon_{R_{TE}}$)
    AQFT$^\dagger$($\varepsilon_{QFT}$, $\varepsilon_{R_{QFT}}$)
```

Here, ${}^cU$ is the controlled version of U. Note that all qubit variables are omitted for better readability.

This programmer-unfriendly approach does not allow code reuse for resource estimation, as implementations of subroutines need to be adapted to the context in which they are used and, in particular, to the choice of accuracy parameters. For example, ^{c}U(ε_{TE}, $\varepsilon_{R_{TE}}$) is implemented using a number of Clifford+T gates that depends on the chosen accuracy parameters.

When using the proposed methodology, programmers need to worry about accuracy parameters only in subroutines where the corresponding errors are introduced. The compiler will take care of extracting all dependencies. Specifically, the pseudo-code for phase estimation can be expressed as follows:

```
**function** ${}^c$R
    declare $\varepsilon_R$
    ...
**function** ${}^c$U
    declare $\varepsilon_{TE}$
    ...
**function** AQFT$^\dagger$
    declare $\varepsilon_{QFT}$
    $l \leftarrow g(\varepsilon_{QFT})$
    **for** $i \leftarrow 0$ to $g'(l)$ **do**
        ...
        ${}^c$R()
        ...
```

```
**function** QPE(U)
    declare $\varepsilon_{QPE}$
    $reg\_size \leftarrow f(\varepsilon_{QPE})$
    **for** $i \leftarrow 0$ to $reg\_size$ **do**
        **for** $j \leftarrow 0$ to $n\_iter(i)$ **do**
            ${}^c$U()
    AQFT$^\dagger$()
```

Using Abstract Syntax Tree (AST) transformations, the methodology is able to handle various levels of granularity: from using the same value for all accuracy parameters to using a different value for every instance that is created during runtime (via an accuracy parameter data structure that mirrors the call graph). This is crucial as there is a substantial trade-off between the number of accuracy parameters being considered and the resulting gate count [208]. This can be illustrated with the following example:

Example 9.4.2 *Consider Beauregard's implementation of Shor's algorithm [216]. In addition to the (semi-classical) inverse Fourier transform of phase estimation, every addition circuit requires two (approximate) QFTs [212] (one inverted, one regular) [217]. While the number of*

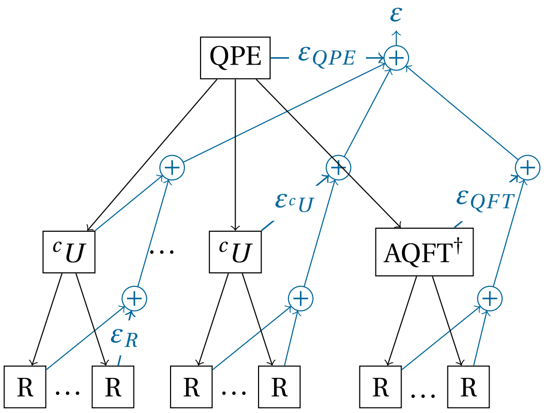


Figure 9.3 – Flow diagram explaining how the code of the QPE algorithm is transformed into a code evaluating the overall approximation error ε.

additions (and thus the number of QFTs) varies with the bit-size n of the number to factor, phase estimation always requires a single QFT. Therefore, it is natural to choose a different accuracy parameter for the (approximate) QFT of the phase estimation than for the (approximate) QFTs of the $O(n^2)$-many n-bit additions.

Besides facilitating accuracy-aware implementations of quantum programs and providing various levels of granularity for assigning accuracy parameters, the methodology allows us to automatically deduce the number of contexts in which a given (approximate) decomposition is applied. This enables automatic selection of the number of accuracy parameters and thus removes the need to perform this task manually.

9.5 Automating accuracy management

The previous section has shown how the quantum programming language can enable the user to define accuracy parameters in the program. In this section, I describe the proposed procedure to automatically determine appropriate values for such accuracy parameters. The optimization problem is defined by two functions: the constraint (total accuracy) and the cost function (circuit cost in terms of expensive gates), both generated by the compiler from the source code. The result is a valid distribution of the available approximation error that in addition aims to minimize the circuit cost.

The prototypes developed in this work use the two-mode simulated annealing described in [208] to solve the optimization problem. Note that any other optimization method accepting a cost and a constraint function could be used instead. The two-mode simulated annealing procedure in [208] activates the first mode when the constraint function is larger than the target accuracy and performs annealing until the target accuracy is reached. Then, it activates the second mode to reduce the circuit cost function. It goes back the the first mode if the constrained function was increased beyond the allowed bound. Both modes iteratively change the value of a randomly chosen accuracy parameter by multiplying it for a random factor $f \in (1, 1+\delta]$, where δ is chosen to achieve an acceptance rate of about 50%. The proposed change is accepted with probability $p = min(1, e^{-\beta\Delta E})$, where β is the inverse of the annealing temperature. Please refer to [208] for a detailed description of the algorithm.

The proposed approach is to extract a (near-)symbolic expression for the total error and gate count from the algorithm and to use the obtained expressions in the annealing procedure. In general, it is possible to use the annealing procedure with any available resource estimation method, the difference being that the runtime of each evaluation depends on the problem size if non-symbolic methods are used.

In the following sections, the T-count is selected as a measure of the implementation cost of a quantum circuit, as this closely captures the cost in a fault-tolerant setting. Besides, the framework is most beneficial in this setting, since larger programs lead to larger improvements over non-symbolic approaches. Nevertheless, note that this approach may be employed using any cost function. For example, one may want to select accuracy parameters while aiming to minimize the number of CNOT or CZ gates when targeting NISQ devices.

9.5.1 Cost/constraint functions: extraction

The methodology proceeds by automatically generating two pieces of code that compute (an upper bound on) (1) the number of costly quantum gates (T) and (2) the overall approximation error as a function of the different approximation errors (E):

$$T(\varepsilon_1, \ldots, \varepsilon_n) \qquad \text{and} \qquad E(\varepsilon_1, \ldots, \varepsilon_n).$$

The automatic generation of these two functions can be achieved via a few simple transformations of the program's Abstract Syntax Tree (AST). Specifically, to generate T, all calls to native operations are removed from the AST, except those corresponding to costly gates that are replaced by counter-increments. The program computing a bound on the overall approximation error (E) can be generated in a similar fashion, where increments are added for every epsilon declaration (see Sec. 9.2).

Example 9.5.1 *Fig. 9.3 shows the different decomposition levels of the QPE algorithm. The standard coherent QPE requires $2^{n_{QPE}} - 1$ controlled time-evolution unitaries ^cU, followed by an inverse QFT. At the next decomposition level, each unitary (including the QFT) is decomposed into rotation gates. In turn, those rotations will be fed into rotation synthesis, which outputs a sequence of $O(\log \frac{1}{\varepsilon_R})$ Clifford+T gates for each rotation, where ε_R denotes the target accuracy of rotation synthesis (per rotation). Since errors accumulate at most linearly due to being unitary (see Sec. 9.2), an upper bound on the overall approximation error can be computed by adding all the ε_i introduced by the various decomposition steps.*

9.5.2 Cost/constraint functions: optimization

Once the two pieces of code evaluating the total approximation error $E(\varepsilon_1,...,\varepsilon_n)$ and the cost $T(\varepsilon_1,...,\varepsilon_n)$ have been generated, they could be fed into the simulated annealing procedure. While this would allow us to perform accuracy management *automatically*, the resulting code would take substantial time to execute: typical quantum applications require on the order of 10^{15} or more operations [111] and the optimization loop is executed hundreds of times until suitable accuracy parameters are found.

As a remedy, custom compiler optimization passes are employed to significantly reduce the time required to evaluate gate counts and error bounds. Specifically, the methodology aims to infer symbolic and loop-free expressions for (upper bounds on) gate count and overall approximation error. The following example demonstrates how beneficial the use of a symbolic approach is.

Example 9.5.2 *Consider the example of the approximate quantum phase estimation algorithm and a two-mode simulated annealing procedure. Even with an optimized annealing schedule, it will require a minimum of about 200 evaluations to guarantee an overall approximation error of at most* 10^{-2}*. As accuracy parameters* $\varepsilon_1 \ldots \varepsilon_n$ *approach the optimal values (minimizing the* T*-count), one evaluation of the non-optimized* $T(\varepsilon_1,...,\varepsilon_n)$ *function on 8 qubits takes* $9m\,10s$*, while evaluating the inferred symbolic expression takes* $0.1\mu s$*. If the number of qubits grows to 16, then it takes* $34m\,14s$ *for the non-optimized case, while evaluating the symbolic expression still takes* $0.1\mu s$*.*

The transformations proposed at the level of the intermediate-representation of the compiler are shown in the following table:

Table 9.1 – Compiler transformations used to optimize the cost and constraint functions.

	Original Code	Symbolic expression
1	**for** $i \leftarrow 0$ to N **do** $v \mathrel{+}= const$	$v \mathrel{+}= const \cdot N$
2	**for** $i \leftarrow 0$ to N **do** $v \mathrel{+}= f(i)$	$v \mathrel{+}= \sum_i f(i)$
	for $i \leftarrow 0$ to N **do** $v \mathrel{+}= \min(g(i), h(i))$	$v \mathrel{+}= \sum_i \min(g(i), h(i))$ $\leq$ $v \mathrel{+}= \min(\sum_i g(i), \sum_i h(i))$
	for $i \leftarrow 0$ to N **do** $v \mathrel{+}= i^p$	$v \mathrel{+}= \sum_i i^p$ $(p+1)^{\text{th}}$ degree polynomial derived from Faulhaber's formula

3	**if** $(\ldots)$ **then** $expr1$
	else $expr2$
		$\max(expr1, expr2)$

In particular, the optimization routine would:

1. Check if there is an addition between a variable v initialized outside the loop and a loop invariant, and if v is not used elsewhere in the loop. If so, apply transformation 1, where N is the number of iterations of the loop.

2. Check if there is an addition between a variable v initialized outside the loop and a function f only depending on the inductive variable and other loop invariants. If so, apply transformation 2. In the particular case where $f(i)$ is min(), the expression can be upper bounded as shown in Table 9.1. In addition, polynomial expressions can be derived from some finite series using Faulhaber's formula [218].

3. Transform generic branching instructions into $\max(if, else)$ instructions, where the branch that gives the largest contribution to the cost function is selected.

The following example shows the described transformations applied to the AQFT quantum algorithm.

Example 9.5.3 *The pseudo-code of the function $AQFT\_T$, which computes the total number of T gates required for the AQFT algorithm, obtained after source-to-source transformation is shown in Alg. 9.1. As the implementation uses three rotation gates for each controlled-rotation, the function takes as input three accuracy parameters. This is why there are three innermost loops in Alg. 9.1. Since the objective is to extract a symbolic expression for the variable T_{count}, as many loops as possible must be flattened. The if statement in the inner-loop may be hoisted, resulting in the following expression:*

```
for $j \leftarrow 0$ to $\min(n-1-i, l)$ do
```

Then all the loops are optimized by applying the transformations in Table 9.1. Finally, the code in Alg. 9.2 is obtained, which shows the closed-form expression for the T_{count} with respect to the algorithm's parameters.

While the methodology succeeds at extracting closed-form expressions for all the proposed examples, note that this is not necessary for the methodology to work: the remaining control flow would not affect applicability or correctness, but merely cause an increase in runtime. Indeed, there are cases in which some residual code remains in the resulting expressions. This happens, for example, when some of the program parameters are read from a file. Consider the program performing phase estimation of the time evolution of a TFIM Hamiltonian described in Section 9.3, but where the parameters J, h and n are read from a file. The program may check whether the input Hamiltonian is valid, e.g., whether $n \geq 0$, before instantiating the circuit. As a consequence, such an if/else statement would remain in the final expression.

Algorithm 9.1 Cost function for the AQFT algorithm

```
function N_ROT($\varepsilon_{rot}$) return $1.5 * \log_2(1./\varepsilon_{rot})$
function AQFT_T($\varepsilon_{QFT}, \varepsilon_{R_1}, \varepsilon_{R_2}, \varepsilon_{R_3}$)
    $T_{count} \leftarrow 0$
    $l \leftarrow \lceil \log_2(n/\varepsilon_{QFT}) \rceil + 3$
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $n-1-i$ do
            if $j \leq l$ then
                for $k \leftarrow 0$ to $N\_ROT(\varepsilon_{R_1})$ do $T_{count}$++
                for $k \leftarrow 0$ to $N\_ROT(\varepsilon_{R_2})$ do $T_{count}$++
                for $k \leftarrow 0$ to $N\_ROT(\varepsilon_{R_3})$ do $T_{count}$++
    return $T_{count}$
```

Algorithm 9.2 Cost function for the AQFT algorithm after loop optimization

```
function AQFT_T($\varepsilon_{QFT}, \varepsilon_{R_1}, \varepsilon_{R_2}, \varepsilon_{R_3}$)
    $T_{count} \leftarrow 0$
    $l \leftarrow \lceil \log_2(n/\varepsilon_{QFT}) \rceil + 3$
    $T_{count} = T_{count} + \min(\frac{n(n-1)}{2}, nl) \cdot (N\_ROT(\varepsilon_{R_1}) + N\_ROT(\varepsilon_{R_2}) + N\_ROT(\varepsilon_{R_3}))$
    return $T_{count}$
```

9.6 Compiler and language requirements

To add this methodology for automatic accuracy management to any quantum programming language, a few features must be added to the compiler if not already supported. Here I list all the features that the compiler must support to bring such an integration into fruition.

I have identified such features by working on the prototype developed using the LLVM project, which was chosen for its modular infrastructure and libraries. The identified integration strategy can be applied to any other quantum programming language. This is demonstrated by the second prototype in Q#. Furthermore, the LLVM prototype shows that support for the framework may also be added to quantum programming languages that are embedded in a classical host language.

Don't-cares. The methodology requires that the compiler identifies subroutine parameters that do not or only negligibly affect the total error and the cost function. Such function parameters are called *don't-cares.* The corresponding arguments will be replaced by a default value in all calls to that subroutine. This allows the compiler to optimize repeated calls to the same function.

Example 9.6.1 *Consider AQFT, which includes many calls to the rotation gate with different rotation angles. Normally, these calls will have to be evaluated several times by the compiler, even if the angle will have no impact on the approximation error selected to decompose the rotation. This problem is addressed by annotating the angle parameter as a don't-care. This will* *result in many calls to the same function with identical arguments that hence may be removed by the compiler.*

Epsilon declarations. To provide language support (see Section 9.4) the compiler must be capable of locating all introduced accuracy parameters. This can be done by matching against a specific class or by using annotations.

Abstract Syntax Tree transformation. The compiler must provide access to the AST (or to a reasonably high-level IR) and allow rewriting and copying. In particular, it needs to generate 3 different versions of the entire program: one that computes the total approximation error, one that computes the total cost, and the original quantum program, which will ultimately be invoked using optimized accuracy parameters.

The programs obtained after the described AST modifications could already be used to estimate the resource requirements of the quantum program. In this case, the only advantage with respect to using state-of-the-art methods would be that the approach provides language support by keeping track of all the introduced approximation errors, work that otherwise would have to be performed manually. In addition, the estimation would be too slow to be used in an optimization procedure (see results in Fig. 9.6). A remedy is to introduce specific rewrites that reduce the time to evaluate these expressions (see next paragraph).

Rewrites to make evaluation more efficient. In order to speed up the optimization process, fast evaluations of the cost and error functions are needed. The two functions extracted from most quantum algorithms will feature many loops performing simple counter increments or floating-point additions. The goal of extracting a symbolic expression from the control flow structure is achieved implementing and employing compiler optimization, see Table 9.1.

9.6.1 LLVM prototype

Having described the features that are necessary to equip a programming language with automatic accuracy management, I now provide implementation details for the LLVM prototype.

Don't cares. The LLVM implementation uses compiler annotations to introduce additional information in the source code. In particular, the user attaches a *don't care* annotation to a parameter declaration if it has negligible effect on the cost/constraint functions.

Epsilon declarations. Epsilon declarations are matched with a custom type. The prototype framework provides a macro that allows the programmer to quickly define new types of error. This can be seen in the source code for the approximate QFT in Fig. 9.4, where `epsilon_QFT`

```
1  #include <qadf/epsilons.hpp>
2  #include <qadf/operations.hpp>
3
4  REGISTER_EPSILON(epsilon_QFT)
5
6  inline void Q_FUNC AQFT(int qubits[], const int n)
7  {
8    epsilon_QFT eps_QFT;
9    int lim = ceil(log2(n / eps_QFT))+3;
10   for(int i = 0; i < n; i++)
11   {
12     H(qubits[i]);
13     for (int j = 0; j < min(n-1-i, lim); j++)
14     {
15       CR(qubits[j + i + 1], M_PI/(1 << (j + 1)), qubits[i]);
16     }
17   }
18 }
```

Figure 9.4 – C++ code for the approximate QFT as consumed by the LLVM prototype.

is registered as a new type of error in line 4 and then used in lines 8–9.

Abstract Syntax Tree transformation. Source-to-source transformation is used to modify the AST. I implemented a *ClangTool* and run an *ASTFrontendAction*: a routine that has access to the AST and allows us to interface with the source code. The *ClangTool* outputs files containing the function to compute the T-count, e.g., the one in Alg. 9.1, and the function computing the total approximation error E. The action exploits the *ASTMatcher* library, which allows us to match nodes in the AST that have some specific properties. The library provides a concise way of describing patterns and is implemented as a *domain-specific language* (DSL). In addition, matchers allow us to access the source code by running a callback function on the matched AST nodes. Once the locations of interest in the code have been identified, one can use an instance of the *Rewriter* class to modify the code accordingly. As the AST is constant by design, a new file containing the transformed source code is generated.

The tool also makes use of header files defining basic quantum operations, such as the ones in the Clifford+T gate set. These header files can be adjusted according to the specific application. For example, in addition to the T gate, one might want to consider other expensive operations, e.g., two-qubit gates for NISQ devices.

The result of running the Clang tool is a new source file computing the total error or gate count as a function of all the accuracy parameters defined in the source code.

Rewrites to make evaluation more efficient. The compiler optimization passes eliminate loops in the expressions for the cost (or the total error) by replacing them with additions and multiplications. For example, the first loop optimization described in Table 9.1 is supported for both integral and floating-point numbers. A custom LLVM *loop pass* is used to perform this optimization.

Example 9.6.2 *Consider the following code that computes the total approximation error of n quantum operations, characterized by the same approximation error* `val`*:*

```
double Eps = 0.00;
double val = 0.02;
for (int i = 0; i < n; i++) {
  Eps += val;
}
```

Once this function is compiled into Intermediate Representation (IR) code, `val` *will be identified as a loop-invariant variable, while* `Eps` *will be assigned to a so-called PHI node. PHI nodes assign a variable with a different value, depending on the predecessor of the current block, where blocks are groups of instructions. In the example, the PHI node would have two incoming values:* 0.00 *(if the predecessor block is outside the loop) and the temporary value containing the addition result (if the predecessor was the previous loop iteration). PHI nodes are defined in the loop header block.*

The loop pass traverses the code from the innermost to the outermost loop in the IR and checks whether:

1. it contains an instruction performing the addition operation between a loop-invariant operand and a variable defined through a PHI node in the loop header,
2. the PHI node is only used in the addition operation inside the loop or in the loop latch block,
3. the result of the addition is only used as the incoming value of the PHI node.

If the described conditions apply, the loop is removed. Referring to the code in Example 9.6.2, the operations $\%1 = val * n$ and $\%2 = Eps + \%1$ would be added to the pre-header loop block, i.e., outside the loop. In addition, the result %2 would replace all uses of the original addition operation, which can then be erased. All the other transformations in Table 1 are implemented in a similar fashion.

Extraction of symbolic representation. The LLVM prototype also has a method that navigates the fully optimized IR code and extracts symbolic expressions for the two estimates.

The pass to extract the symbolic expression from the IR is implemented as an LLVM *function pass.* Given the main function, it starts from the return instruction and recursively visits all instruction's operands annotating the respective functionality. The recursion terminates when the operand is a constant or, in general, not an instruction.

```
namespace Accuracy {

  open Microsoft.Quantum.Arrays;
  open Microsoft.Quantum.Convert;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Math;

  // intrinsic function for epsilon declaration
  function EpsilonValue() : Double {
    body intrinsic;
  }

  operation AQFT(qs : Qubit[]) : Unit is Adj+Ctl {
    let nQubits = Length(qs);
    let eps_QFT = EpsilonValue(); // epsilon-declaration
    let lim = Ceiling(Lg(IntAsDouble(nQubits) / eps_QFT)) + 3;
    for ((i, q) in Enumerated(qs)) {
      H(q);
      for (j in 0 .. MinI(nQubits - i - 1, lim)) {
          (Controlled R1Frac)([q], (1, j + 1, qs[j + i + 1]));
      }
    }
  }
}
```

Figure 9.5 – Q# implementation of the approximate quantum Fourier transform with epsilon declarations.

The extraction pass supports the following instructions: casting, PHI nodes, selects, truncations, zero extensions, call instructions, compare instructions, shifts, addition, multiplication, division, and subtraction. The expression is written in the Wolfram language, such that Mathematica [219] can be used for conversion to LaTeX and further expression simplifications.

9.6.2 Q# integration

Q# is a standalone quantum programming language developed by Microsoft to facilitates the description of hybrid quantum-classical programs. Fig. 9.5 shows a snippet of Q# code that implements the approximate quantum Fourier transform operation, as described in Section 9.3.

Qubits are represented in Q# using the type `Qubit` and they are treated as opaque items that can be passed to both functions and operations, but that can only be interacted with by passing them to intrinsic (built-in) operations. Q# also uses namespaces to group definitions together, and elements from other namespaces may be referenced. Q# distinguishes functions from operations. Functions are pure and free of side effects, whereas operations can have side effects, such as the application of an intrinsic operation to a qubit or register. Q# can perform type-safe symbolic computations to automatically derive the adjoint (inverse) and the controlled variants of an operation, enforced by providing the `is Adj+Ctl` declaration in line 13 of Fig. 9.5. Line 15 shows a declaration of an approximation parameter, i.e., `eps_QFT`, which is possible by adding language support for approximation errors. The remainder of this section provides details on how this and other features have been implemented in the Q# compiler.

Don't cares. Currently, the Q# prototype does not support *don't cares*. Operations such as rotation gates—in which *don't cares* can help to declare that the rotation angle affects the gate cost only negligibly—are implemented as intrinsic operations that introduce an epsilon variable and increment the gate counter explicitly by a value depending on the accuracy parameter. For more general cases, Q# should support parameter-level annotations to declare *don't cares* explicitly.

Epsilon declarations. An intrinsic function is used for epsilon declarations. Being intrinsic, it does not require an implementation inside the Q# program to be used, but it can be located inside the AST by the transformation passes. In the AQFT example, the function is declared in lines 9-11 and allows the programmer to declare and use accuracy parameters, e.g., as done in lines 15-16.

Abstract Syntax Tree transformation. An AST transformation pass detects all `EpsilonValue` declarations, removes them, and adds them as arguments to the operation signature. This step is performed before producing the two pieces of code that compute the number of costly quantum gates and the upper bound on the overall approximation error. Note that the intrinsic `EpsilonValue` function is never called in the resulting programs.

There exist no global variables and no call-by-reference parameters in Q#. However, it is possible to declare an intrinsic operation in Q# and implement it in C#. In order to count the number of T gates, the operation `IncrementCounter(id, value)` is introduced, whose implementation in C# increments a global counter, called `id`, by `value`. A similar technique is used to accumulate the bound on the overall approximation error. Each declaration of an epsilon value is replaced by a call to an operation `IncrementValue(id, value)` whose implementation in C# increments a global variable called `id` by `value`.

Rewrites to make evaluation more efficient. The Q# compiler already contains some transformation passes, e.g., for operation inlining, propagating constants, or removing unused code. Two additional transformation passes need to be added to optimize the use of `IncrementCounter` and `IncrementValue` calls. Without loss of generality the transformation passes are explained by means of the `IncrementCounter` operation, remarking that they work analogously for the `IncrementValue` operation.

The first transformation pass collects all `IncrementCounter` calls inside a scope level that have the same `id` and do not contain values incorporating mutable variables. These calls can be merged into a single call by accumulating all values, benefiting from further optimization, e.g., constant propagation. The second transformation pass lifts `IncrementCounter` calls inside a for-loop. If the call is the only statement in the body of the for-loop and does not use the loop variable to compute the value, the for-loop can be removed when multiplying the value in the `IncrementCounter` call by the number of loop iterations.

9.7 Qualitative evaluation

This section evaluates the prototypes using different quantum algorithms. The first example is a simple quantum Fourier transform. As a highlight, the LLVM prototype automatically extracts a symbolic expression for the phase estimation of a Trotter-decomposed time-evolution under a transverse-field Ising model Hamiltonian, where the phase estimation features an approximate QFT. The C++ code for the approximate QFT can be found in Fig. 9.4 and the corresponding Q# code is shown in Fig. 9.5. In addition, the prototype is tested on Shor's algorithm [18], to provide the reader with a large-scale example. The period finding quantum routine as implemented in the open-source programming framework ProjectQ [61] is used. All the obtained expressions for the T-count and the total approximation error are reported in Table 9.2.

Exact QFT The first example is an implementation of the exact quantum Fourier transform. The prototype is able to directly optimize all loops, including the outermost loop which yields a sum of the form $\sum_i^{n-1} i = \frac{n(n-1)}{2}$. It finds the correct closed-form expression for the total error and it correctly identifies the number of T gates to be $O(n^2 \log(1/\varepsilon_R))$. Table 9.2 shows the detailed output.

Approximate QFT (AQFT) Next, consider the approximate quantum Fourier transform. The C++ source code that serves as the input to the prototype is depicted in Fig. 9.4. The prototype upper bounds an intermediate expression of the form $c + \sum_i \min(f(i), g(i))$ by choosing one of the arguments to the min-function and successfully derives a closed-form expression for both the T gate count and the total approximation error, see Table 9.2.

Quantum phase estimation (QPE) The example combines the time evolution of a TFIM with QPE, to find the ground state of the TFIM. This first QPE example makes the simplifying assumption that the inverse QFT can be performed natively. As can be seen in Table 9.2 (labeled *QPE simplified*), the methodology is capable of removing all loops and outputs two closed-form expressions for the T-count and the total error.

The next step is to drop the simplifying assumption that the inverse QFT can be performed natively. The inverse QFT is implemented as discussed in Section 9.3.1. The closed-form expressions obtained by the prototype for this case can be found in Table 9.2 labeled *QPE with*

QFT.

Finally, the exact QFT is replaced by an approximate QFT (see Fig. 9.4 for the C++ code). As in the AQFT example, the optimization pass upper bounds an intermediate expression of the form $c+\sum_i \min(f(i), g(i))$ by choosing one of the arguments to the min-function. Table 9.2 shows the detailed output, which consists of two fully-symbolic expressions for the T-count and for the approximation error.

Shor's algorithm The last example is Beauregard's implementation of Shor's algorithm [216], which defines two approximation errors, one for the rotation gates ε_R and one for the approximate QFT (ε_{QFT}). In this implementation, each controlled unitary in the phase-estimation procedure is a modular multiplication (by a constant). Please note that the QPE example performs a phase-estimation on the time-evolution operator that evolves the system according to the transverse-field Ising model Hamiltonian. Therefore, while both examples make use of phase estimation, their cost and error functions are vastly different because the phase estimation is performed on different unitaries.

In summary, the proposed methodology successfully produces closed-form expressions for the total error and gate count for all the examples. The next section shows that access to symbolic expressions enables a significant reduction in the time required to optimize accuracy parameters.

9.8 Quantitative evaluation

This section demonstrates how the LLVM prototype enables faster evaluations of the cost and constraint functions by leveraging the transformations in Table 9.1. The runtimes of the symbolic resource estimation method are compared against a *non-symbolic* approach. The latter does not use symbolic estimations for resources and errors. The non-symbolic estimates are generated by the methodology during the AST transformation step (see Section 9.6).

All experiments were run on a MacBook Pro with an Intel Core i5 processor with 3.1 GHz processor clock frequency and 16 GB main memory. All source codes have been compiled using Clang version 9.0.0 with level 3 optimization (*-O3*) and with the fast-math mode enabled (*-ffast-math*).

As previously described, the prototype uses a two-mode annealing procedure to find suitable assignments for all accuracy parameters. I measure the runtime for performing one evaluation of the T and E functions using accuracy parameters provided by the annealer, guaranteeing an upper bound on the overall approximation equal to $5 \cdot 10^{-3}$. The runtime measurements are performed using different input sizes for Shor's algorithm and for the QPE example (with approximate QFT). The plots in Fig. 9.6 depict the sum of the runtimes required for evaluating

Table 9.2 – Symbolic expressions for the T-count (T) and total approximation error (E) as extracted by the LLVM prototype from the source code of various example programs.

Algorithm (selected function)	Symbolic expression
QFT	$E = \frac{3}{2}\varepsilon_R n(n-1)$ $T \simeq 3.246n(n-1)\log\left(\frac{1}{\varepsilon_R}\right)$
AQFT	$E \simeq 3\varepsilon_R n\left(\left\lceil\frac{\log(\frac{n}{\varepsilon_{QFT}})}{\log 2}\right\rceil + 3\right) + \varepsilon_{QFT}$ $T \simeq 6.492n\log\left(\frac{1}{\varepsilon_R}\right)\left(\left\lceil\frac{\log(\frac{n}{\varepsilon_{QFT}})}{\log 2}\right\rceil + 3\right)$
QPE simplified	$E \simeq \frac{\left(\varepsilon_R(n-\frac{1}{2})+\frac{\varepsilon_{TE}^{3/2}}{4}\right)\left(2^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+4}-4\right)}{\sqrt{\varepsilon_{TE}}} + \varepsilon_{QPE}$ $T \simeq \frac{34.625(n-\frac{1}{2})\log\left(\frac{1}{\varepsilon_R}\right)(2^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil-\frac{1}{4}})}{\sqrt{\varepsilon_{TE}}}$
QPE with QFT	$E \simeq \frac{16\left(\varepsilon_R(n-\frac{1}{2})+\frac{\varepsilon_{TE}^{3/2}}{4}\right)\left(2^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil}-\frac{1}{4}\right)}{\sqrt{\varepsilon_{TE}}} + \frac{3}{2}\varepsilon_R(\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+1)(\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+2)+\varepsilon_{QPE}$ $T \simeq \log\left(\frac{1}{\varepsilon_R}\right)\left(\frac{(n-\frac{1}{2})(34.625^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil}-8.656)}{\sqrt{\varepsilon_{TE}}} + \lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil(3.246\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+9.738)+6.492\right)$
QPE with AQFT	$E \simeq 3\varepsilon_R(\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+2\left(\left\lceil\frac{\log(\frac{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+2}{\varepsilon_{QFT}})}{\log 2}\right\rceil+3\right) + \frac{16\left(\varepsilon_R(n-\frac{1}{2})+\frac{\varepsilon_{TE}^{3/2}}{4}\right)\left(2^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil}-\frac{1}{4}\right)}{\sqrt{\varepsilon_{TE}}} + \varepsilon_{QFT}+\varepsilon_{QPE}$ $T \simeq \log\left(\frac{1}{\varepsilon_R}\right)\left(6.492(\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+2)\left(\left\lceil\frac{\log\left(\frac{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil+2}{\varepsilon_{QFT}}\right)}{\log 2}\right\rceil+3\right) + \frac{34.625(n-\frac{1}{2})\left(2^{\lceil 2.652-1.443\log(\varepsilon_{QPE})\rceil}-\frac{1}{4}\right)}{\sqrt{\varepsilon_{TE}}}\right)$
Shor's algorithm [216]	$E \simeq 4n^2\left(\varepsilon_R(n+1)\left(30\left\lceil\frac{\log\left(\frac{n+1}{\varepsilon_{QFT}}\right)}{\log 2}\right\rceil+109\right)+10\varepsilon_{QFT}\right)$ $T \simeq 2n^2\left(129.843(n+1)\log\left(\frac{1}{\varepsilon_R}\right)\left\lceil\frac{\log\left(\frac{n+1}{\varepsilon_{QFT}}\right)}{\log 2}\right\rceil+471.761(n+1)\log\left(\frac{1}{\varepsilon_R}\right)+84n+91\right)$

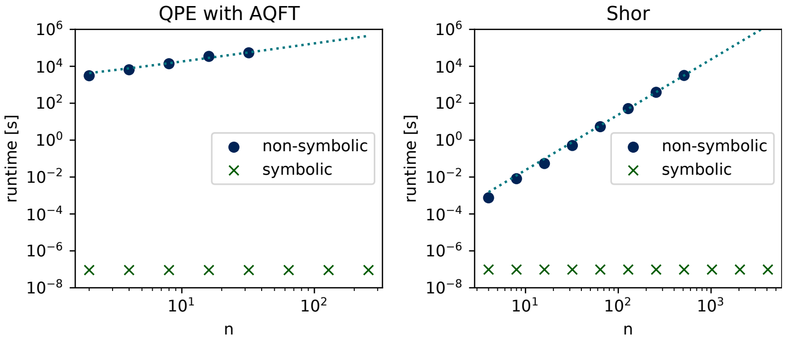


Figure 9.6 – A comparison between the runtimes of the non-symbolic approach and the symbolic approach developed in this thesis is shown. Each data point marks the runtime required for a single evaluation of the T function plus a single evaluation of the E function for the QPE algorithm with approximate QFT and for Shor's algorithm. Optimized accuracy parameters are used as input in order to assert that the total approximation error is at most $5 \cdot 10^{-3}$. I provide polynomial extrapolations from the collected runtimes for the non-symbolic approach as the runtime tops out for bit sizes above 512 for Shor and 32 for QPE.

the constraint and cost functions once, as they are always evaluated the same number of times in the simulated annealing procedure. While the non-symbolic approach exhibits a growing runtime as a function of the problem size, the symbolic method shows the expected constant behavior. In particular, the runtime of the non-symbolic approach grows linearly for the QPE example and (roughly) cubically for the implementation of Shor's algorithm.

Given that both examples are valid applications of quantum computers only for large problem sizes (e.g., the target number of bits for Shor's algorithm is $n \approx 4000$), Fig. 9.6 shows function extrapolations for the non-symbolic approach. The two resulting functions are $f_{QPE} = 1737.30x + 816.98$ and $f_{Shor} = 2.38 \cdot 10^{-5} x^3$.

To estimate the time it would take to optimize accuracy parameters using a *non-symbolic* approach, the runtime results in Fig. 9.6 are multiplied by a lower bound on the number of function evaluations. A loose lower bound of 100 evaluations (of each function) is estimated. To determine suitable accuracy parameters for Shor's algorithm, it would theoretically take approximately 1890 days for $n = 4096$ bits and an overall approximation error of at most $5 \cdot 10^{-3}$.

Therefore, it is possible to conclude that non-symbolic approaches are not suitable for large-scale applications.

9.9 Summary

In this chapter, I described the first framework with the ability to automatically manage approximation errors and outputting (near-)symbolic resource estimates. The methodology targets approximation errors which are introduced during compilation, examples are given in Section 9.2. The algorithms used in the chapter as running examples are described in Section 9.3.

Sections 9.4 described how integrating language support for expressing approximation errors in quantum programs facilitates the task of programming a quantum computer in an approximation-aware fashion. Then, Section 9.5 explained how it is possible to automate the process of optimizing accuracy parameters at the compiler level. It is possible to integrate the proposed methodology into any quantum programming language, if its compiler supports the features discussed in Section 9.6. The latter section also discusses the two proposed prototype implementations in LLVM and Q#. In Section 9.7, I test the ability of the LLVM prototype to extract symbolic expressions for the T-count and the total approximation error for the sample quantum algorithms described in Section 9.3. The demonstration that a symbolic resource estimation is the only applicable strategy to solve the optimization problem for large-scale examples, is given in Section 9.8.

The currently proposed technique would require additional inputs to handle branching on measurement and repeat-until-success-like structures [213]. Future work could focus on improving the handling of similar structures. Future work could also compare the accuracy upper bounds estimated by the proposed method to the actually achieved errors on example applications, as the proposed methodology provides a pessimistic estimate.

Open-source development Part IV

10 Caterpillar

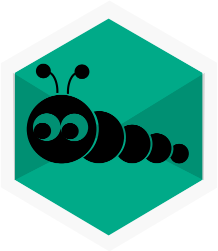


Figure 10.1 – *Caterpillar*'s logo

All the algorithms and the techniques for the compilation of quantum circuits in this research work have been implemented as part of open-source projects. At the beginning of my doctoral studies, I worked on developing the open-source project RevKit [194], a toolkit for reversible circuit design written in C++. RevKit provided core functionality and elaborated methods for synthesis, optimization, and verification of reversible (and quantum) circuits. The experiments in Section 7.4 and the techniques in Chapter 8 were all implemented in RevKit.

More recently, together with other members of the Integrated Systems Laboratory (LSI) at the École Polytechnique Fédérale de Lausanne (EPFL), we started a collective project: the EPFL logic synthesis libraries. Among these, I personally develop and maintain the library *caterpillar*, which is dedicated to the compilation of quantum oracles and to quantum memory management. The purpose of this open-source effort is to let other researchers use our developed methods to enable advancement in the field.

The modular open-source C++- 14 and C++-17 libraries developed at LSI provide efficient implementations of common logic synthesis tasks. A detailed description of each library and some show-cases can be found in [181]. In addition, the GitHub repository of each library contains detailed and specific documentation. The page https://github.com/lsils/lstools-showcase contains links to all the repositories. Each library targets one general aspect of logic synthesis:

- ***alice*** eases the implementation of user interfaces and their integration in scripting languages;
- ***mockturtle*** provides generic synthesis and optimization algorithms for logic networks;
- ***lorina*** parses logic and networks in various representation formats;
- ***kitty*** provides an effective way for explicit representation and manipulation of Boolean functions;
- ***bill*** serves as an integration layer for symbolic reasoning engines;
- ***percy*** synthesizes optimum logic networks;
- ***easy*** represents and synthesizes Exclusive Sum-Of-Products (ESOP) forms;
- ***tweedledum*** is a generic quantum compilation library with synthesis, optimization, and mapping algorithms;
- ***caterpillar*** is a quantum circuit compilation library for fault-tolerant quantum computing.

The libraries are modular, in the sense that many may be required for one specific research project. For example, *mockturtle* includes *kitty* to represent the Boolean functions computed by the nodes of k-LUT networks and *caterpillar* includes *mockturtle* to provide the initial multi-level network representation required by its compilation algorithms.

10.1 Compiling with caterpillar

Caterpillar, whose logo is shown in Fig 10.1, is an open-source header-only C++ library dedicated to the compilation of large quantum circuits. The code and the complete documentation can be found at https://github.com/gmeuli/caterpillar.

Caterpillar includes the library *mockturtle* to make use of its logic network data structures and manipulation algorithms. Besides, quantum circuits are represented in *caterpillar* using the EPFL library *tweedledum*, which contains all the abstractions and the data structures to efficiently represent quantum circuits.

The pivot compilation method, called `logic_network_synthesis`, performs the hierarchical reversible synthesis described in Chapter 5 to synthesize a quantum circuit from a multi-level logic network. The synthesis method can take all the networks implemented in *mockturtle* as input: AIGs, XAGs, XMGs, MIGs and LUT networks. The following code snippet shows the interface of the function template `logic_network_synthesis`.

```
// header: /caterpillar/synthesis/lhrs.hpp
template<class QuantumNetwork, class LogicNetwork, class
  SingleTargetGateSynthesisFn = tweedledum::stg_from_pprm>

bool logic_network_synthesis( QuantumNetwork& qnet,
                              LogicNetwork const& ntk,
                              mapping_strategy<LogicNetwork>& strategy,
                              SingleTargetGateSynthesisFn const& stg_fn = {},
                              logic_network_synthesis_params const& ps = {},
                              logic_network_synthesis_stats* pst = nullptr );
```

The function takes as template parameters the quantum network, expressed by the `netlist` template class in *tweedledum* and a logic network, that is one of the logic networks included in *mockturtle*.

The function also requires to specify a member of a class derived from the base class `mapping_strategy`, as third parameter. By specifying a mapping strategy one can control how the multi-level logic network is translated into a reversible circuit, while guaranteeing uncomputation of all intermediate results, i.e., a garbage-free circuit. As explained in Section 5.3, several strategies are possible, each one with a different impact on the number of helper qubits of the final quantum circuit.

The fourth parameter is a method to decompose each single-target gate into Clifford+T circuits. This method is only applied when the compilation is performed with a k-LUT network as input. In this case, each single target gate is controlled by a Boolean function and needs to be decomposed into quantum instructions. The default decomposition method is the ESOP decomposition based on Pseudo-Kronecker Reed Muller (PKRM) expressions [192], provided by *tweedledum*. The *alpha* version of *tweedledum*, currently integrated into *caterpillar*, also includes the `stg_from_exact_esop` method, which implements the exact synthesis method proposed in 7.2.2. When the compilation algorithm is applied to logic networks characterized by nodes computing functions with known quantum implementations, e.g., XAGs, such decomposition methods are not used.

The last two parameters, respectively `logic_network_synthesis_params` and `logic_network_synthesis_stats` are needed to provide specific settings to the procedure and to access runtime information on a specific compilation process.

10.1.1 Mapping strategies

The hierarchical synthesis method in *caterpillar* requires to specify the `mapping_strategy` to be used to map the logic of each node of the input logic network into the available helper qubits. In general, such strategies are also required in quantum compilation to place pre-optimized quantum circuits blocks into a circuit, whenever the data dependency is expressed using a Directed Acyclic Graph (DAG).

Every mapping strategy in caterpillar is derived by the base `mapping_strategy` template class.

```
//header: caterpillar/synthesis/strategies/mapping_strategy.hpp

template<class LogicNetwork>
class mapping_strategy {
public:
  using step_function_t = std::function<void( mockturtle::node<LogicNetwork> const&, mapping_strategy_action const& )>;
  using step_vec_t = std::vector<std::pair<mockturtle::node<LogicNetwork>, mapping_strategy_action>>;

  /* take the logic network as input and defines the strategy's steps sequence */
  virtual bool compute_steps( LogicNetwork const& ntk ) = 0;

  /* iterate through the strategy's steps applying the given function */
  void foreach_step( step_function_t const& fn ) const {
    for ( auto const& [n, a] : _steps ) {
      fn( n, a );
    }
  }

protected:
  step_vec_t& steps() { return _steps; }

private:
  step_vec_t _steps;
};
```

Every mapping strategy must then implement the public functions `foreach_step`, which iterates through the found strategy steps. A strategy is a vector of pairs with the network's node and the corresponding `mapping_strategy_action`. The supported actions are: `compute_action` and `uncompute_action` to compute (uncompute) the node functionality on a clean helper qubit; `compute_inplace_action` and `uncompute_inplace_action` to compute (uncompute) on top of an existing qubit; `compute_level_action` and `uncompute_level_action` to compute (uncompute) an entire level of the network in one single step.

Several mapping strategies are available in *caterpillar*:

- **Bennett strategy.** This strategy computes all the nodes in topological order and uncomputes them in inverse topological order. Every node logic is mapped on a clean helper qubit, i.e., out-of-place. Originally described in [170], it provides a solution that always returns, given an initial network, the smallest number of single-target gates and the highest number of helper qubits if compared to the other strategies.
- **Eager strategy.** This strategy makes sure of computing at first the outputs with the

largest transitive fan-in cone. Then, as soon as a primary output is computed, all nodes in the transitive fan-in cone are uncomputed if their value is not needed for an output that has not being computed yet. The present strategy achieves the same number of single-target gates as the Bennett strategy. As well as the Bennett startegy, the Eager strategy only supports out-of-place computation, which requires a clean helper qubit.

- **Best-fit strategy.** This strategy applies a k-LUT mapping that decomposes the network into cells. Cells are initially placed to form a reversible network following an eager strategy. The logic contained into each cell is decomposed by means of a second k-LUT mapping. The method selects the minimum k such that there are enough helper qubits to save intermediate results. A different "best-fit" k is selected for each cell. I described this method in Section 7.3.

- **Pebbling strategies.** The pebbling strategies are obtained by solving the reversible pebbling game on the given network. The problem is encoded as a SAT problem as shown in Section 5.3.

- **XAG strategies.** These strategies work specifically on XAGs. They correspond to the algorithms described in Chapter 6.

10.1.2 Pebbling strategies

The pebbling mapping strategies address the problem of building a reversible circuit of single-target gates by iteratively solving the SAT formulation of the reversible pebbling game played on the initial logic network. The pebbling strategies rely on a general interface to different SAT solvers. At the moment of writing this thesis, the ABC's[1] `bsat_solver` is integrated into *caterpillar*. In addition, the library works on top of a local installation of Z3 [148], if the corresponding CMake option is enabled: `USE_Z3`.

In addition to the standard reversible pebbling problem, *caterpillar* also enables to take into consideration weights associated with each node of a DAG. I have already explained in Section 6.3.3 how it is possible to modify the SAT encoding to optimize the total weight of the pebbling solution. The `weighted_pebbling_mapping_strategy` takes as template parameter the class of the input network`Network`. To be valid, the selected network class must provide the following method:

```
uint32_t get_weight (const Network::node n) const;
```

Some of the network data structures in *mockturtle* do not provide such a method. Luckily *mockturtle* itself provides the needed solution, which allows us to optimize the pebbling solution for any logic network: *views*. *Views* can add, modify or remove the methods implemented in a network interface. *Caterpillar* implements the `pebbling_view`, which adds a method to

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

set the weight of a node and one that returns such a weight. This view of the network can be passed to the weighted pebbling strategy.

A second variation of the reversible pebbling game is the in-place pebbling. This case considers that some network's node can be computed on top of one of the inputs. Take for example a node implementing the XOR operation, the corresponding CNOT gate can be computed on top of a qubit correspinding to one of the input signals, if this is not used by successive nodes. To being able to generate a pebbling solution which enables in-place operations, the encoding of the corresponding SAT problem requires a method to access all the parents of a node.

```
std::vector<node> get_parents( node const& n) const
```

Again, the `pebbling_view` adds such method into any *mockturtle* 's network interface.

10.1.3 Mapping strategies for XAGs

Some of the mapping strategies that are available in *caterpillar* are specific to the XAG data structure. The strategies `xag_mapping_strategy` and `xag_low_depth_mapping_strategy` implement Alg. 6.1 and Alg. 6.2, respectively. They both derive an abstract representation of the graph from the XAG, where each node corresponds to an AND node of the XAG and has an unlimited number of inputs. These correspond to the input signals of the parity functions, which were inputs to the AND node, as illustrated in Fig. 6.6. Alternatively, *caterpillar* provides strategies designed to perfom such algorithms directly on the abstract network implemented in *mockturtle*.

Besides, a mapping strategy specifically designed to pebble XAG graphs is also available: `xag_pebbling_mapping_strategy`. It takes an XAG as input and implicitly works on the abstract graph. The results obtained applying this strategy are available in Fig. 6.7. Alternatively, it is possible to use the weighted pebbling strategy on top of *mockturtle*'s `abstract_network`.

10.2 Show-case: resource estimation for large quantum circuits

This section illustrates one possible application of the *caterpillar* library. This example targets the problem of evaluating the resources required to perform a given large Boolean function. Such function is specified by an input file expressed in Verilog.

The following code can be used to print the resource estimation obtained using the compilation Algorithm 6.1. Note that I neglected some library namespaces to show a more compact code.

```
1  /* declare the XAG */
2  xag_network xag;
3
4  /* read XAG from benchmark using the library lorina */
5  auto const result = read_verilog( filename.v, verilog_reader( xag ) );
6  if ( result != return_code::success ) { return; }
7
8  /* set up strategy and synthesis parameters */
9  xag_mapping_strategy strategy;
10
11 /* start tracer */
12 xag_tracer_stats st;
13 xag_tracer(xag, strategy, {}, &st);
14
15 /* print statistics */
16 std::cout << "Quantum circuit stats: \n";
17 std::cout << "#CNOT = " << st.CNOT_count << "\n";
18 std::cout << "T-count = " << st.T_count << "\n";
19 std::cout << "T-depth = " << st.T_depth << "\n";
20 std::cout << "#qubits = "  << st.qubit_count << "\n";
```

Initially, the XAG object `xag`, is declared. Then, using the parser library *lorina* , and a Verilog reader included in *mockturtle* , the XAG represents the Boolean function defined by `filename.v`. If the parsing of the file is not successful, the execution is stopped. Please note that *lorina* also includes diagnostic methods.

Once the input XAG is defined, the mapping strategy is selected. The described case evaluates the resources obtained by synthesizing a quantum circuit using the XAG-based algorithm defined in Alg. 6.1. In line 15, the program uses the template function `xag_tracer` to collect the statistics of the obtained quantum circuit. The tracer is an alternative function to the `logic_network_synthesis` function. The difference is that the latter builds a quantum circuit: a netlist of quantum operations. For very large functions, as the ones synthesized in Chapter 6, this may require a prohibitive amount of memory. The tracer executes the synthesis algorithms specified by the mapping strategy while keeping track of the necessary quantum operation; hence without building and storing the quantum circuit.

10.3 Show-case: compilation with a fixed number of helper qubits

This second show-case considers the problem of mapping a logic function represented using an XAG (`xag` in the code) into a reversible network. Once synthesized, the reversible circuit will contain CNOT and Toffoli gates. The mapping is performed using a fixed number of helper qubits. In addition, it is possible to verify that the synthesized reversible network actually computes the desired Boolean function.

```
/* declare the reversible network */
netlist<stg_gates> rev;

/* set up strategy and synthesis parameters */
pebbling_mapping_strategy_params ps;
ps.pebble_limit = helper_qubits; // fix the number of helper qubits
ps.solver_timeout = timeout_ms; // timeout for the SAT solver [ms]
ps.conflict_limit = 0; // 0 means no conflict limits for the SAT solver
ps.optimize_weight = true;

xag_pebbling_mapping_strategy strategy {ps};

/* start logic network synthesis */
logic_network_synthesis_stats st;
logic_network_synthesis( rev, xag, strategy, {}, {}, &st );

/* check equivalence */
bool cec = check_equivalence_tt(xag, rev, st.i_indexes, st.o_indexes);
```

The example code starts with declaring the reversible network, which is a netlist of single-target gates. Then, it declares the selected mapping strategy: the *xag_pebbling_mapping_strategy*, solving the reversible pebbling game on the abstract graph derived from the XAG. This strategy enables to set many different pebbling parameters. In this example: the number of available pebbles is limited to the maximum allowed number of helper qubits; a timeout for the SAT solver is set up; no limit is given for the number of conflicts; weight optimization is enabled. Once the circuit is synthesized, the function `check_equivalence_tt`, verifies that the XAG and the reversible network have the same functionality. The last two arguments give the indices of the input and the output lines in the reversible circuit. This method internally transforms the reversible circuit into a multi-level logic network and then uses a generic simulation algorithm from *mockturtle* . This equivalence-checking method relies on truth tables and, as a consequence, does not provide a scalable solution. A more scalable approach is to use `check_equivalence_ntk`, which builds a miter of the logic network derived from the reversible circuit and the original one and then checks for satisfying assignments using SAT. A miter is a logic network derived from two other logic networks. As input it has the union of the two input sets and as output the XOR between the two original outputs. The function implemented by the miter is satisfied when the two original networks have different outputs.

10.4 Show-case: looking for the minimum pebbling number

For this last show-case, suppose there is no fixed limit on the number of available helper qubits for the mapping of the previous show-case. Still, it would be desirable to generate a quantum circuit with a reduced amount of qubits. It is possible to set up the pebbling strategy parameters to enable searching the smaller number of pebbles for which a solution can be found in a given time. The smallest number of pebbles required to pebble a given DAG is called *pebbling number*.

The following code snippet shows how to set the strategy parameters to perform this search.

```
/* set up strategy and synthesis parameters */
pebbling_mapping_strategy_params ps;
ps.pebble_limit = init_helper_qubits; // sets the initial #pebbles
ps.search_timeout = t1_s; // sets a timeout for the search
ps.solver_timeout = t2_ms;
ps.decrement_pebbles_on_success = true;
```

With the chosen settings, the algorithm starts looking for a solution with a number of pebbles equal to `init_helper_qubits`. If a solution is found, the SAT engine will be called again with a decremented number of pebbles in order to find a more constrained solution (enabled `decrement_pebbles_on_success`). This is repeated until either (i) the SAT solver returns *unsat*, (ii) the SAT solver hits the solver timeout or the conflict limit, (iii) the search hits its dedicated timeout.

10.5 Summary

This chapter introduced the modular C++ libraries for logic synthesis developed in my research group. In the context of this collective open-source effort, I personally develop and maintain the library *caterpillar*. This library is dedicated to the synthesis of quantum circuits starting from logic network specifications. The main synthesis method in *caterpillar* is described in Section 10.1. The method takes as template parameters different mapping strategies, including the one solving the reversible pebbling game. Section 10.1.2 provided more details on this latter strategy, while Section 10.1.3 focused on the strategies specifically designed to work with XAGs. Finally, three show-cases are given to illustrate the capabilities and the applicability of this library, in Sections 10.2, 10.3, 10.4.

The library is currently used by the following open-source projects: staq [2] and RevKit 3.0 [3].

[2] https://github.com/softwareQinc/staq

[3] https://github.com/msoeken/revkit

11 Conclusions and future directions

The research work that I presented in this thesis focuses on tackling the problem of translating a high-level description of a quantum algorithm into a low-level set of quantum instructions. This problem is generally referred to as *quantum compilation*. As for their reduced sizes, NISQ devices can efficiently rely on a manual or semi-manual compilation. The same is not true for larger-scale quantum computers that, according to many leading technology companies, are *soon* to be developed. Those will be characterized by thousands of qubits and will embed error-correcting techniques. They will be programmed at a high-level of abstraction and will rely on a complex software stack to enable their operation. Such quantum computers are expected to be capable of performing complex quantum algorithms, reaching computational capabilities far beyond what can be achieved with classical systems.

Many promising quantum algorithms require some classical logic function to be computed by the quantum systems. Quantum circuits implementing such Boolean functions are often called *oracles* and can require a large amount of resources. This is what motivates researching new and efficient quantum compilation techniques, capable of synthesizing quantum oracles while aiming at reducing their resource footprint. As this work targets fault-tolerant quantum computing and the Clifford+T library, the implementation cost functions considered are related to the T gates and the number of qubits.

This research project stands at the intersection between three research fields, described in the first chapters of this thesis: (1) classical logic synthesis, (2) quantum computing, and (3) reversible computing. Chapter 2 presents the field of logic synthesis, from which I borrowed many representations and methods, adapting them to the problem of quantum compilation. Chapter 3 introduces basic concepts of quantum computing, as the quantum circuit representation and the Clifford+T universal gate library. Finally, Chapter 4 introduces reversible gates and some known methods to decompose them into quantum gates. Following the background, the second part of this manuscript focuses on the synthesis and optimization of quantum circuits.

Chapter 5 introduces the k-LUT-based hierarchical synthesis to compile quantum circuits performing a Boolean function specified by a multi-level logic network. The method consists of several steps. The first step is the k-LUT mapping, which is used to decompose the original network into sub-networks with at most k inputs. I discussed how the choice of the k parameter, and more in general, the cost functions considered when performing this first decomposition step, can impact the final result. The second step is in charge of translating a possibly non-reversible Boolean function representation into a reversible circuit. The overall synthesis method owes the name *hierarchical* to this step. Indeed, the translation is done by traversing the graph and placing on the reversible circuit a single-target gate targeting a clean helper line for each node. The transformation leans on a one-to-one correspondence between LUT nodes and single-target gates. This second step has to guarantee that the resulting reversible network is garbage-free, meaning that intermediate values that are initially stored on clean helper lines are uncomputed. The last step of the hierarchical synthesis method is responsible for decomposing each single-target gate into a quantum circuit. This last step performs STG decomposition to generate quantum circuits for smaller Boolean functions, for which it is possible to take advantage of less scalable techniques.

In this work, I researched possible improvements of the described k-LUT-based hierarchical synthesis framework. The first contribution is described in Section 5.2 and focuses on the cost functions that guide the first step of the procedure: the k-LUT mapping. The original framework performed this step using state-of-the-art mappers, which were developed for the technology mapping of classical logic circuits. Such mappers are typically designed to optimize area and delay—cost functions that do not apply to the quantum compilation problem. For this reason, I proposed a new mapping that takes into account cost functions related to quantum circuit synthesis. The second contribution concerns the second step of the procedure and is described in Section 5.3. The key idea is that there are several strategies for mapping the logic of each network's node into the reversible circuit, called uncomputation or *pebbling* strategies. Indeed, to find a valid strategy it is necessary to solve the reversible pebbling game. I encoded the problem as a SAT problem and solved it using state-of-the-art SAT solvers, e.g., Z3. This procedure enables the user to fix the available number of helper qubits and to consequently obtain a reversible circuit that minimizes the number of single-target gates while not exceeding the given constraint. The improvement brought by the combination of these two contributions is shown in Section 5.4. In particular, the presented experiments show how the results obtained by the state-of-the-art procedure are improved both in the number of qubits and the number of gates.

Chapter 6 presents hierarchical compilation algorithms that are specific for XAGs. While the k-LUT-based method applies to any logic network and enables the user to specify the number of helper qubits, it fails at reducing the number of generated T gates. Differently, the XAG-based methods presented in this chapter are capable of synthesizing quantum circuits with $O(N)$ qubits and $O(N)$ gates, where N is the number of nodes of the XAG. In particular, the first presented method targets the minimization of the T-count, and achieves an upper bound on the number of T gates that is four times the multiplicative complexity of the network (see Section 6.3.1). The second algorithm focuses on reducing the T-depth of the final quantum circuit. It achieves the same T-count of the previous method but leverages more qubits. The quantum circuits synthesized using these two algorithms, with application in post-quantum cryptography, are reported in Section 6.4.

A key feature of such algorithms is that it is possible to correlate the resources of the compiled quantum circuits with the characteristics of the XAGs. This suggests that future works on the topic may focus on restructuring and rewriting the XAG to reduce the identified features: the number of AND nodes, i.e., the multiplicative complexity (affecting the T-count) and the multiplicative depth (affecting the T-depth).

The last algorithm of this chapter, in Section 6.3.3, is a SAT-based algorithm to find uncomputation strategies by playing the reversible pebbling game on an abstract graph derived from an XAG in which each node is characterized by a weight. Such a weight depends on the number of gates required for the quantum circuit to perform the node's functionality. The section also describes how the problem of reducing the overall weight of a pebbling strategy with P pebbles and K steps is encoded in an additional clause for the SAT problem. Section 6.4 shows plots illustrating the efficiency of this last algorithm to trade-off qubits for gates, and to optimize the obtained results.

In general, the problem of finding a good uncomputation strategy by solving the reversible pebbling game finds an application whenever the computation of a quantum algorithm is decomposed into several parts that need to be composed back together. Pebbling may be an enabling technology for quantum memory management, allowing us to fit the (often limited) available memory resources of quantum systems. In this research work, I focused on solving the problem iteratively using SAT. With the basic formulation of the problem, this is quite a scalable method, applicable to networks with hundreds of nodes. Nevertheless, as soon as the problem is strongly constrained in the number of pebbles or one attempts to solve variations of the original problem, e.g., weighted pebbling and in-place pebbling, the SAT-based approach becomes too slow. For this reason, further research on the topic could focus on the development of heuristic pebbling strategies.

Chapter 7 describes several techniques for the decomposition of single-target gates into quantum circuits. When the control functions are small enough it is possible to build a database of precomputed optimal implementations. Section 7.1 describes how the database's size can be compressed using spectral classification and how any Boolean function up to 5 inputs can be synthesized from a spectral-equivalent function implementation in the database by only adding CNOT and NOT gates, i.e., with the same T-count and T-depth. The database is publicly available[1] and contains circuits expressed in the Q# language.

A general method to synthesize quantum circuits is the ESOP decomposition. Section 7.2 proposes a study on the impact of different ESOP synthesis methods on the resulting compiled circuits. From the study, it is possible to conclude that different cost functions must be

[1] https://github.com/gmeuli/stg-benchmark

considered when these expressions are used in quantum compilation flows. In addition, advanced ESOP synthesis methods integrated, e.g., in the hierarchical synthesis flow, show promising results. The number of literals is identified as a better fitting cost function than the number of terms in the ESOP expression. In particular, a MAX-SAT problem is encoded to exactly synthesize ESOP expression minimizing a custom cost function. The technique is unfortunately not applicable to large Boolean functions. In the latter case, state-of-the-art heuristics should be used. As a future research direction, one could investigate heuristic methods guided by specific cost functions.

The last presented technique addresses the decomposition of STGs controlled by large Boolean functions, given some helper lines available. In this case, it is possible to perform a k-LUT decomposition and store intermediate results on helper qubits. Then each obtained STG is decomposed using the ESOP decomposition.

Chapter 8 is dedicated to two techniques developed for the optimization of quantum circuits. The first one is based on solving the weighted graph matching problem and relies on pairing certain multiple-controlled Toffoli gates to reduce the corresponding number of T gates. The second one is based on SAT, and seeks the reduction of the number of CNOT gates in a Clifford+T quantum circuit. The optimization method extracts subcircuits consisting of CNOT and T gates, rewrites them using an exact SAT approach that keeps the T-count unchanged while reducing the number of CNOT to the minimum. Then, the original circuit is reconstructed. The technique shows an improvement of up to 45.45%, as reported in Section 8.2.4.

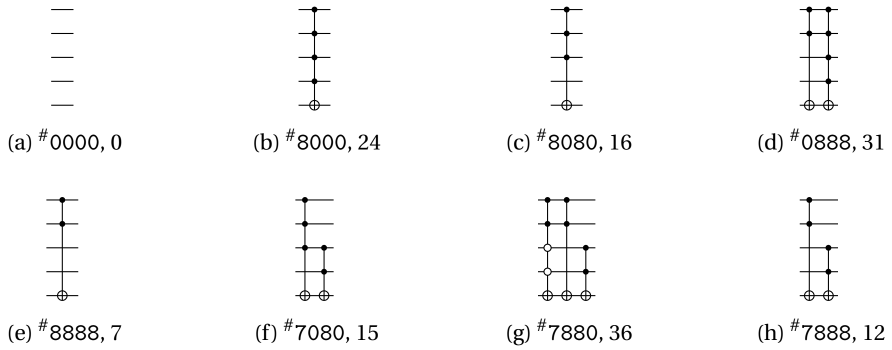
This concludes the part of this thesis dedicated to the synthesis and the optimization of quantum circuits.

Chapter 9 presents a methodology to automatically manage approximation errors in quantum programming languages. The methodology introduces language support, to facilitate expressing approximation errors into quantum programs. Then, it automatizes the choice of the best accuracy parameters to guarantee an upper bound on the overall approximation errors. The constraint and cost functions of the corresponding optimization process are extracted directly from the quantum program in the form of (near-)symbolic expressions.

The methodology can be added to any quantum software framework, thereby greatly facilitating the resource estimation of quantum programs. Such integration will allow even domain experts from, e.g., chemistry or machine learning, to write accuracy-aware quantum programs without having to manually derive and prove error bounds. Future work could implement improved handling of branching on measurement results and repeat-until-success-like structures. To handle such programs, the methodology requires additional input such as the maximal or expected iteration count (e.g., as a program annotation). For verification purposes, one could instrument the code in order to assert that the actual number of iterations does not deviate (too much) from the provided estimate. Future work could also compare upper bounds to actually achieved errors on example applications. Currently, the methodology does not take into account gate cancellations that may be performed by circuit optimization. This, in addition to the repeated use of the triangle inequality to bound the overall error, likely leads to pessimistic error bounds.

Finally, Chapter 10 describes the C++ open-source library that I develop and maintain, and that implements many of the algorithms and techniques presented in this manuscript.

A Database integrated into RevKit



(a) #0000, 0

(b) #8000, 24

(c) #8080, 16

(d) #0888, 31

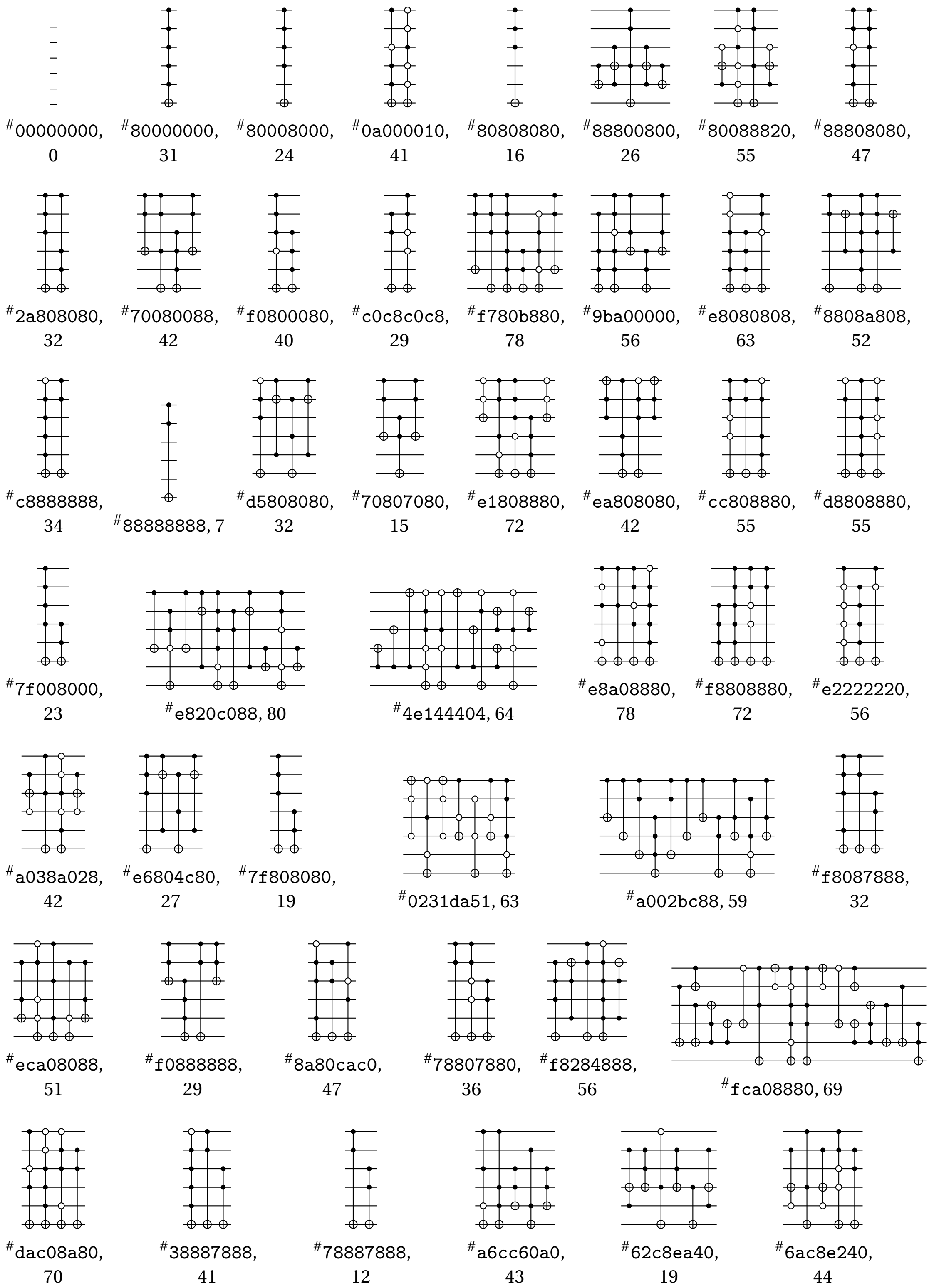
(e) #8888, 7

(f) #7080, 15

(g) #7880, 36

(h) #7888, 12

Figure A.1 – Representative functions, starting point Toffoli networks, and obtained T-count for single-target gates with 4-input control functions.



#00000000, 0 · #80000000, 31 · #80008000, 24 · #0a000010, 41 · #80808080, 16 · #88800800, 26 · #80088820, 55 · #88808080, 47

#2a808080, 32 · #70080088, 42 · #f0800080, 40 · #c0c8c0c8, 29 · #f780b880, 78 · #9ba00000, 56 · #e8080808, 63 · #8808a808, 52

#c8888888, 34 · #88888888, 7 · #d5808080, 32 · #70807080, 15 · #e1808880, 72 · #ea808080, 42 · #cc808880, 55 · #d8808880, 55

#7f008000, 23 · #e820c088, 80 · #4e144404, 64 · #e8a08880, 78 · #f8808880, 72 · #e2222220, 56

#a038a028, 42 · #e6804c80, 27 · #7f808080, 19 · #0231da51, 63 · #a002bc88, 59 · #f8087888, 32

#eca08088, 51 · #f0888888, 29 · #8a80cac0, 47 · #78807880, 36 · #f8284888, 56 · #fca08880, 69

#dac08a80, 70 · #38887888, 41 · #78887888, 12 · #a6cc60a0, 43 · #62c8ea40, 19 · #6ac8e240, 44

Figure A.2 – Representative functions, starting point Toffoli networks, and obtained T-count for single-target gates with 5-input control functions.

Bibliography

[1] K. A. Britt and T. S. Humble, “High-performance computing with quantum processing units,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, pp. 1–13, 2017.

[2] R. Van Meter and S. J. Devitt, “Local and distributed quantum computation,” *arXiv preprint arXiv:1605.06951*, 2016.

[3] M. Soeken and M. K. Thomsen, “White dots *do* matter: Rewriting reversible logic circuits,” in *Int’l Conf. on Reversible Computation*, pp. 196–208, 2013.

[4] E. Rutherford, “The scattering of α and β particles by matter and the structure of the atom,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 21, no. 125, pp. 669–688, 1911.

[5] J. C. Maxwell, “A dynamical theory of the electromagnetic field,” *Philosophical Trans. of the Royal Society of London*, vol. 155, pp. 459–512, 1865.

[6] S. Carroll, *Something deeply hidden: quantum worlds and the emergence of spacetime*. Dutton, 2019.

[7] M. Planck, “Ueber das Gesetz der Energieverteilung im Normalspectrum,” *Annalen der Physik*, vol. 309, no. 3, pp. 553–563, 1901.

[8] A. Einstein, “Über einem die Erzeugung und Verwandlung des Lichtes betreffenden heuristischen Gesichtspunkt,” *Annalen der Physik*, vol. 322, no. 6, pp. 132–148, 1905.

[9] L. De Broglie, “Recherches sur la théorie des quanta,” vol. 10, no. 3, pp. 22–128, 1925.

[10] E. Schrödinger, “Quantisierung als Eigenwertproblem,” *Annalen der Physik*, vol. 384, no. 4, pp. 361–376, 1926.

[11] H. Capellmann, “Later criticism of the Copenhagen interpretation,” in *The Development of Elementary Quantum Theory*, pp. 77–81, Springer, 2017.

[12] H. Everett III, “"Relative state" formulation of quantum mechanics,” *Reviews of Modern Physics*, vol. 29, no. 3, p. 454, 1957.

[13] B. S. DeWitt and N. Graham, *The many worlds interpretation of quantum mechanics*, vol. 63. Princeton University Press, 2015.

[14] "IBM quantum computing." https://www.ibm.com/quantum-computing/. Accessed: 5 October 2020.

[15] R. P. Feynman, "Simulating physics with computers," *Int'l Journal of Theoretical Physics*, vol. 21, no. 6/7, 1982.

[16] D. Deutsch, "Quantum theory, the Church–Turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.

[17] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 439, no. 1907, pp. 553–558, 1992.

[18] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *IEEE Symp. on Foundations of Computer Science*, pp. 124–134, Nov 1994.

[19] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[20] B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. James, A. Gilchrist, and A. G. White, "Experimental demonstration of a compiled version of Shor's algorithm with quantum entanglement," *Physical Review Letters*, vol. 99, no. 25, p. 250505, 2007.

[21] D. P. DiVincenzo, "The physical implementation of quantum computation," *Fortschritte der Physik: Progress of Physics*, vol. 48, no. 9-11, pp. 771–783, 2000.

[22] J. Preskill, "Quantum computing and the entanglement frontier," *arXiv preprint arXiv:1203.5813*, 2012.

[23] A. W. Harrow and A. Montanaro, "Quantum computational supremacy," *Nature*, vol. 549, no. 7671, pp. 203–209, 2017.

[24] J. Preskill, "Why I called it 'Quantum Supremacy'," *Quanta Magazine*, 2019.

[25] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, "Validating quantum computers using randomized model circuits," *Physical Review A*, vol. 100, no. 3, p. 032328, 2019.

[26] K. R. Brown, J. Kim, and C. Monroe, "Co-designing a scalable quantum computer with trapped atomic ions," *npj Quantum Information*, vol. 2, no. 1, pp. 1–10, 2016.

[27] J. I. Cirac and P. Zoller, "Quantum computations with cold trapped ions," *Physical Review Letters*, vol. 74, no. 20, p. 4091, 1995.

[28] C. Monroe, D. Meekhof, B. King, W. M. Itano, and D. J. Wineland, "Demonstration of a fundamental quantum logic gate," *Physical Review Letters*, vol. 75, no. 25, p. 4714, 1995.

[29] B. Lekitsch, S. Weidt, A. G. Fowler, K. Mølmer, S. J. Devitt, C. Wunderlich, and W. K. Hensinger, "Blueprint for a microwave trapped ion quantum computer," *Science Advances*, vol. 3, no. 2, 2017.

[30] R. Blakestad, C. Ospelkaus, A. VanDevender, J. Amini, J. Britton, D. Leibfried, and D. J. Wineland, "High-fidelity transport of trapped-ion qubits through an X-junction trap array," *Physical Review Letters*, vol. 102, no. 15, p. 153002, 2009.

[31] D. Kielpinski, C. Monroe, and D. J. Wineland, "Architecture for a large-scale ion-trap quantum computer," *Nature*, vol. 417, no. 6890, pp. 709–711, 2002.

[32] C. Monroe, R. Raussendorf, A. Ruthven, K. Brown, P. Maunz, L.-M. Duan, and J. Kim, "Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects," *Physical Review A*, vol. 89, no. 2, p. 022317, 2014.

[33] Y. Wang, M. Um, J. Zhang, S. An, M. Lyu, J.-N. Zhang, L.-M. Duan, D. Yum, and K. Kim, "Single-qubit quantum memory exceeding ten-minute coherence time," *Nature Photonics*, vol. 11, no. 10, pp. 646–650, 2017.

[34] T. P. Harty, D. T. C. Allcock, C. J. Ballance, L. Guidoni, H. A. Janacek, N. M. Linke, D. N. Stacey, and D. M. Lucas, "High-fidelity preparation, gates, memory, and readout of a trapped-ion quantum bit," *Physical Review Letters*, vol. 113, p. 220501, Nov 2014.

[35] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, "Trapped-ion quantum computing: Progress and challenges," *Applied Physics Reviews*, vol. 6, no. 2, p. 021314, 2019.

[36] "Achieving Quantum Volume 128 on the Honeywell quantum computer." https://www.honeywell.com/us/en/news/2020/09/achieving-quantum-volume-128-on-the-honeywell-quantum-computer. Accessed: 12 December 2020.

[37] K. Wright, K. Beck, S. Debnath, J. Amini, Y. Nam, N. Grzesiak, J.-S. Chen, N. Pisenti, M. Chmielewski, C. Collins, *et al.*, "Benchmarking an 11-qubit quantum computer," *Nature Communications*, vol. 10, no. 1, pp. 1–6, 2019.

[38] "IonQ unveils world's most powerful quantum computer." IonQ Press Release, 1 October 2020.

[39] Y. Nakamura, Y. A. Pashkin, and J. S. Tsai, "Coherent control of macroscopic quantum states in a single-Cooper-pair box," *Nature*, vol. 398, no. 6730, pp. 786–788, 1999.

[40] J. Kelly, "A preview of Bristlecone, Google's new quantum processor," *Google Research Blog*, vol. 5, 2018.

[41] C. Nay, "IBM unveils world's first integrated quantum computing system for commercial use," *IBM Research Communications*, 2019.

[42] C. Nay, "IBM opens quantum computation center in New York; brings world's largest fleet of quantum computing systems online, unveils new 53-qubit quantum system for broad use," *IBM Research Communications*, 2019.

[43] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[44] E. Pednault, J. Gunnels, D. Maslov, and J. Gambetta, "On quantum supremacy," *IBM Research Blog*, vol. 21, 2019.

[45] P. Jurcevic, A. Javadi-Abhari, L. S. Bishop, I. Lauer, D. F. Bogorin, M. Brink, L. Capelluto, O. Günlük, T. Itoko, N. Kanazawa, A. Kandala, G. A. Keefe, K. Krsulich, W. Landers, E. P. Lewandowski, D. T. McClure, G. Nannicini, A. Narasgond, H. M. Nayfeh, E. Pritchett, M. B. Rothwell, S. Srinivasan, N. Sundaresan, C. Wang, K. X. Wei, C. J. Wood, J.-B. Yau, E. J. Zhang, O. E. Dial, J. M. Chow, and J. M. Gambetta, "Demonstration of quantum volume 64 on a superconducting quantum computing system," *arXiv preprint arXiv:2008.08571*, 2020.

[46] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, "A quantum-classical cloud platform optimized for variational hybrid algorithms," *Quantum Science and Technology*, vol. 5, p. 024003, Apr 2020.

[47] L. Childress and R. Hanson, "Diamond NV centers for quantum computing and quantum networks," *MRS bulletin*, vol. 38, no. 2, pp. 134–138, 2013.

[48] M. Veldhorst, J. Hwang, C. Yang, A. Leenstra, B. de Ronde, J. Dehollain, J. Muhonen, F. Hudson, K. M. Itoh, A. Morello, *et al.*, "An addressable quantum dot qubit with fault-tolerant control-fidelity," *Nature Nanotechnology*, vol. 9, no. 12, pp. 981–985, 2014.

[49] M. Veldhorst, C. Yang, J. Hwang, W. Huang, J. Dehollain, J. Muhonen, S. Simmons, A. Laucht, F. Hudson, K. M. Itoh, *et al.*, "A two-qubit logic gate in silicon," *Nature*, vol. 526, no. 7573, pp. 410–414, 2015.

[50] L. Petit, H. Eenink, M. Russ, W. Lawrie, N. Hendrickx, S. Philips, J. Clarke, L. Vandersypen, and M. Veldhorst, "Universal quantum logic in hot silicon qubits," *Nature*, vol. 580, no. 7803, pp. 355–359, 2020.

[51] S. D. Sarma, M. Freedman, and C. Nayak, "Majorana zero modes and topological quantum computation," *npj Quantum Information*, vol. 1, no. 1, pp. 1–13, 2015.

[52] E. Knill, "Conventions for quantum pseudocode," tech. rep., Los Alamos National Lab., NM (United States), 1996.

[53] R. LaRose, "Overview and comparison of gate level quantum software platforms," *Quantum*, vol. 3, p. 130, 2019.

[54] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore, "Quantum programming languages," *Nature Reviews Physics*, pp. 1–14, 2020.

[55] K. M. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Real World Domain Specific Languages Workshop*, pp. 7:1–7:10, 2018.

[56] G. Aleksandrowicz and et al., "Qiskit: An Open-source Framework for Quantum Computing," *Zenodo*, 2019.

[57] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," *arXiv preprint arXiv:1608.03355*, 2017.

[58] A. Ho and D. Bacon, "Announcing Cirq: An open source framework for NISQ algorithms," *Google AI Blog*, 2018.

[59] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: a scalable quantum programming language," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 333–342, 2013.

[60] A. Javadi-Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. Chong, and M. Martonosi, "ScaffCC: a framework for compilation and analysis of quantum computing programs," in *ACM Conference on Computing Frontiers*, 2014.

[61] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *arXiv preprint arXiv:1612.08091*, 2016.

[62] A. Paler, S. J. Devitt, and A. G. Fowler, "Synthesis of arbitrary quantum circuits to topological assembly," *Scientific Reports*, vol. 6, no. 1, pp. 1–16, 2016.

[63] D. Wecker, B. Bauer, B. K. Clark, M. B. Hastings, and M. Troyer, "Gate-count estimates for performing quantum chemistry on small quantum computers," *Physical Review A*, vol. 90, no. 2, p. 022305, 2014.

[64] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[65] S. Bravyi, D. Browne, P. Calpin, E. Campbell, D. Gosset, and M. Howard, "Simulation of quantum circuits by low-rank stabilizer decompositions," *Quantum*, vol. 3, p. 181, 2019.

[66] Y. Shi, R. Tao, X. Li, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu, "CertiQ: A mostly-automated verification of a realistic quantum compiler," *arXiv preprint arXiv:1908.08963*, 2020.

[67] D. Kudrow, K. Bier, Z. Deng, D. Franklin, Y. Tomita, K. R. Brown, and F. T. Chong, "Quantum rotations: a case study in static and dynamic machine-code generation for quantum computers," in *Int'l Symposium on Computer Architecture*, pp. 166–176, 2013.

[68] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[69] S. Pirandola, U. L. Andersen, L. Banchi, M. Berta, D. Bunandar, R. Colbeck, D. R. Englund, T. Gehring, C. Lupo, C. Ottaviani, J. Pereira, M. Razavi, J. S. Shaari, M. Tomamichel, V. C. Usenko, G. Vallone, P. Villoresi, and P. Wallden, "Advances in quantum cryptography," *arXiv preprint arXiv:1906.01645*, 2019.

[70] C. H. Bennett and G. Brassard, "Quantum cryptography: public key distribution and coin tossing," *Theoretical Computer Science*, vol. 560, no. 12, pp. 7–11, 2014.

[71] G. H. Low and I. L. Chuang, "Hamiltonian simulation by qubitization," *Quantum*, vol. 3, p. 163, 2019.

[72] R. Babbush, D. W. Berry, Y. R. Sanders, I. D. Kivlichan, A. Scherer, A. Y. Wei, P. J. Love, and A. Aspuru-Guzik, "Exponentially more precise quantum simulation of fermions in the configuration interaction representation," *Quantum Science and Technology*, vol. 3, no. 1, p. 015006, 2017.

[73] R. Babbush, D. W. Berry, I. D. Kivlichan, A. Y. Wei, P. J. Love, and A. Aspuru-Guzik, "Exponentially more precise quantum simulation of fermions in second quantization," *New Journal of Physics*, vol. 18, no. 3, p. 033032, 2016.

[74] S. McArdle, S. Endo, A. Aspuru-Guzik, S. Benjamin, and X. Yuan, "Quantum computational chemistry," *arXiv preprint arXiv:1808.10402*, 2018.

[75] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta, "Supervised learning with quantum-enhanced feature spaces," *Nature*, vol. 567, no. 7747, pp. 209–212, 2019.

[76] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *ACM Symp. on Theory of Computing*, pp. 212–219, 1996.

[77] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical Review Letters*, vol. 103, no. 15, p. 150502, 2009.

[78] N. Wiebe, D. Braun, and S. Lloyd, "Quantum algorithm for data fitting," *Physical Review Letters*, vol. 109, no. 5, p. 050505, 2012.

[79] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates," in *Post-Quantum Cryptography*, 2016.

[80] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover oracles for quantum key search on AES and LowMC," in *Int'l Conference on the Theory and Applications of Cryptographic Techniques*, pp. 280–310, Springer, 2020.

[81] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," 2016.

[82] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms," in *International Conference on Post-Quantum Cryptography*, pp. 425–444, Springer, 2020.

[83] M. Amy, O. D. Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3," *arXiv preprint arXiv:1603.09383*, 2016.

[84] A. Parent, M. Roetteler, and K. M. Svore, "Reversible circuit compilation with space constraints," *arXiv preprint arXiv:1510.00377*, 2015.

[85] B. Langenberg, H. Pham, and R. Steinwandt, "Reducing the cost of implementing the advanced encryption standard as a quantum circuit," *IEEE Trans. on Quantum Engineering*, vol. 1, pp. 1–12, 2020.

[86] P. Kim, D. Han, and K. C. Jeong, "Time–space complexity of quantum search algorithms in symmetric cryptanalysis: applying to AES and SHA-2," *Quantum Information Processing*, vol. 17, Oct 2018.

[87] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[88] E. Fredkin and T. Toffoli, "Conservative logic," *Int'l Journal of Theoretical Physics*, vol. 21, no. 3-4, pp. 219–253, 1982.

[89] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Physical Review A*, vol. 52, no. 5, p. 3457, 1995.

[90] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Physical Review A*, vol. 93, no. 2, p. 022311, 2016.

[91] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conference*, pp. 318–323, 2003.

[92] D. Maslov, G. W. Dueck, and D. M. Miller, "Techniques for the synthesis of reversible Toffoli networks," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 4, p. 42, 2007.

[93] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, “Exact multiple-control Toffoli network synthesis with SAT techniques,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 703–715, 2009.

[94] M. Soeken, R. Wille, and R. Drechsler, “Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition,” in *Int'l Design and Test Symp.*, pp. 143–148, 2010.

[95] M. Soeken, F. Mozafari, B. Schmitt, and G. De Micheli, “Compiling permutations for superconducting QPUs,” in *DATE*, pp. 1349–1354, 2019.

[96] A. De Vos and Y. Van Rentergem, “Young subgroups for reversible computers,” *Advances in Mathematics of Communications*, vol. 2, no. 2, pp. 183–200, 2008.

[97] K. Fazel, M. Thornton, and J. E. Rice, “ESOP-based Toffoli gate cascade generation,” in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 206–209, 2007.

[98] M. Amy, P. Azimzadeh, and M. Mosca, “On the controlled-NOT complexity of controlled-NOT–phase circuits,” *Quantum Science and Technology*, vol. 4, no. 1, p. 015002, 2019.

[99] N. Schuch and J. Siewert, “Programmable networks for quantum algorithms,” *Physical Review Letters*, vol. 91, no. 2, p. 027902, 2003.

[100] M. Soeken and M. Roetteler, “Quantum circuits for functionally controlled NOT gates,” *arXiv preprint arXiv:2005.12310*, 2020.

[101] M. Soeken, G. W. Dueck, and D. M. Miller, “A fast symbolic transformation based algorithm for reversible logic synthesis,” in *Int'l Conf. on Reversible Computation*, pp. 307–321, 2016.

[102] A. Parent, M. Roetteler, and K. M. Svore, “Reversible circuit compilation with space constraints,” *arXiv preprint arXiv:1510.00377*, 2015.

[103] M. Rawski, “Application of functional decomposition in synthesis of reversible circuits,” in *Int'l Conf. on Reversible Computation*, pp. 285–290, 2015.

[104] M. Soeken and A. Chattopadhyay, “Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis,” in *Design Automation Conference*, pp. 149:1–149:6, 2016.

[105] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, “Design automation and design space exploration for quantum computers,” in *Design, Automation and Test in Europe*, pp. 470–475, 2017.

[106] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, “LUT-based hierarchical reversible logic synthesis,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[107] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *arXiv preprint arXiv:1710.07345*, 2017.

[108] M. Amy, D. Maslov, and M. Mosca, "Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1476–1489, 2014.

[109] V. Gheorghiu, S. M. Li, M. Mosca, and P. Mukhopadhyay, "Reducing the CNOT count for Clifford+T circuits on NISQ architectures," *arXiv preprint arXiv:2011.12191*, 2020.

[110] J. M. Smith, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: concrete resource estimation in quantum algorithms," *arXiv preprint arXiv:1412.0625*, 2014.

[111] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, "Elucidating reaction mechanisms on quantum computers," *Proceedings of the National Academy of Sciences*, p. 201619152, 2017.

[112] A. Scherer, B. Valiron, S.-C. Mau, S. Alexander, E. van den Berg, and T. 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*, vol. 16, Jan 2017.

[113] S.-H. Hung, K. Hietala, S. Zhu, M. Ying, M. Hicks, and X. Wu, "Quantitative robustness analysis of quantum programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 31:1–31:29, 2019.

[114] J. Watrous, "Semidefinite programs for completely bounded norms," *arXiv preprint arXiv:0901.4709*, 2009.

[115] G. Bioul, M. Davio, and J.-P. Deschamps, "Minimization of ring-sum expansions of Boolean functions," *Philips Research Reports*, vol. 28, pp. 17–36, 1973.

[116] A. Mishchenko and M. A. Perkowski, "Fast heuristic minimization of exclusive-sum-of-products," in *Reed-Muller Workshop*, 2001.

[117] S. Stergiou, K. Daskalakis, and G. K. Papakonstantinou, "A fast and efficient heuristic ESOP minimization algorithm," in *ACM Great Lakes Symposium on VLSI*, pp. 78–81, 2004.

[118] T. Sasao, "AND-EXOR expressions and their optimization," in *Logic Synthesis and Optimization*, Kluwer Academic, 1993.

[119] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, pp. 24–40, Springer, 2010.

[120] Synopsys, "Design compiler graphical." https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html. Accessed: April 2020.

[121] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, “Majority-inverter graph: A new paradigm for logic optimization,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[122] L. G. Amarù, P.-E. Gaillardon, S. Mitra, and G. De Micheli, “New logic synthesis as nanotechnology enabler,” *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2168–2195, 2015.

[123] P.-E. Gaillardon, L. G. Amarù, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, “The programmable logic-in-memory (PLiM) computer,” in *Design, Automation and Test in Europe*, pp. 427–432, 2016.

[124] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis,” in *Design Automation Conference*, pp. 532–535, 2006.

[125] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, “Hierarchical reversible logic synthesis using LUTs,” in *Design Automation Conference*, pp. 78:1–78:6, 2017.

[126] G. Meuli, M. Soeken, M. Roetteler, and G. De Micheli, “Enumerating optimal quantum circuits using spectral classification,” in *Int’l Symp. on Circuits and Systems*, pp. 1–5, 2020.

[127] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, “Fast Boolean matching based on NPN classification,” in *Int’l Symp. on Field-Programmable Technology*, pp. 310–313, 2013.

[128] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. De Micheli, “Heuristic NPN classification for large functions using AIGs and LEXSAT,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, pp. 212–227, 2016.

[129] C. R. Baugh, “Generation of representative functions of the NPN equivalence classes of unate Boolean functions,” *IEEE Trans. on Computers*, vol. 21, no. 12, pp. 1373–1379, 1972.

[130] C. R. Edwards, “The application of the Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis,” *IEEE Trans. on Computers*, vol. 24, no. 1, pp. 48–62, 1975.

[131] R. J. Lechner, “Harmonic analysis of switching functions,” in *Recent Developments in Switching Theory*, pp. 121–228, Academic Press, 1971.

[132] E. R. Berlekamp and L. R. Welch, “Weight distributions of the cosets of the $(32,6)$ Reed-Muller code,” *IEEE Trans. on Information Theory*, vol. 18, no. 1, pp. 203–207, 1972.

[133] J. A. Maiorana, “A classification of the cosets of the Reed-Muller code $\mathscr{R}(1,6)$,” *Mathematics of Computation*, vol. 57, no. 195, pp. 403–414, 1991.

[134] J. E. Fuller, *Analysis of affine equivalent Boolean functions for cryptography.* PhD thesis, Queensland University of Technology, 2003.

[135] D. M. Miller and M. Soeken, *An Algorithm for Linear, Affine and Spectral Classification of Boolean Functions*, pp. 195–215. 2020.

[136] S. L. Hurst, *The Logical Processing of Digital Signals.* London, UK: Arnold, 1978.

[137] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic.* London, UK: Academic Press, 1985.

[138] J. L. Walsh, "A closed set of normal orthogonal functions," *American Journal of Mathematics*, vol. 45, no. 1, pp. 5–24, 1923.

[139] H. Rademacher, "Einige Sitze uber Reihen von allgemeinen Orthogonalfunktionen," *Math. Annen*, vol. 87, pp. 112–138, 1922.

[140] M. A. Thornton, R. Drechsler, and D. M. Miller, "Computation of spectral coefficients," in *Spectral Techniques in VLSI CAD*, pp. 83–116, Springer, 2001.

[141] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[142] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *Int'l Conf. on Computer-Aided Design*, pp. 752–759, 2004.

[143] S. Ray, A. Mishchenko, N. Eén, R. K. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures," in *Design, Automation and Test in Europe*, pp. 1579–1584, 2012.

[144] S. A. Cook, "The complexity of theorem-proving procedures," in *ACM Symp. on Theory of Computing*, p. 151–158, Association for Computing Machinery, 1971.

[145] B. A. Trakhtenbrot, "A survey of russian approaches to perebor (brute-force searches) algorithms," *Annals of the History of Computing*, vol. 6, no. 4, pp. 384–400, 1984.

[146] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability.* IOS Press, 2009.

[147] D. E. Knuth, *The Art of Computer Programming, Volume 3, Second Edition.* Addison-Wesley, 1998.

[148] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2008.

[149] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation = lazy clause generation," in *Principles and Practice of Constraint Programming*, pp. 544–558, Springer Berlin Heidelberg, 2007.

[150] M. Soeken, G. Meuli, B. Schmitt, F. Mozafari, H. Riener, and G. De Micheli, "Boolean satisfiability in quantum compilation," *Philosophical Trans. of the Royal Society A*, vol. 378, no. 2164, p. 20190161, 2019.

[151] S. Hill and W. K. Wootters, “Entanglement of a pair of quantum bits,” *Physical Review Letters*, vol. 78, no. 26, p. 5022, 1997.

[152] A. G. Fowler, A. M. Stephens, and P. Groszkowski, “High-threshold universal quantum computation on the surface code,” *Physical Review A*, vol. 80, p. 052312, Nov 2009.

[153] S. Bravyi and A. Kitaev, “Universal quantum computation with ideal Clifford gates and noisy ancillas,” *Physical Review A*, vol. 71, p. 022316, 2005.

[154] A. G. Fowler, “Time-optimal quantum computation,” *arXiv preprint arXiv:1210.4626*, 2013.

[155] D. Maslov, G. W. Dueck, and D. M. Miller, “Synthesis of Fredkin-Toffoli reversible networks,” *IEEE Trans. of VLSI Systems*, vol. 13, no. 6, pp. 765–769, 2005.

[156] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler, “Ancilla-free synthesis of large reversible functions using binary decision diagrams,” *Journal of Symbolic Computation*, vol. 73, pp. 1–26, 2016.

[157] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler, “Synthesis of reversible circuits with minimal lines for large functions,” in *Asia and South Pacific Design Automation Conference*, pp. 85–92, 2012.

[158] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[159] R. Drechsler, A. Finder, and R. Wille, “Improving ESOP-based synthesis of reversible logic using evolutionary algorithms,” in *European Conference on the Applications of Evolutionary Computation*, pp. 151–161, Springer, 2011.

[160] T. Toffoli, “Reversible computing,” in *Int’l. Coll. on Automata, Languages, and Programming*, pp. 632–644, 1980.

[161] B. Valiron, “Generating reversible circuits from higher-order functional programs,” in *Int’l Conf. on Reversible Computation*, pp. 289–306, 2016.

[162] R. Wille and R. Drechsler, “BDD-based synthesis of reversible logic for large functions,” in *Design Automation Conference*, pp. 270–275, 2009.

[163] A. Chattopadhyay, A. Littarru, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, “Reversible logic synthesis via biconditional binary decision diagrams,” in *Int’l Symp. on Multiple-Valued Logic*, pp. 2–7, 2015.

[164] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, “A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 818–830, 2013.

[165] D. Gosset, V. Kliuchnikov, M. Mosca, and V. Russo, "An algorithm for the T-count," *Quantum Information and Computation*, vol. 14, no. 15–16, pp. 1261–1276, 2014.

[166] C. Jones, "Low-overhead constructions for the fault-tolerant Toffoli gate," *Physical Review A*, vol. 87, no. 2, p. 022328, 2013.

[167] C. Gidney, "Halving the cost of quantum addition," *Quantum*, vol. 2, no. 74, pp. 10–22331, 2018.

[168] P. Selinger, "Quantum circuits of T-depth one," *Physical Review A*, vol. 87, p. 042302, 2013.

[169] N. Abdessaied, M. Amy, M. Soeken, and R. Drechsler, "Technology mapping of reversible circuits to Clifford+T quantum circuits," in *Int'l Symp. on Multiple-Valued Logic*, pp. 150–155, 2016.

[170] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM Journal on Computing*, vol. 18, no. 4, pp. 766–776, 1989.

[171] R. Královic, "Time and space complexity of reversible pebbling," in *Conf. on Current Trends in Theory and Practice of Informatics*, pp. 292–303, 2001.

[172] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

[173] S. M. Chan, *Pebble games and complexity.* PhD thesis, University of California, Berkeley, 2013.

[174] S. M. Chan, M. Lauria, J. Nordstrom, and M. Vinyals, "Hardness of approximation in PSpace and separation results for pebble games," in *IEEE Symp. on Foundations of Computer Science*, vol. 00, pp. 466–485, 2015.

[175] E. Knill, "An analysis of Bennett's pebble game," *arXiv preprint arXiv:math/9508218*, 1995.

[176] M. Li and P. Vitányi, "Reversibility and adiabatic computation: trading time and space for energy," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 452, no. 1947, pp. 769–789, 1996.

[177] B. Komarath, J. Sarma, and S. Sawlani, "Pebbling meets coloring: reversible pebble game on trees," *Journal of Computer and System Sciences*, vol. 91, pp. 33–41, 2018.

[178] S. F. de Rezende, O. Meir, J. Nordström, and R. Robere, "Nullstellensatz size-degree trade-offs from reversible pebbling," *arXiv preprint arXiv:2001.02481*, 2020.

[179] G. Meuli, M. Soeken, M. Roetteler, N. Bjorner, and G. De Micheli, "Reversible pebbling game for quantum memory management," in *Design, Automation and Test in Europe*, 2019.

[180] J. W. Bos, C. Costello, H. Hisil, and K. Lauter, “Fast cryptography in genus 2,” in *Advances in Cryptology – EUROCRYPT 2013*, pp. 194–210, Springer Berlin Heidelberg, 2013.

[181] M. Soeken, H. Riener, W. Haaswijk, and G. D. Micheli, “The EPFL logic synthesis libraries,” *arXiv preprint arXiv:1805.05121*, 2018.

[182] G. Meuli, M. Soeken, M. Roetteler, and G. De Micheli, “ROS: Resource constrained oracle synthesis for quantum circuits,” in *Quantum Physics and Logic*, 2019.

[183] E. Testa, M. Soeken, H. Riener, L. Amaru, and G. De Micheli, “A logic synthesis toolbox for reducing the multiplicative complexity in logic networks,” in *Design, Automation and Test in Europe*, 2020.

[184] T. Häner and M. Soeken, “Lowering the T-depth of quantum circuits by reducing the multiplicative depth of logic networks,” *arXiv preprint arXiv:2006.03845*, 2020.

[185] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. De Micheli, “The role of multiplicative complexity in compiling low T-count oracle circuits,” *Int’l Conf. on Computer-Aided Design*, 2019.

[186] G. Meuli, B. Schmitt, R. Ehlers, H. Riener, and G. De Micheli, “Evaluating ESOP optimization methods in quantum compilation flows,” in *Int’l Conf. on Reversible Computation*, 2019.

[187] M. Perkowski and M. Chrzanowska-Jeske, “An exact algorithm to minimize mixed-radix exclusive sums of products for incompletely specified Boolean functions,” in *Int’l Symp. on Circuits and Systems*, pp. 1652–1655, 1990.

[188] C. M. Li and F. Manyà, “MaxSAT, hard and soft constraints,” in *Handbook of Satisfiability*, pp. 613–631, 2009.

[189] H. Riener, R. Ehlers, B. d. O. Schmitt, and G. D. Micheli, *Exact Synthesis of ESOP Forms*, pp. 177–194. Cham: Springer International Publishing, 2020.

[190] A. Ignatiev, A. Morgado, and J. Marques-Silva, “PySAT: A Python toolkit for prototyping with SAT oracles,” in *Theory and Applications of Satisfiability Testing*, pp. 428–437, 2018.

[191] I. Zhegalkin, “The technique of calculation of statementsin symbolic logic,” in *Mathe. Sbornik*, vol. 34, pp. 9–28, 1927. (in Russian).

[192] R. Drechsler, “Pseudo-Kronecker expressions for symmetric functions,” *IEEE Trans. on Computers*, vol. 48, no. 9, pp. 987–990, 1999.

[193] G. Meuli, M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, “A best-fit mapping algorithm to facilitate ESOP-decomposition in Clifford+ T quantum network synthesis,” in *Asia and South Pacific Design Automation Conference*, pp. 664–669, IEEE Press, 2018.

[194] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, “RevKit: A toolkit for reversible circuit design,” *Multiple-Valued Logic and Soft Computing*, vol. 18, no. 1, pp. 55–65, 2012.

[195] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, “Logic synthesis for quantum computing,” *arXiv preprint arXiv:1706.02721*, 2017.

[196] D. M. Miller, R. Wille, and R. Drechsler, “Reducing reversible circuit cost by adding lines,” *Multiple-Valued Logic and Soft Computing*, vol. 19, no. 1–3, pp. 185–201, 2012.

[197] R. Wille, M. Soeken, C. Otterstedt, and R. Drechsler, “Improving the mapping of reversible circuits to quantum circuits using multiple target lines,” in *Asia and South Pacific Design Automation Conference*, pp. 145–150, IEEE, 2013.

[198] C. Bandyopadhyay, H. Rahaman, and R. Drechsler, “Improved cube list based cube pairing approach for synthesis of ESOP based reversible logic,” *Trans. on Computational Science*, vol. 24, pp. 129–146, 2014.

[199] S. P. Parlapalli, C. Vudadha, and M. B. Srinivas, “Optimizing the reversible circuits using complementary control line transformation,” in *Int’l Conf. on Reversible Computation*, pp. 111–126, 2017.

[200] K.-C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar, “Dag-map: Graph-based fpga technology mapping for delay optimization,” *IEEE Design and Test of Computers*, vol. 9, no. 3, p. 7–20, 1992.

[201] J. Edmonds, “Paths, trees, and flowers,” *Canadian Journal of Mathematics*, vol. 17, pp. 361–379, 1965.

[202] R. Duan and S. Pettie, “Linear-time approximation for maximum weight matching,” *Journal of the ACM*, vol. 61, no. 1, pp. 1–23, 2014.

[203] Y. S. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, “Automated optimization of large quantum circuits with continuous parameters,” *arXiv preprint arXiv:1710.07345*, 2017.

[204] K. N. Patel, I. L. Markov, and J. P. Hayes, “Efficient synthesis of linear reversible circuits,” *arXiv preprint arXiv:quant-ph/0302002*, 2003.

[205] G. Meuli, M. Soeken, and G. De Micheli, “SAT-based {CNOT, T} quantum circuit synthesis,” in *Int’l Conf. on Reversible Computation*, pp. 175–188, Springer, 2018.

[206] M. Suchara, J. Kubiatowicz, A. I. Faruque, F. T. Chong, C. Lai, and G. Paz, “QuRE: The quantum resource estimator toolbox,” in *Int’l Conf. on Computer Design*, pp. 419–426, 2013.

[207] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[208] T. Häner, M. Roetteler, and K. M. Svore, “Managing approximation errors in quantum programs,” *arXiv preprint arXiv:1807.02336*, 2018.

[209] V. Kliuchnikov, D. Maslov, and M. Mosca, “Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates,” *Quantum Information and Computation*, vol. 13, pp. 607–630, June 2013.

[210] N. J. Ross and P. Selinger, “Optimal ancilla-free Clifford+T approximation of z-rotations,” *Quantum Information and Computation*, vol. 16, no. 11&12, pp. 901–953, 2016.

[211] A. Kitaev, “Quantum measurements and the abelian stabilizer problem,” *arXiv preprint arXiv:quant-ph/9511026*, 1995.

[212] D. Coppersmith, “An approximate Fourier transform useful in quantum factoring,” *arXiv preprint arXiv:quant-ph/0201067*, 2002.

[213] A. Paetznick and K. M. Svore, “Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries,” *Quantum Information and Computation*, vol. 14, no. 15–16, p. 1277–1301, 2014.

[214] E. Bernstein and U. Vazirani, “Quantum complexity theory,” *SIAM Journal on Computing*, vol. 26, pp. 1411–1473, oct 1997.

[215] D. Poulin, M. B. Hastings, D. Wecker, N. Wiebe, A. C. Doberty, and M. Troyer, “The trotter step size required for accurate quantum simulation of quantum chemistry,” *Quantum Information and Computation*, vol. 15, no. 5–6, p. 361–384, 2015.

[216] S. Beauregard, “Circuit for Shor’s algorithm using 2n+3 qubits,” *Quantum Information and Computation*, vol. 3, 06 2002.

[217] T. G. Draper, “Addition on a quantum computer,” *arXiv preprint quant-ph/0008033*, 2000.

[218] J. H. Conway and R. K. Guy, *The Book of Numbers.* Springer New York, 1996.

[219] Wolfram Research Inc., “Mathematica, Version 12.0,” 2019. Champaign, IL.

Giulia Meuli

Contact information

giu.meuli@gmail.com

Research interests

Quantum computing, quantum compilation, reversible computation, logic synthesis

Education

PhD in Electrical Engineering

May 2017 - Dec 2020, Lausanne, CH
Institute: École Polytechnique Fédérale de Lausanne
Thesis: Program compilation for large-scale quantum computers

Master Degree in Nanotechnologies for ICTs

Oct. 2016, Torino, IT
Institute: Politecnico di Torino/INP-Phelma/EPFL
Thesis: Rad-hard Overcurrent Circuit Protection of the DCDC converter for HL-LHC experiment upgrade
(with highest honours)

Bachelor Degree in Electronic Engineering

July 2014, Roma, IT
Institute: Università Roma Tre
(with highest honours)

Experience

Microsoft, Summer Intern

May-July 2019, Zurich, CH
title: Automatic accuracy distribution for quantum programming languages
description: I worked with the LLVM Compiler infrastructure (LibTooling and compiler passes)

Microsoft, Summer Intern

May-July 2018, Redmond, WA, USA
title: Quantum memory management
description: I developed a SAT-based quantum memory management tool using Microsoft's Z3 prover and its C# API

CERN, Intern

Feb-Sept 2016, Genève, CH
title: Master thesis in analog design for rad-hard electronics
description: I designed a DC/DC converted for the new high-luminosity experiment

Tyndall National Institute, Summer Intern

Jun-Aug 2015, Cork, IE
title: Lab-on-a-Chip for DNA amplification and detection
description: experiment set up and FEM simulations for Lab-on-a-Chip experiment

Università di Roma Tre (Biometric Lab), Lab student assistant

2013-2014, Roma, IT
title: Human identification using EEG signals

Teaching experience

Lab. in EDA based design, M.Sc. course, Fall 2019, EPFL

Design Technologies for Integrated Systems, M.Sc. course, Fall 2018, EPFL

Awards

Best Interactive Presentation by the DATE Best IP Award committee, DATE 2019

Patents

T. Häner, G. Meuli and M. Roetteler, *"Automatic accuracy management for quantum programs via symbolic resource estimation"*, U.S. Patent Application 16/843,733. Filed: April 8, 2020

M. Roetteler, G. Meuli, Microsoft Technology Licensing LLC, 2020. *"Reversible pebbling game for quantum memory management"*. U.S. Patent Application 16/457,408.

Professional services

ACM JETC, Journal on Emerging Technologies in Computing Systems

Reversible Computation Conference

IEEE TVLSI, IEEE Transactions on Very Large Scale Integration Systems

Journal of Circuits, Systems, and Computers

Invited talks

COST Action IC1405 meeting 2019, Valletta, Malta, *"Reversible Pebbling game for quantum memory management"*

2nd International Workshop on Quantum Compilation (IWQC) 2018, *"Reversible Pebbling Game for Quantum Memory Management"*

2nd International Workshop on Quantum Compilation (IWQC) 2018, *"ESOP Optimization for Quantum Cost Reduction"*

Novi Sad University invited by Prof. Jovanka Pantović, 2018, *"Reversible circuits for quantum computing"*

Nis University invited by Prof. Radomir S. Stanković and Prof Milena M. Stanković, 2018, *"Reversible circuits for quantum computing*

COST Action IC1405 meeting 2018, Larcana, Cyprus, *"Spectral Classification for the Synthesis of Quantum Circuits"*

COST Action IC1405 meeting 2017, Toruń, Poland, *"LUT-based hierarchical reversible synthesis"*

Publications

G. Meuli, M. Soeken, M. Roetteler, and T. Häner, “Enabling accuracy-aware quantum

compilers using symbolic resource estimation," in *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 2020.

G. Meuli, M. Soeken, M. Roetteler, and G. De Micheli, "Enumerating optimal quantum circuits using spectral classification," in *IEEE Int'l Symp. on Circuits and Systems*, 2020.

G. Meuli, M. Soeken, M. Roetteler, N. Bjorner, and G. D. Micheli, "Reversible pebbling game for quantum memory management," in *Design, Automation and Test in Europe*, 2019.

G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. De Micheli, "The role of multiplicative complexity in compiling low T-count oracle circuits," in *Int'l Conf. on Computer-Aided Design*, 2019.

G. Meuli, B. Schmitt, R. Ehlers, H. Riener, and G. De Micheli, "Evaluating ESOP optimization methods in quantum compilation flows," in *Int'l Conf. on Reversible Computation*, 2019.

G. Meuli, M. Soeken, M. Roetteler, and G. De Micheli, "ROS: Resource-constrained oracle synthesis for quantum computers," in *Quantum Physics and Logic*, 2019.

M. Soeken, **G. Meuli**, B. Schmitt, F. Mozafari, H. Riener, and G. De Micheli, "Boolean satisfiability in quantum compilation," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190161, 2019.

G. Meuli, M. Soeken, and G. De Micheli, "SAT-based {CNOT, T} quantum circuit synthesis," in *Int'l Conf. on Reversible Computation*, 2018.

G. Meuli, M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "A best-fit mapping algorithm to facilitate ESOP-decomposition in Clifford+T quantum network synthesis," in *Asia and South Pacific Design Automation Conference*, 2018.

Pre-prints

M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, **G. Meuli**, F. Mozafari and G. D. Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2019.

Authored papers without proceedings

G. Meuli, M. Soeken, M. Roetteler and G. De Micheli, "ROS: Resource constrained oracle synthesis for quantum computers", in *Int'l Workshop on Logic Synthesis*, 2019.

G. Meuli, M. Soeken, M. Roetteler, D. M. Miller, M. Amy, N. Wiebe, G. De Micheli, "Estimating single-target gate T-count using spectral classification", in *Int'l Workshop on Logic Synthesis*, 2018.

G. Meuli, M. Soeken, P. E. Gaillardon, G. De Micheli, “A compiler for parallel and resource-constrained programmable in-memory computing”, in *Int’l Workshop on Logic Synthesis*, 2017.

Under review

G. Meuli, M. Soeken, G. De Micheli, “Xor-And-Inverter graphs for quantum compilation”.