Optimization of Circuit Transformation and Scheduling in Quantum Compilers

（量子コンパイラーにおける回路変換とスケジューリングの最適化）

March 2021

Toshinari Itoko

Optimization of Circuit Transformation and Scheduling in Quantum Compilers

Toshinari Itoko

Degree Programs in Systems and Information Engineering
University of Tsukuba

March 2021

Abstract

Rapid progress in quantum technology in this decade has dramatically increased the coherence time of qubits (quantum bits) and the fidelity of operations on qubits. Although currently available quantum computers are noisy and intermediate in scale, it is believed that they will be used in some specific applications such as quantum chemistry, machine learning, optimization, and sampling in the near future.

Along with progress in quantum computing hardware, development of the software stack has also become more active. A quantum compiler, which translates a quantum program into a sequence of control instructions, is one of the most essential software components to provide a human-friendly programming interface for quantum algorithms. In quantum compilers, optimizing the output instructions is very important: For noisy quantum computers, it increases the probability of completing all the operations before any qubit decoheres, yielding computational results with higher fidelity. Even for fault-tolerant quantum computers, optimization increases the throughput.

In this thesis, we address two important optimization tasks in quantum compilers: *quantum circuit mapping* and *quantum circuit scheduling*. We provide their formulations as classical optimization problems considering commutation between quantum gates, which are not fully considered in previous studies. We also provide exact and heuristic algorithms for solving them. We examine how considering gate commutation in our algorithms improves the quality of solutions by computational experiments in practical settings.

The thesis is organized as follows. After reviewing studies on quantum compilers, we consider the problem of mapping quantum circuits to quantum computers, whose operations are limited by their coupling architecture, with as few additional gates as possible. We propose exact and heuristic algorithms for quantum circuit mapping, which use gate commutation rules and a gate transformation rule of Bridge gate. We compare these algorithms with the state-of-the-art algorithms by computational experiments with circuits of a commonly used benchmark dataset. Next, we consider the problem of scheduling operations in a given circuit with the shortest total execution time. We demonstrate that quantum circuit scheduling can be seen as an optimization problem only when we consider gate commutation and can be interpreted as a special type of job-shop problem. We provide Constraint Programming and Mixed Integer Programming formulations and show how solving circuit scheduling independently improved the schedule length through experiments with real circuits and a compiler.

We believe that our algorithms are useful in practical quantum compilers and the developed techniques provide new insight into optimization in quantum compilers. We hope that they will be helpful to advance software stack development and improve the utility of quantum computers.

謝辞 (Acknowledgements)

数多くの方々の支えと励ましのお蔭で私はこの博士論文を完成させることができました。ここに謝意を表したいと思います。

まず初めに、指導教員の久野誉人教授に心より御礼申し上げます。量子回路の最適化という専門外の応用を含む研究課題に取り組む私を快く研究室に受け入れて指導くださったことに深く感謝いたします。副査をお引き受けくださった亀山幸義教授、繁野麻衣子教授、中井央准教授、佐野良夫准教授にも御礼申し上げます。

共同研究者でありまた同僚でもある今道貴司氏、ルディー・レイモンド氏の両名には、ほぼ全ての論文執筆に際して、まるで研究室の先輩のように的確なアドバイスをいただき、いくら感謝しても感謝しきれないほどです。本当にありがとうございました。

I would like to express my gratitude to Dr. Dmitri Maslov and Dr. Ali Javadi-Abhari for number of helpful comments and suggestions. I am also thankful to Lauren Capelluto, Thomas Alexander, Naoki Kanazawa, Andrew W Cross, Atsushi Matsuo, and all members in IBM Quantum team, who gave me useful feedback on the research projects, which this dissertation based on.

久野・佐野研究室の方々にも感謝いたします。コロナ禍で直接お会いできませんでしたが、オンラインで若い才気に触れることができて刺激になりました。IBM 東京基礎研究所の方々にも感謝いたします。特に私のもう一つの研究室とも言える量子チームのメンバーからはいつも温かい励ましをいただきありがとうございました。また就職しながらの博士課程修学にあたって全面的に支援くださった研究所の所長・副所長に御礼申し上げます。さらに研究に行き詰まった際の憩いの場を提供してくださった研究所の豆部・カメラ部の方々にも感謝いたします。

最後に、私の家族、特に妻の絵美と三人の子どもたちに感謝します。お陰さまでこの研究に専念しまた挫けず完遂することができました。ありがとう。

This page is mainly written in Japanese.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Quantum Computer

A quantum computer is a device that processes quantum information, using quantum-mechanical phenomena such as superposition and entanglement. It usually stores information within *qubits* (quantum bits), carries out computation by applying unitary operations on qubits, and obtains computational results by observing the state of qubits. Since the concept of quantum computing was suggested by Richard Feynman [1] and the solution to a toy problem with a quantum algorithm was found by David Deutsch [2] in the early 1980s, various quantum algorithms have been proposed: prime factorization [3], search-based algorithms [4], and linear systems of equations [5] (refer to [6] for more information). These quantum algorithms potentially offer significant advantages over their classical counterparts, but most of them are meant to be run on fault-tolerant quantum computers that include error correction mechanisms. Unfortunately, no fault-tolerant quantum computer has yet been realized. However, recent progress in quantum technology in this decade has dramatically increased the coherence time of qubits and the fidelity of operations on qubits. For example, in 2016, IBM made a 5-qubit quantum computer publicly available via a cloud platform [7] and offered a 65-qubit quantum computer to private users in 2020. Although currently available quantum computers are noisy and have only dozens of qubits, it is expected that they can be used in some specific applications, such as quantum chemistry, machine learning, optimization, and sampling, in the near future [8].

There are many candidate technologies for physically implementing a quantum computer, including nuclear magnetic resonance (NMR), trapped-ion, and superconduction [9]. The first successful experiment implementing a quantum computer was based on NMR technology, which is used to control ensembles of nuclear spins in a single molecule as a qubit [10]. Although NMR quantum computers with up to 7 qubits were built, the technical difficulty of further scale-up was revealed [11]. In contrast, two other types of quantum computers are rapidly progressing. A trapped-ion quantum computer uses the electron or nuclear energy states of ions trapped using electromagnetic fields as qubits and manipulate them via optical excitation using lasers. As of 2020, trapped-ion quantum computers with

at least 10 qubits have been developed by Honeywell and IonQ and made available via cloud platforms such as Azure Quantum [12] and Amazon Bracket [13]. A superconducting quantum computer uses the discrete Cooper-pair charge states on a superconducting tunnel junction, called a Josephson junction, as a qubit and manipulates it by microwave pulses. As of 2020, superconducting quantum computers with no less than 28 qubits has been developed by Google, IBM, and Rigetti, and made available via APIs of quantum SDKs such as Cirq [14], Qiskit [15] and Forrest [16]. In this thesis, we focus on superconducting quantum computers as target devices for compilation.

1.2 Quantum Compiler

Along with the progress in quantum computing hardware, software stack development has also become more active. Various software architectures for quantum computing have been proposed [17, 18, 19, 20, 21, 22] and many development tool kits for quantum programming have been provided [14, 15, 16, 23, 24, 25, 26, 27].

A quantum compiler (i.e., a compiler for quantum computers) is one of the most important software components. It translates a quantum program into a sequence of control instructions supported by the target quantum computer. It is essential for quantum computer users to run their algorithms without handwriting control instructions of the computing device. Other important components include simulators and verifiers [28]. In this thesis, we consider compilers for noisy quantum computers and do not consider software modules specialized for error correction, which can be seen as an independent module to be attached later.

1.2.1 Quantum Circuit Model

Throughout this thesis, we assume a quantum program is provided in the form of a quantum circuit, which is a mathematical abstraction. The quantum circuit model is one of the most commonly used for quantum computation and was originally developed in the analysis of computability [29] and complexity [30] of quantum computation. A quantum circuit is a sequence of operations applying to a subset of qubits. Many of them are unitary operations called *gates*. A common non-unitary operation is measurement, which observes a qubit and collapses its state into binary state, $|0\rangle$ or $|1\rangle$. We refer to a gate or a measurement as an *operation* in this thesis. A quantum circuit can be represented by a list of operations with its acting qubits: for example, $[H(1), CX(1,2), X(2)]$. Here, $H(1)$ denotes a Hadamard gate acting on qubit 1, $CX(1,2)$ denotes a Controlled-NOT (or CNOT) gate acting on control qubit 1 and target qubit 2, and $X(2)$ denotes a NOT (or Pauli X) gate acting on qubit 2. A quantum circuit is often depicted in a circuit diagram, as shown in Fig. 1.1. Single-qubit gates, such as a Hadamard gate (denoted as H) or a Pauli X gate (denoted as X), are depicted as boxes labeled with their names. A CNOT gate is depicted with vertical lines whose ends are $\bullet$ and $\oplus$ denoting the control and target qubits, respectively. These

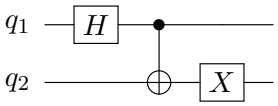


Figure 1.1: Diagram representation of a quantum circuit

gates act on qubits each represented by a horizontal line in the figure, and are applied from left to right.

In this thesis, we focus on the combinatorial structure of quantum circuits and largely disregard what the circuits compute (refer to [11], for example, for details of quantum computing). In addition, considering quantum circuits as inputs, we focus on the parts of compilers that do not depend on programming language, although that is an interesting research topic itself and, in fact, many languages for quantum programming are proposed (e.g., [31, 32, 33, 34, 35, 36, 37]; refer to [38, 39] for more information).

1.2.2 Optimization of Quantum Circuits

There are three major tasks in quantum compilation (discussed in details in Chapter 2):

- **Synthesis**: Generating a quantum circuit from a given specification provided as a reversible Boolean function, a unitary matrix, etc.
- **Mapping**: Transforming a quantum circuit into an equivalent one satisfying constraints on the qubit topology of a target quantum computing device.
- **Scheduling**: Scheduling operations in a quantum circuit while mapping each operation into control instructions of a target hardware in order to determine its processing time.

These are naturally optimization tasks, since there exist many possible synthesized, mapped, or scheduled circuits for a given input, and the best among them should be chosen. Optimizing the output circuit is very important. For noisy quantum computers, it increases the probability of completing all the operations before any qubit decoheres, yielding computational results with higher fidelity. Even for fault-tolerant quantum computers, it increases the throughput by decreasing the execution time of each operation on the error correction code.

Comparing with the synthesis task, the mapping and scheduling tasks are not thoroughly studied yet, especially those in compilation for superconducting quantum computers, whose technologies have progressed most recently. This thesis work aims to provide better methods (formulations and algorithms) for optimizing those mapping and scheduling tasks, assuming superconducting quantum computers as compilation targets.

1.3 Overview of this Thesis

In this thesis, we address two optimization problems that arise in quantum compilation: quantum circuit mapping and quantum circuit scheduling. We provide their formulations as classical optimization problems taking into account gate commutation, which has not been fully considered in previous studies. We also provide exact and heuristic algorithms for solving them. We examine how the consideration of gate commutation in our algorithms improves the optimality of solutions by computational experiments in practical settings.

The thesis is organized as follows. In Chapter 2, we briefly review the three major tasks in quantum compilation: synthesis, mapping and scheduling of quantum circuits.

In Chapter 3, we consider quantum circuit mapping, which is a problem of mapping quantum circuits to noisy quantum computers, whose operations are limited by their coupling architecture, with as few additional gates as possible. We propose exact and heuristic algorithms for the circuit mapping, which use gate commutation rules and a gate transformation rule of Bridge gate. We compare them with the state-of-the-art circuit mapping algorithms by computational experiments with circuits of a commonly used benchmark dataset.

In Chapter 4, we consider quantum circuit scheduling, which is a problem of scheduling operations in a given circuit with the shortest total execution time. We demonstrate that the problem can be seen as an optimization problem only when we consider the gate commutation, and it can be interpreted as a special type of job-shop problem. We provide Constraint Programming and Mixed Integer Programming formulations and show how solving quantum circuit scheduling independently improves the schedule length through experiments with real circuits and a compiler.

Chapter 2

Optimization Tasks in Quantum Compilers

In this chapter, we briefly review the three major tasks in quantum compilation: synthesis, mapping and scheduling of quantum circuits (listed again as below).

- **Synthesis**: Generating a quantum circuit from a given specification provided as a reversible Boolean function, a unitary matrix, etc.
- **Mapping**: Transforming a quantum circuit into an equivalent one satisfying constraints on the qubit topology of a target quantum computing device.
- **Scheduling**: Scheduling operations in a quantum circuit while mapping each operation into control instructions of a target hardware in order to determine its processing time.

These are naturally optimization tasks, since there exist many possible synthesized, mapped, or scheduled circuits for a given input, and the best among them should be chosen. Although this thesis mainly focuses on the mapping and scheduling tasks, a review on the synthesis task is included in this chapter for the sake of completeness.

2.1 Quantum Circuit Synthesis

Quantum circuit synthesis is the task of generating a quantum circuit composed of elementary gates (or basis gates) from a given specification or decomposing complex gates in a given circuit into basis gates. The basis gates are carefully chosen depending on what kind of quantum computer we consider as a target. If we think about a noisy quantum computer, which can be seen as an analog computer, we often use an infinite number of basis gates (i.e., we include gates with real parameters in basis gates). In contrast, if we think about a fault-tolerant quantum computer, which is essentially a digital computer, we use a finite set of gates as basis gates. If there are any gates with real parameters in a

given circuit, they must be approximately decomposed using the finite basis gates with arbitrary precision; such decomposition methods are well studied (e.g., [11, 40, 41, 42]).

Problems in quantum circuit synthesis can be classified according to the kind of specification given as input.

When an arbitrary unitary matrix is given, the problem is called quantum-logic synthesis or *unitary synthesis*, which is a theoretically important problem. In general, any unitary operation over n qubits can be represented by a $2^n \times 2^n$ unitary matrix. There is a large variety of research on unitary synthesis, such as combination with mapping tasks [43, 44] and optimization of T-gates preparing compilers for fault-tolerant quantum computers [45, 46]. Synthesis of an arbitrary unitary matrix from one-qubit gates and CNOT gates is especially well studied, and the upper bound on the number of CNOT gates has been improved [47, 48]. However, the size of the input matrix and the number of required gates grow exponentially with n, and thus it is not practical to assume unitary matrices as the input for large n. Consequently, thinking about restricted class of input makes sense in practice as follows.

When considering a reversible logic function as input, the problem is called reversible logic synthesis. The logic reversibility of computation was proved first in terms of a Turing machine [49] and then in terms of circuits [50]. As reversible logic synthesis has applications other than quantum circuit synthesis, there is a large amount of research on it from various points of view, such as how to represent and store the input logic function, what objective to be optimized, and what algorithm to be used. Much of the research up to 2013 was surveyed by Saeedi et al. [51], and there has been more recent progress in asymptotically optimal synthesis [52, 53] and in ancilla-free synthesis of Hidden Weighted Bit function [54].

For the synthesis of an arbitrary reversible logic function, an exponential number of basis gates are still required in the worst case. Therefore, synthesis of arithmetic operations, such as multiplication or Galois Field arithmetic, as well as that of uniquely quantum operations, such as Quantum Fourier Transformation (QFT), has a rich history of research. In fact, there are many studies even on addition, one of the most fundamental arithmetic operations [55, 56, 57, 58, 59]. Synthesis of specific operations is more restricted but frequently used in various applications such as quantum factoring (e.g., [60, 61, 62, 63]). Decomposition of multiple-control Toffoli (MCT) gates is one of the most well-studied topics (e.g., [64, 65, 66, 67]).

Another important problem in quantum circuit synthesis is state preparation, which generates a quantum circuit to create a target quantum state from a specific quantum state [68]. Recently, state preparation and unitary synthesis have been considered as special cases of a generalized problem called isometry synthesis [69].

Simplification of quantum circuits is a sub-task that can be considered as post-processing of the synthesis task. Several methods specialized for it are proposed (e.g., [70, 71]).

2.2 Quantum Circuit Mapping

Quantum circuit mapping is the task of transforming a given quantum circuit into an equivalent quantum circuit so that it satisfies the physical constraint of a target quantum computer, called connectivity constraint or *coupling constraint*, which restricts application of two-qubit gates to specific pairs of qubits. This restriction is common in some type of quantum computers, such as superconducting quantum computers. A coupling constraint can be represented by an undirected graph, called a connectivity graph or *coupling graph*, whose nodes represent qubits and edges represent qubit pairs such that two-qubit gates are physically operable. Two examples of coupling graphs with five qubits are shown in Fig. 2.1. In both examples, a CNOT gate between qubits 1 and 2—i.e., $CX(1,2)$—is

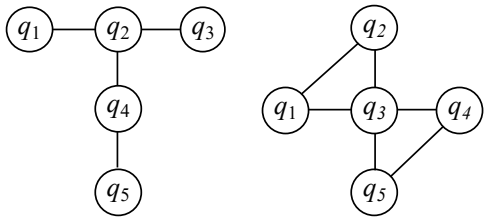


Figure 2.1: Coupling graphs for different five-qubit systems

operable but $CX(1,5)$ is not operable. The sparser the coupling graph becomes, the more additional gates are required to satisfy the constraint, which lessens the fidelity of the computational outcome. Therefore, quantum circuit mapping is a critical task that affects the actual performance of quantum computers with coupling constraint [72, 73].

Quantum circuit mapping has various names in the literature [74]: quantum circuit mapping/transformation, mapping/compiling quantum circuits to nearest neighbor architecture, or qubit routing/allocation. In this thesis, we call the whole problem *quantum circuit mapping*. We call the sub-problem of finding the best initial layout of qubits *qubit assignment* and call the sub-problem of finding the best way to resolve the violation of a coupling constraint under a given initial qubit layout *qubit routing*.

The content of quantum circuit mapping changes depending on which type of quantum computer we consider as a target. For trapped-ion quantum computers, we do not need to consider any mapping task because all qubits are usually coupled, i.e., the coupling graph is perfect. However, in order to apply two-qubit operations, qubits must be moved next to each other. That makes the succeeding task, scheduling, very complex [75, 76, 77]. NMR quantum computers are able to operate any pair of two qubits as well, but the cost (operation time) for each qubit pair differs. Therefore, it makes sense to look for the best qubit layout. That means quantum circuit mapping coincides with qubit assignment for NMR quantum computers [78].

There are many variations in the formulation of quantum circuit mapping, and a few of them as decision problems are shown to be NP-complete [78, 79, 80]. We refer to more related work on quantum circuit mapping in Section 3.6.

2.3 Quantum Circuit Scheduling

Quantum circuit scheduling is the task of determining the execution start time for each operation in a given quantum circuit while mapping each operation into a corresponding sequence of control instructions such as microwave pulses. The literature includes various ways of handling the scheduling task. Some consider it as the next task (or a post-processing task) of the mapping task, as in this thesis [81, 82], while others do not distinguish the mapping task but rather consider it as integrated task with the mapping task [83, 84]. In the case of trapped-ion quantum computers, qubits must be "carried" to a place where they are operated on demand, so the scheduling task becomes more complex and is different from those considered in this thesis (e.g., [75, 76, 77]). Although quantum circuit scheduling is well-defined without depending on the choice of basis gates, it is becoming common to use basis gates implemented as single control instructions rather than composite ones [82, 85]. As post-processing of the scheduling task, a method for filling the idle time with special pulse sequences is proposed to reduce noise and improve the fidelity of computation results [86]. We refer to more related work on quantum circuit scheduling in Section 4.5.

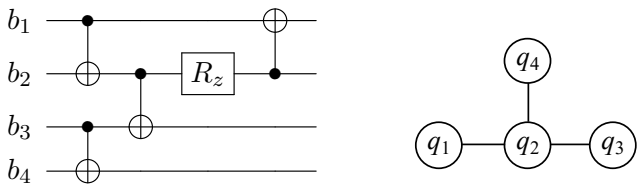
In this thesis, we assume the mapping from an operation to a sequence of control instructions is given. Although the optimization of control instructions is beyond the scope of quantum compilers, it is an important topic in quantum system control and there is a large amount of research, which cannot all be listed here, ranging from pulse shape optimization [87, 88, 89] to system model learning [90].

Chapter 3

Quantum Circuit Mapping

3.1 Overview

To run a quantum algorithm on a quantum computing device, the quantum algorithm must be translated into a series of operations on qubits, i.e. a quantum circuit. A set of quantum gates that consists of arbitrary one-qubit rotation gates and two-qubit controlled-NOT (or CNOT) gates is universal: any unitary operation can be realized by a combination of such basic gates. However, CNOT gates cannot be applied on all pairs of qubits of a target device if the device has limited qubits connectivity or coupling constraint. In this case, an input circuit must be transformed into an output circuit that obeys the limitation of the device. The transformation can further limit the usability of the device, and therefore, it is one of the most essential tasks in quantum compilation. We call this task *quantum circuit mapping* and address it in this chapter[1]. The task deals with mapping qubits in the given virtual circuit to the actual qubits of a target computing device and inserting necessary additional gates in order to satisfy the coupling constraint.

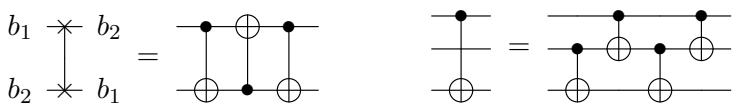


(a) Quantum circuit (b) Coupling graph

Figure 3.1: Quantum circuit (left), and coupling graph of a quantum processor that limits two-qubit operations (right). The problem is finding a mapping of $\{b_i\}$ to $\{q_j\}$ and additional gates so that the circuit can be run in the processor with minimum cost.

Figure 3.1(a) is an example of an input circuit that uses a single-qubit gate, called the phase-shift gate (denoted as R_z), and CNOT gates (depicted with vertical lines whose

[1]Most of the contents in this chapter was published in [91] (its preliminary version was presented in [92])



(a) SWAP gate (b) Bridge gate

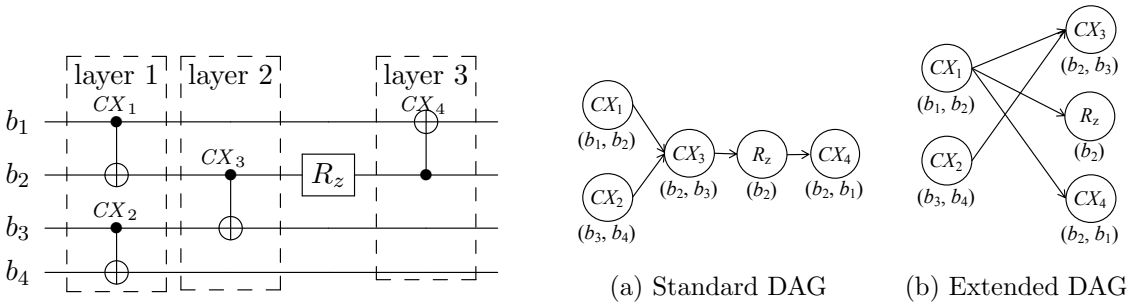
Figure 3.2: Supplementary gates to be used in circuit mapping

ends are $\bullet$ and $\oplus$ denoting the control and target qubits, respectively). Figure 3.1(b) is an example of a *coupling graph*, which is a graph representing on which pairs of qubits CNOT gates can be physically operated. Currently available noisy quantum computers, such as IBM Quantum Systems [7], are limited by their coupling architecture in the sense that not all pairs of qubits can be directly manipulated due to hardware issues. If we map the qubit b_i (as numbered in the circuit) in Fig. 3.1(a) to q_i (as numbered in the hardware) for $i = 1, \ldots, 4$, the CNOT gate from b_3 to b_4 is not directly possible. This can be resolved by swapping control and/or target qubits with their neighboring qubits until they are adjacent in the coupling graph. Such swapping can be carried out by so-called SWAP gates (Fig. 3.2(a)). Another way to run a CNOT gate between non-adjacent qubits is replacing it with a sequence of CNOT gates. This composite gate is called a Bridge gate and is composed of four CNOT gates as shown in Fig. 3.2(b). In either case, we need to add supplementary gates to the original circuit, resulting in more noise and extra computational time since CNOT gates are noisy and take significantly longer computational time than single-qubit gates. Thus, minimizing additional gates in the circuit mapping is important as it translates into minimizing the increase of time and noise cost to run the circuit.

In this chapter, we discuss the drawback of resolving the two-qubit operations layer by layer in many existing approaches and show the importance of taking into account gate commutation rules in Section 3.2. We provide the formulation of quantum circuit mapping as an optimization problem with commutation rules and transformation rules of running CNOT gates on non-adjacent qubits by choice of replacing with Bridge gates or inserting SWAP gates in Section 3.3. Although SWAP and Bridge gates seem to require different numbers of CNOT gates (three as in Fig. 3.2(a), and four as in Fig. 3.2(b)), a Bridge gate results in running the CNOT gate, whereas a SWAP gate only swaps its two qubits. Thus, the total number of additional CNOT gates to run a CNOT gate between an unconnected qubit pair is the same for both SWAP and Bridge gates. However, the SWAP gate permutes the ordering of the logical qubits, whereas the Bridge gate does not. Depending on the sequence of gates afterwards, the selection of SWAP and Bridge gates can affect the possibility of further reducing the additional CNOT gates in the mapping. We provide an exact algorithm for quantum circuit mapping and a heuristic algorithm for qubit routing, either of which leverages both of commutation rules and the Bridge gate in Section 3.4. We show experimental results that confirm our heuristic algorithm outperforms the state-of-the-art routing algorithms in Section 3.5.

3.2 Motivation

Many previous studies, such as [93, 94, 95], assumed a *fixed-layer formulation*, where the layers of a quantum circuit are given as input and all gates in a layer must be mapped, and when necessary resolved with SWAP gates, before any gate in the next layer. A *layer* is defined as a set of CNOT gates that can be executed in parallel (see the left part of Fig. 3.3).



(a) Standard DAG (b) Extended DAG

Figure 3.3: Quantum circuit with three layers (left) and its dependency graphs (right)

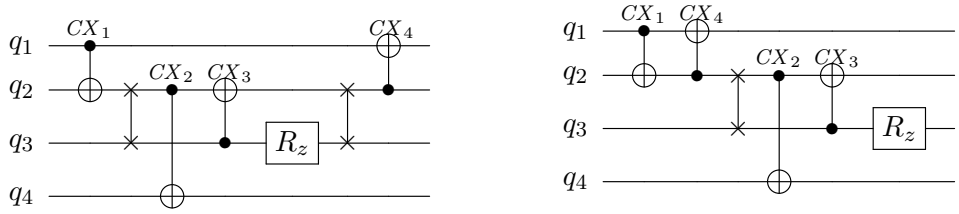
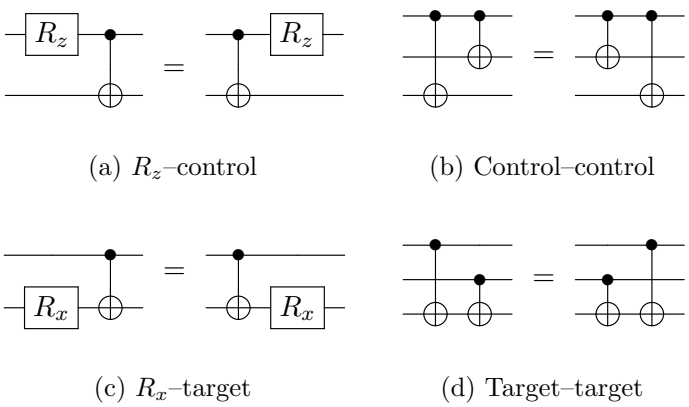


Figure 3.4: Mapped circuits with two SWAP gates (left) and one SWAP gates (right)

The drawback of the fixed-layer approach can be seen from the circuit in Fig. 3.1(a) when the coupling graph is as shown in Fig. 3.1(b). As shown in Fig. 3.4, the number of additional SWAP gates by the fixed-layer formulation is two, but it is one if we allow to be changed the order of gates by applying the standard commutation rules of quantum gates illustrated in Fig. 3.5. The reason is as follows. For the circuit in Fig. 3.3, by the R_z–control rule (Fig. 3.5(a)), we can move the CNOT gate CX_4 to the left so that it precedes R_z. Now, the CNOT gates CX_3 and CX_4 share the control qubit, and by the control–control rule (Fig. 3.5(b)) we can move CX_4 to the left. As a result, when b_i is mapped to q_i, CX_1 and CX_4 can be applied before swapping b_2 and b_3 to run CX_2, CX_3, and CX_5. This is not possible for the circuit *with fixed layers* in Fig. 3.3 because CX_4 in layer 3 cannot precede CX_2 in layer 2.

Because the partial order of gates in a circuit is naturally modeled with a directed acyclic graph (DAG), some studies [15, 96] have considered trivial commutation between consecutive gates that do not share qubits. For example, Fig. 3.3(a) is a DAG representation of the circuit in the left part of Fig. 3.3. We call this *the standard-DAG formulation*. Although it is more flexible than the fixed-layer one, it still may fall into suboptimal



(a) R_z–control (b) Control–control

(c) R_x–target (d) Target–target

Figure 3.5: Standard commutation rules of quantum gates

solutions: it leads to two additional SWAP gates for the circuit in Fig. 3.3.

To overcome this suboptimality problem, we propose a novel formulation taking into account gate commutation rules in combination with the concept *dependency graph* (or extended-DAG) that extends the standard-DAG, as discussed in the following sections. For example, the dependency graph of the circuit in Fig. 3.3 is shown in Fig. 3.3(b).

3.3 Problem Formulation

We propose a formulation of quantum circuit mapping that takes into account gate commutation and transformation rules to minimize the number of additional CNOT gates. We first describe the notations used hereafter.

We use the term *logical qubits* to represent the qubits in the original quantum circuit (denoted by B). This is not to be confused with the term logical qubits in the context of quantum error correction. We use the term *physical qubits* to represent the qubits of the quantum hardware (denoted by Q). For example, $B = \{b_i\}$ and $Q = \{q_j\}$ denote logical qubits and physical qubits in Fig. 3.1(a) and 3.1(b), respectively. Similarly, we call the input circuit the *logical (quantum) circuit* and its corresponding output of quantum circuit mapping the *physical (quantum) circuit*. The physical circuit is equivalent to its logical circuit in terms of computation but may contain additional gates due to coupling constraints. A coupling graph $C = (V_C, E_C)$ is a graph with physical qubits as nodes $V_C (= Q)$ and coupled physical qubits as edges E_C. We assume that $|B| = |Q|$ by adding ancillary qubits to B if $|B| < |Q|$.

A mapping of logical qubits to physical qubits can be seen as a permutation function $B \to Q$, which we call a *layout*. A logical/physical circuit can be seen as a list of gates acting on logical/physical qubits. We say a physical circuit $\hat{L}$ *complies* with a coupling graph C if, for any CNOT gate in $\hat{L}$, its control and target qubits are adjacent in C. A mapping of a logical circuit L to a quantum computer with a coupling graph C can be

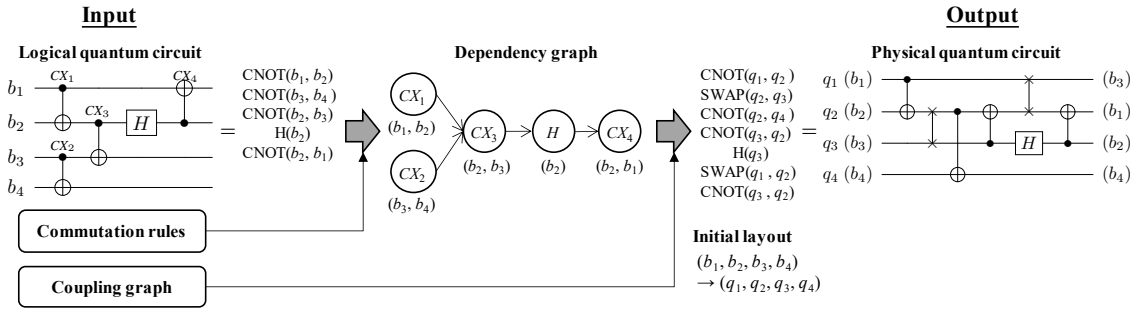


Figure 3.6: Overview of our algorithms for Minimum CNOT Gate Mapping (MCGM)

represented by an *initial layout* l_0 of qubits and a physical circuit $\hat{L}$ that complies with C.

The objective of quantum circuit mapping is to find a cost-optimal physical circuit of a logical circuit. In this chapter, we consider the number of additional CNOT gates as the cost of mapping because they are usually much slower and noisier to run than single-qubit gates.

The problem of finding a corresponding physical circuit of a logical circuit with the minimum number of CNOT gates is formalized as follows.

Minimum CNOT Gate Mapping (MCGM) *Given a logical circuit L, the coupling graph C of a quantum hardware, and a set of commutation rules $\mathcal{R}$, find an initial layout l_0 and an equivalent physical circuit $\hat{L}$ with the fewest CNOT gates that complies with C.*

Note that fixed-layer and standard-DAG formulations in many previous work can be regarded as special cases of MCGM: the former with $\mathcal{R} = \emptyset$, and the latter with $\mathcal{R}$ containing trivial commutation between consecutive gates that do not share qubits. In the next section, we propose algorithms that utilize the commutation rules in Fig. 3.5 for $\mathcal{R}$ in MCGM because they are complete for rules involving two consecutive gates from the universal gate set $\{R_x, R_z, \text{CNOT}\}$, and the transformation rules of CNOT gates with SWAP or Bridge gates. Here R_x and R_z denote single-qubit rotation (respectively, around the x-axis and z-axis) gates. Without loss of generality, we assume any input circuit is composed of gates from the universal gate set.

3.4 Algorithms

We show how to solve the Minimum CNOT Gate Mapping (MCGM) with an exact algorithm based on dynamic programming (DP), and a heuristic algorithm based on a look-ahead scheme. These two algorithms complement each other. The exact algorithm can obtain the optimal solutions but can only be used for small circuits because of its long computational time. The heuristic algorithm can obtain good solutions for a larger circuit within a reasonable amount of computational time. The building blocks of the algorithms are dependency graphs and blocking gates, as explained below. Figure 3.6 shows an overview of our algorithms for MCGM.

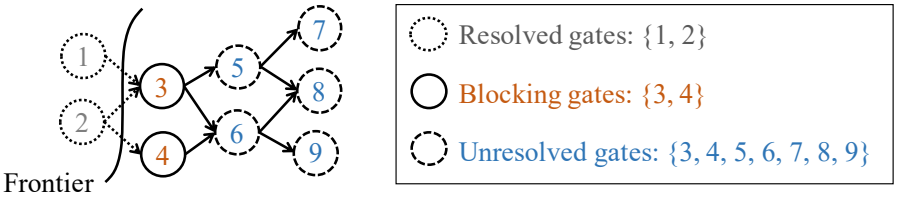


Figure 3.7: Blocking gates in dependency graph

Dependency Graph A dependency graph $D = (V_D, E_D)$ is a DAG that represents the precedence relation of the gates in an input circuit taking commutation rules into account. The nodes V_D correspond to the gates in the logical circuit, and the edges E_D to the dependencies in the order of gates. A gate g_i must precede g_j under the commutation rules if and only if there exists a path from g_i to g_j on D. See Fig. 3.3 for examples of dependency graphs: one for trivial commutation rules of gates that do not share qubits, and the other for commutation rules in Fig. 3.5. The labels below a gate (node) in the dependency graphs represent the logical qubits that the gate acts on, e.g., CX_1 acts on (b_1, b_2).

We can construct a dependency graph D from a given logical quantum circuit straightforwardly as follows (see A.1 for the details). We check all pairs of the gates V_D in the circuit, and include an edge (g_i, g_j) in E_D if the following conditions are satisfied: (1) g_i and g_j share at least one logical qubit b, and (2) the set of gate symbols from g_i to g_j on each shared b in the logical quantum circuit is not a subset of $R_z, \bullet$ (i.e, R_z gate and control of CNOT gate) or a subset of $R_x, \oplus$ (i.e., R_x gate and target of CNOT gate).

Blocking Gates Both exact and heuristic algorithms take the same strategy by resolving the gates in the dependency graph D one by one. They maintain the progress by recording *blocking gates*, which are defined as leading unresolved gates in D as shown in Fig. 3.7. For given gates G (usually blocking gates), we define *blocking gates from G for l* by blocking gates after running leading gates that comply with the coupling graph C repeatedly starting from G under the layout l. At the beginning of mapping, the gates with no in-edges in D, say G_0, can be seen as blocking gates. For a given initial layout l, we define the *initial blocking gates for l* (denoted by $K_0(l)$) to be the blocking gates from G_0 for l.

3.4.1 Exact Algorithm

We design an exact algorithm to solve MCGM based on DP. To achieve this, we define a *state* similar to DP by (l, K), a pair of a layout l and blocking gates K.

A transition between a pair of states occurs when a SWAP or Bridge gate is inserted. For example, let us consider the logical circuit and dependency graph as in Fig. 3.6 and the coupling graph as in Fig. 3.1(b). Then, the blocking gates from $\{CX_1, CX_2\}$ for an initial layout $l_0 : (b_1, b_2, b_3, b_4) \mapsto (q_1, q_2, q_3, q_4)$ is $\{CX_2\}$. This is because the leading CX_1 can be run, but CX_2 blocks the rest of the gates. If we choose the SWAP gate between

Algorithm 1 DP-based exact algorithm for MCGM

IN: A dependency graph $D = (V_D, E_D)$ with logical qubits B, a coupling graph $C = (V_C, E_C)$ with physical qubits Q

OUT: The minimum number of SWAP and Bridge gates in mapping of D to C

```
1: S ← {(l, K_0(l)) | layout l from B to Q}      // active states
2: f(l, K) ← 0 for all initial state (l, K) ∈ S
3: while True do
4:     S' ← ∅      // next active states
5:     for state (l, K) in S do
6:         for edge e in E_C do
7:             l' ← layout after swapping e of l
8:             K' ← blocking gates from K for l'
9:             if f(l', K') is not yet defined then
10:                f(l', K') ← 1 + f(l, K)
11:                S' ← S' ∪ {(l', K')}
12:            end if
13:            if K' = ∅ then return f(l', K')
14:        end for
15:        for CNOT gate g (acting on (b_i, b_j)) in K do
16:            if distance between l(b_i) and l(b_j) on C = 2 then
17:                l' ← l      // no change in layout
18:                K' ← blocking gates from K \ {g} for l
19:                Do the same procedure between line 9 and 13
20:            end if
21:        end for
22:    end for
23:    S ← S'
24: end while
```

b_2 and b_3, SWAP(b_2, b_3), (or equivalently SWAP(q_2, q_3) under l_0), it changes the state $(l_0, \{CX_2\})$ to another state $((b_1, b_2, b_3, b_4) \mapsto (q_1, q_3, q_2, q_4), \{CX_4\})$. Alternately, we can choose the Bridge gate from $q_3 = l_0(b_3)$ to $q_4 = l_0(b_4)$ via q_2 to run $CX_2 = \text{CNOT}(b_3, b_4)$, where for simplicity we denote the mapping of a logical qubit b_i to a physical qubit q_j under layout l_0 as $q_j = l_0(b_i)$. The transformation of CNOT into a Bridge gate changes the state $(l_0, \{CX_2\})$ to another state $(l_0, \emptyset)$. That is, the transformation does not change the layout but updates the blocking gates $\{CX_2\}$ to $\emptyset$. After CX_2 is run as the Bridge gate, all the remaining gates can be run under the same layout l_0. This example shows how Bridge gates are important to realize better mapping algorithms.

Let $f(l, K)$ denote the minimum number of additional SWAP and Bridge gates required to reach a state (l, K) starting from any initial state. Here an initial state is a state defined by $(l, K_0(l))$ for an initial layout l. The minimum number of SWAP and Bridge gates in a mapping is thus the minimum value of $f(l, \emptyset)$ for all possible ls. This can be computed as follows. The algorithm first sets 0 to f of the initial states then checks the states that can be reached with one SWAP or Bridge gate (and set 1 to fs of the states that have not yet seen), those reached with two SWAP and/or Bridge gates (and set 2 to the newly checked states), and so on. The algorithm terminates when it reaches a state whose set of blocking gates is empty. Note that each state is activated at most once because of the minimality of

Algorithm 2 Look-ahead heuristic algorithm for MCGM

1: C: a coupling graph, D: a dependency graph
2: Initialize blocking gates K as the sources in D and layout l as the given initial layout.
3: **loop**
4: Run the gates complying with C and update K.
5: **if** $K = \emptyset$ **then terminate**
6: Compute the swap score for each edge in C.
7: Let (q_s, q_t) be the edge with the highest swap score.
8: Let $(b_{s'}, b_{t'}) \mapsto (q_s, q_t)$ be the current mapping in l.
9: Let $S \subseteq K$ be CNOT gates whose acting qubits has distance two in C for l.
10: **if** S is not empty and the highest swap score < 1 **then**
11: Transform some $g \in S$ acting on (q_s, q_t) in l into Bridge gate between (q_s, q_t).
12: **continue**
13: **end if**
14: **if** the swap of the mapping at (q_s, q_t) in l decreases $\sum_{g \in K} SP(g, l)$ **then**
15: Swap the mapping, i.e., $(b_{s'}, b_{t'}) \mapsto (q_t, q_s)$.
16: **else**
17: Choose any $\hat{g} \in K$ and add swap gates that strictly decrease $c(l, \{\hat{g}\})$ until $\hat{g}$ is resolved.
18: **end if**
19: **end loop**

f. See Algorithm 1 for details. The optimal gate list $\hat{L}$ can be obtained by recording the gates resolved by each of the swaps that update f at line 10 and traversing from the last state in the opposite direction.

3.4.2 Heuristic Algorithm

Our *look-ahead* heuristic algorithm takes into account not only the blocking gates but also the other unresolved gates in the selection of a qubit pair to be swapped. The look-ahead algorithm is different from those based on the fixed-layer formulation [94, 97] in the sense that it does not require any definition of layers.

The idea of the look-ahead mechanism is as follows. With regards to a layout l, for each unresolved two-qubit gate g we can compute the length of the shortest path between the two qubits of g as the minimal number of SWAP gates required to apply g. Thus, the number of SWAP gates of the layout l is proportional to the total length of the shortest paths of all unresolved two-qubit gates. Slightly modifying the layout l with swapping (q_i, q_j), denoted as $\tilde{l}(q_i, q_j)$, may decrease the length of the shortest paths at some unresolved two-qubit gates; but it may also increase those at other unresolved gates. The look-ahead heuristic prefers modifying the layout so that the total length of the shortest paths is reduced.

The pseudocode of the look-ahead heuristic algorithm is shown in Algorithm 2, where the lines from 9 to 13 are for considering Bridge gates. Like the exact one, the heuristic algorithm also maintains a layout l and blocking gates K; however, unlike the exact one, we assume that an initial layout is given. The heuristic algorithm updates K at the beginning of each loop. If K is empty, it terminates. Otherwise, it selects a qubit pair to be swapped on the basis of its swap *score*, i.e., the difference between the sum of the length of the shortest paths before and after swapping the qubit pair. Note that the selected qubit pair

should be an edge of the coupling graph C. Let U be a subset of unresolved CNOT gates including the current K and $SP(g, l)$ be the length of the shortest path from the two qubits of g with regards to the layout l. The (swap) score of a qubit pair $(q_i, q_j) \in E_C$ with l and U is defined as follows:

$$score((q_i, q_j), l, U) = c(l, U) - c(\tilde{l}(q_i, q_j), U),$$
$$c(l, U) = \sum_{g \in U} \gamma(g)\, SP(g, l),$$

where $\tilde{l}(q_i, q_j)$ denotes the layout after swapping (q_i, q_j) from l and $\gamma(g)$ denotes the weight of the shortest path of the gate g. To prioritize swapping a qubit pair reducing shortest paths of blocking gates and other unresolved gates close to them, we let $\gamma(g) = 1$ for $g \in K$ and $\gamma(g) = \alpha^{d(K,g)}$ $(0 < \alpha < 1)$ for $g \notin K$, where $d(K, g)$ denotes the longest path length among the paths from $g' \in K$ to g. At each loop of Algorithm 2, SWAP gates are added at either line 15 or at line 17. The former always decreases the length of the shortest paths of blocking gates, while the latter always decreases the size of K. This guarantees that the algorithm terminates after a finite number of loops.

The algorithm can be extended to handle Bridge gates by defining the score for replacing a non-adjacent CNOT with a Bridge gate. Since a Bridge gate does not change the layout, we can define $score((q_i, q_j), l, U) = 1$ for any (q_i, q_j) such that $(q_i, q_j) = (l(b_{i'}), l(b_{j'}))$ for a CNOT gate acting on $(b_{i'}, b_{j'})$ and the distance between (q_i, q_j) is two. Such extension are depicted at the lines from 9 to 13 in Algorithm 2.

3.5 Experiment

We conducted two sets of experiments: comparing the effectiveness of our formulation against that of formulations with fewer commutation rules and one without Bridge gates, and evaluating the performance of our heuristic algorithm against those of state-of-the-art algorithms. Both were conducted on a laptop PC with an Intel Core i7-6820HQ 2.7 GHz and 16 GB memory.

3.5.1 Comparison to other Formulations

In the first set of experiments, we compared the optimal numbers of additional gates with our formulation to those with the other formulations; two formulations with fewer commutation rules, i.e. the fixed-layer and standard-DAG, and one without Bridge gates. We obtained the optimal numbers of additional gates by applying our exact algorithm discussed in Section 3.4.1. To obtain the optimal numbers for the formulations with fewer commutation rules, we added extra edges to the dependency graphs so that they represent the fixed layers and standard-DAG, and then we applied our exact algorithm to them. We used two sets of 10 random circuits with five or six qubits. Each circuit contained 100

Table 3.1: Comparison of averages of optimal numbers of additional SWAP and Bridge gates of our formulation (in the Proposed column) and those of fixed-layer and standard-DAG formulation for two sets of 10 random circuits with five or six qubits. The average numbers of Bridge gates are stated in $(\cdot)$. The No-Bridge column lists the optimal numbers of SWAP gates from the original formulation [92]. ($|B|$: Number of qubits used in circuit)

$\|B\|$	Coupling	Fixed-layer	Std-DAG	Proposed	No-Bridge
5	ibmqx4	9.4 (3.4)	8.9 (2.6)	7.2 (1.1)	7.4
5	LNN	22.1 (6.3)	21.8 (7.0)	19.5 (4.4)	21.2
6	2×3	11.8 (1.7)	11.7 (2.0)	10.1 (1.4)	10.7
6	LNN	28.2 (6.3)	27.5 (8.0)	23.7 (5.3)	26.4

gates in which each gate was either a R_z, H, or CNOT gate with probability of 25%, 25%, or 50%, respectively. Here H is a single-qubit gate called the Hadamard gate.

Table 3.1 compares additional gates for our formulation and the other formulations. Recall that each of SWAP and Bridge gate introduces three extra CNOT gates as in Fig. 3.2(a) and 3.2(b), so the total number of additional CNOT gates is exactly three times the number of additional SWAP and Bridge gates. By comparing our formulation (Proposed) with the standard-DAG formulation (Std-DAG), we observed that Proposed had fewer numbers of additional gates. We certainly succeeded in reducing the numbers of additional SWAP and Bridge gates from standard-DAG formulation, which is slightly better than fixed-layer formulation (Fixed-layer) as expected, for each of the coupling architectures. By comparing Proposed with the original formulation without Bridge gates (No-Bridge), we can see that Bridge gates indeed improve optimality of the formulation. It is not surprising that the consideration of Bridge gates works best in LNN coupling architecture, which have leaves in the coupling graph. Unfortunately, the runtime of the exact algorithm grows exponentially with the number of qubits and gates; while all runs for 5-qubit instances are within 1 minute, but those for 6-qubit instances are around 12 minutes.

We experimented with random circuits to see clearly the difference in formulations and coupling architectures simultaneously in Table 3.1. We also conducted another set of experiments with standard circuits on a fixed coupling constraint (see Appendix A.2 for the details), and confirmed that there exist some instances in which our formulation is strictly better than the other formulations.

3.5.2 Comparison with other Heuristic Algorithms for Larger Circuits

In the second set of experiments, we evaluated the heuristic algorithm against two state-of-the-art heuristic algorithms. One is a randomized heuristic algorithm (called QRAND) implemented in Qiskit, which is a Python software development kit for quantum programming [15]. The other is an A*-based heuristic search algorithm (called ZPW) proposed by Zulehner et al. [94]. Their C++ implementation is available to the public, including their test circuit data, which originated from the RevLib benchmarks [98]. From their data,

Table 3.2: Comparison of numbers of additional SWAP and Bridge gates in mappings with our proposed heuristic algorithm (Proposed), its original one without Bridge gates (No-Bridge), as well as QRAND and ZPW for circuits with 10 to 16 qubits from RevLib benchmark. The fewest (i.e. best) numbers are in bold. The compositions of the numbers of SWAP and Bridge gates of Proposed are listed in the (#SWAP+Bridge) column. The runtime of Proposed are listed in the Time column. ($|B|$: Number of qubits used in circuit)

Circuit name	$\|B\|$	#gates	QRAND	ZPW	No-Bridge	Proposed	(#SWAP+Bridge)	Time [s]
mini_alu_305	10	173	80	46	**40**	41	(28+13)	0.5
qft_10	10	200	82	40	**33**	**33**	(33+0)	1.9
sys6-v0_111	10	215	116	67	46	**34**	(22+12)	0.6
rd73_140	10	230	100	58	49	**34**	(23+11)	0.6
ising_model_10	10	480	18	14	**12**	**12**	(12+0)	0.7
rd73_252	10	5,321	2,054	1,541	1,212	**999**	(644+355)	20.1
sqn_258	10	10,223	4,060	2,867	2,254	**1,875**	(1,108+767)	57.5
sym9_148	10	21,504	8,001	5,907	4,456	**3,770**	(1,856+1,914)	237.5
max46_240	10	27,126	10,833	8,012	5,905	**4,932**	(2,573+2,359)	469.8
wim_266	11	986	385	269	225	**155**	(65+90)	4.8
dc1_220	11	1,914	721	548	446	**329**	(154+175)	6.4
z4_268	11	3,073	1,200	907	718	**600**	(364+236)	11.2
life_238	11	22,445	9,181	7,209	5,264	**4,539**	(2,638+1,901)	268.8
9symml_195	11	34,881	14,470	10,682	8,124	**6,630**	(3,909+2,721)	684.1
sym9_146	12	328	173	85	66	**51**	(33+18)	0.9
cm152a_212	12	1,221	423	341	269	**175**	(76+99)	4.3
sqrt8_260	12	3,009	1,263	956	728	**587**	(377+210)	11.8
cycle10_2_110	12	6,050	2,701	1,856	1,518	**1,192**	(725+467)	27.9
rd84_253	12	13,658	5,817	4,271	3,271	**2,784**	(1,777+1,007)	108.9
rd53_311	13	275	162	98	77	**68**	(53+15)	1.9
ising_model_13	13	633	28	28	**18**	**18**	(18+0)	1.1
squar5_261	13	1,993	797	605	**422**	446	(285+161)	6.0
radd_250	13	3,213	1,347	951	721	**623**	(382+241)	11.3
adr4_197	13	3,439	1,529	1,044	806	**675**	(422+253)	12.6
root_255	13	17,159	7,172	5,417	4,070	**3,516**	(2,160+1,356)	152.4
dist_223	13	38,046	16,548	11,930	9,326	**7,879**	(4,952+2,927)	809.2
0410184_169	14	211	88	76	**44**	45	(35+10)	0.6
sym6_316	14	270	123	70	65	**58**	(41+17)	0.8
cm42a_207	14	1,776	677	494	402	**321**	(207+114)	8.9
cm85a_209	14	11,414	4,906	3,598	2,694	**2,370**	(1,395+975)	75.8
clip_206	14	33,827	14,845	11,011	8,187	**6,855**	(4,185+2,670)	624.1
sao2_257	14	38,577	16,974	12,511	9,188	**7,479**	(4,691+2,788)	780.2
rd84_142	15	343	192	103	67	**56**	(39+17)	1.0
misex1_241	15	4,813	1,844	1,520	1,216	**1,067**	(602+465)	21.8
square_root_7	15	7,630	3,243	2,369	**1,504**	1,546	(1,393+153)	46.7
ham15_107	15	8,763	3,635	2,552	1,979	**1,685**	(980+705)	50.0
dc2_222	15	9,462	4,112	2,933	2,326	**1,798**	(1,099+699)	58.5
co14_215	15	17,936	8,423	6,566	4,318	**3,294**	(1,780+1,514)	183.9
cnt3-5_179	16	175	69	54	38	**35**	(23+12)	0.5
cnt3-5_180	16	485	183	124	98	**88**	(49+39)	3.0
qft_16	16	512	296	117	**82**	**82**	(82+0)	4.8
ising_model_16	16	786	20	24	**18**	**18**	(18+0)	1.3
inc_237	16	10,619	4,351	3,138	2,542	**2,098**	(1,198+900)	68.3
mlp4_245	16	18,852	8,104	6,212	4,547	**3,988**	(2,495+1,493)	179.7

we chose 44 circuits with #qubits ($|B|$) ≥ 10 and #gates $\leq 50,000$ for the experiment. For all the circuits, we computed the mappings to the ibmqx3 coupling architecture [99]. We implemented our algorithm on top of Qiskit 0.6. We set the parameter $\alpha = 0.5$ and restricted the g of $d(K, g)$ to the gates within 10 steps from K.

The proposed heuristic algorithm outperformed QRAND and ZPW for all instances. See Table 3.2 for all results. The numbers of additional SWAP and Bridge gates in the mappings with the heuristic algorithm (Proposed) decreased by 10.0–72.3% (Avg. 53.2%) and 10.9–49.8% (Avg. 34.6%) from those of QRAND and ZPW, respectively. Even without Bridge gates (No-Bridge), in the mappings with commutation rules the numbers of additional SWAP gates decreased by 10.0–72.3% (Avg. 45.5%) and 7.1–42.1% (Avg. 23.8%) from those of QRAND and ZPW, respectively. From the columns of Proposed and No-Bridge, we can confirm that Bridge gates can find better mappings in almost all cases except for a few small circuits. The numbers of additional gates decreased by 14.2% on average. This implies, in spite of larger search space due to the consideration of Bridge gates, the proposed heuristic algorithm can efficiently explore it to find better solutions. All runs of the heuristic algorithm were less than 15 minutes.

Since we needed to give the initial layout l_0 for our algorithm and QRAND, we used the trivial layout $l_0 : (b_1, \ldots, b_{|Q|}) \mapsto (q_1, \ldots, q_{|Q|})$. The number of additional gates may possibly be further reduced by finding a good initial layout for the heuristics. For our algorithm, we added a post-processing to remove leading useless SWAP gates by changing the given initial layout. The reverse traversal technique by Li et al. [96] can be applied instead of the post-processing, and to find a better initial layout.

3.6 Related Work

There have been many studies of quantum circuit mapping, e.g., those dealing with the mapping on 1D-chain (known as linear nearest neighbor (LNN)) topology [100, 101, 102, 103, 104, 105, 106], those on 2D-grid nearest neighbor topology [97, 107, 108, 109], and, like ours, those on the general topology [83, 93, 94, 95, 96, 110, 111]. However, to the best of our knowledge, those studies either did not fully consider the gate commutation rules or did not consider transformation rules other than SWAP gates. As shown in Section 3.2, by fixing the layers, previous studies, such as on mixed integer programming (MIP) [93], A* search [94], and dynamic programming (DP) [95], can fail to obtain optimal solutions.

To overcome this suboptimality problem, we propose a novel formulation taking into account gate commutation rules in combination with the concept *dependency graph* that extends the standard-DAG, as discussed in Section 3.4. The dependency graphs with fewer commutation rules than those in Fig. 3.5 were first considered by Matsuo et al. [100]. Venturelli et al. [83] considered commutation rules specific to their circuits of interest and proposed a solution based on a temporal planner; however, they did not provide systematic methods exploiting the rules, unlike our approaches. Recently, Li et al. [96]

independently proposed a heuristic algorithm very similar to ours. To be precise, although they do not explicitly consider commutation rules in their heuristic, it is easy to modify their algorithm to take into account our proposed dependency graph. However, like other mapping methods, they only use SWAP gates in mapping.

In addition, we consider Bridge gate as well as SWAP gate as a transformation rule used in mapping. There are only a few studies considering Bridge gate (aka, chain template) in qubit routing, e.g. [47, 95]. Although their main purpose was for the synthesis of quantum circuit from a unitary matrix, Shende et al. [47] hinted that quantum circuit mapping can be solved with only chain templates. Siraichi et al. [95] provided exact and heuristic algorithms using both SWAP and Bridge gates, but they reported the implementation of their heuristic did not perform well for non-synthetic circuits, which may be caused by not using the dependency graph.

3.7 Summary

We addressed the problem of mapping quantum circuits to noisy quantum computers, whose operations are limited by their coupling architecture, with as few additional gates as possible. Our proposed solution to the mappings is to use gate commutation rules and gate transformation rules in the form of SWAP and Bridge gates. Many previous studies did not consider such gate commutation rules or Bridge gates to reduce additional gates in the mappings. We developed exact and heuristic algorithms that take advantage of such rules. Comparing them with the state-of-the-art circuit mapping algorithms, we demonstrated that our proposed algorithms can find better mappings with fewer additional gates for the circuits of a commonly used benchmark dataset.

Chapter 4

Quantum Circuit Scheduling

4.1 Overview

Quantum compilers take a quantum circuit and generate a corresponding sequence of control instructions that are executable on the target hardware. For example, in the case of quantum computers using superconducting qubits, a quantum operation is compiled into several controls (e.g., a microwave pulse) for a certain period of time. In general, any given quantum operation has its own processing time and occupies its acting qubits for the duration as a computational resource. For this reason, scheduling, through which the execution start time of each quantum operation is determined without any overlap, is an essential task in quantum compilation. We call this task *quantum circuit scheduling*. In this chapter, we aim to minimize the overall execution time. In the context of scheduling tasks across multiple resources (qubits, in the case of circuit scheduling), the time between the start of the first task and the end of the final task across all resources is known as the makespan of the schedule. Schedule length, overall execution time, and makespan are used interchangeably in this work.

Most of the previous studies on quantum compilers have handled circuit scheduling within the context of its before and after tasks, e.g., qubit routing and control instruction mapping [81, 82, 83, 84]. However, in practice, decomposing an entire compilation job into independent tasks is becoming more common in the software architecture of quantum compilers, similar to that of classical compilers, e.g., [15]. Therefore, we focus on the following research question: How much can we optimize the resulting schedule in quantum circuit scheduling by itself?

In this chapter[1], we examine quantum circuit scheduling and analyze its theoretical properties and practical usefulness. Our main contributions are as follows.

- We show that quantum circuit scheduling obtains greater degrees of freedom for optimizing the resulting schedule by further considering the commutativity of particular quantum operations (Section 4.2.1).

[1]Most of the contents in this chapter was published in [112].

- We demonstrate that quantum circuit scheduling can be reduced to a special type of job-shop problem that has a disjunctive graph representation (Section 4.2.3) so that we can formulate quantum circuit scheduling as Constraint Programming and Mixed Integer Programming, which are common techniques for the job-shop problem or scheduling in general (Section 4.3).
- We demonstrate through experiments with two common benchmark sets that the consideration of commutativity in scheduling reduces the schedule length by up to 7.36% (Section 4.4).

4.2 Problem

4.2.1 Quantum Circuit Scheduling

We define quantum circuit scheduling as the problem of finding a *schedule* for a given quantum circuit. A quantum circuit is a sequence of quantum operations. Many of them are unitary operations called *gates*. Each of the gates has acting qubits and its own processing time. A quantum circuit is given as a sequence: e.g., $[H(1), CX(1,2), X(2)]$. Here, $H(1)$ denotes a Hadamard gate acting on qubit 1, $CX(1,2)$ denotes a Controlled-NOT (or CNOT) gate acting on control qubit 1 and target qubit 2, and $X(2)$ denotes a NOT gate acting on qubit 2. Quantum circuits are typically depicted in a circuit diagram, as shown in Fig. 4.1. For simplicity, we assume all of the operations have the same unit processing

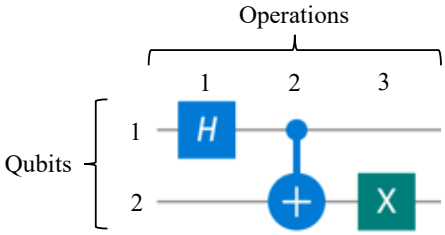


Figure 4.1: Diagram representation of a quantum circuit

time in Fig. 4.1. If we ignore commutation between gates, the gate dependency is linear, i.e., $H(1)$ must precede $CX(1,2)$ and $CX(1,2)$ must precede $X(2)$, and we obtain a trivial schedule whose length equals the sum of the three gates, as shown in the top of Fig. 4.2. We call the graph representing the dependencies among gates in a circuit the *dependency graph*. In contrast, if we consider that $CX(1,2)$ and $X(2)$ commute, we have a different dependency graph: $H(1)$ must precede $CX(1,2)$, but there is no restriction on $X(2)$, so we can obtain a shorter schedule, as shown in the bottom of Fig. 4.2. This is compelling evidence that commutation rules should be considered when scheduling circuit operations.

A schedule is defined by the start times of the operations in a given circuit. Any schedule must satisfy two elementary constraints: *precedence* and *non-overlap*. The precedence constraint restricts the execution order of operations to obey a partial order represented

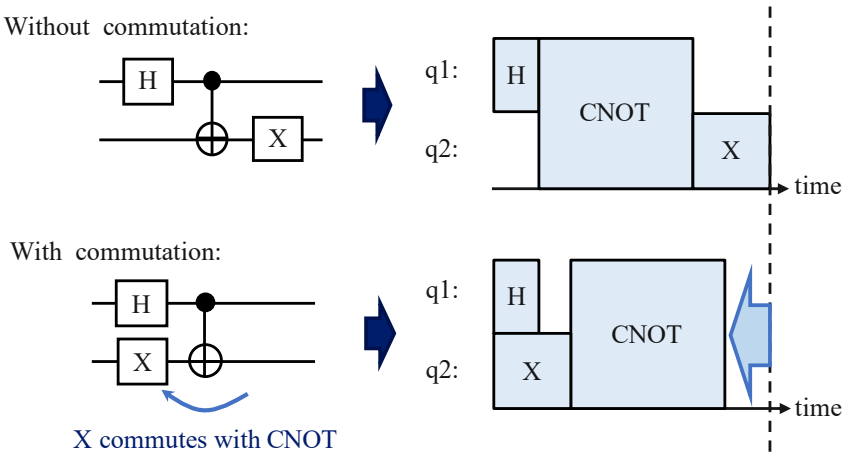


Figure 4.2: How gate commutation affects to quantum circuit scheduling

as a dependency graph. The non-overlap constraint allows only the processing of one operation on a qubit at a time.

Generally, the supported basic operations and the processing times depend on the target hardware. Hereafter, we assume that basic operations are given and that all circuits have already been decomposed into them. We also assume that each processing time of the basic operations is fixed and given as a parameter, while we allow it varies depending on qubits to which the operation applies. The dependency graph of a provided quantum circuit varies depending on which commutation rules are considered. Taking these details into account, we formally define a quantum circuit scheduling problem as follows.

Quantum Circuit Scheduling *Given a quantum circuit as a sequence of basic operations, each processing time of each operation, and a set of commutation rules between basic operations, find a schedule that satisfies precedence and non-overlap constraints with the minimum makespan.*

4.2.2 Job-shop Problem and its Disjunctive Graph Representation

We review a basic version of the job-shop problem as follows. Let $J = \{J_1, \ldots, J_n\}$ be a set of n jobs and $M = \{M_1, \ldots, M_m\}$ be a set of m machines. Each job J_j has an operation sequence O_j to be processed in a specific order, called the *precedence constraint*. We denote the k-th operation in O_j by O_{jk}. Each operation O_{jk} requires exclusive use of a specific machine for its processing time p_{jk}, called the *non-overlap constraint*. A schedule is a set of start (or completion) times for each operation t_{jk} that satisfies both constraints. The objective of the job-shop problem is minimization of the makespan.

The job-shop problem is often represented by a disjunctive graph $G = (V, C \cup D)$, where

- V is a set of nodes representing the operations O_{jk},
- C is a set of conjunctive (directed) edges representing the order of the operations in any job, and

- D is a set of disjunctive edges representing pairs of operations that must be processed on the same machine.

For each node, the processing time and the required machine of its corresponding operation is attached. Conjunctive edges C represent the precedence constraint and disjunctive edges D represent the non-overlap constraint. Note that disjunctive edges whose direction is fixed by some conjunctive edges can be omitted. That means any disjunctive edge can be removed if there exists a path from one end of the edge to the other on a conjunctive graph (V, C). Figure 4.3 shows an example of a disjunctive graph representing the job-shop problem. The operation O_{11} must be processed in machine M_1 and it takes 1 time unit. The disjunctive edge (O_{12}, O_{13}) is omitted since its direction is fixed by the conjunctive edge at the same place.

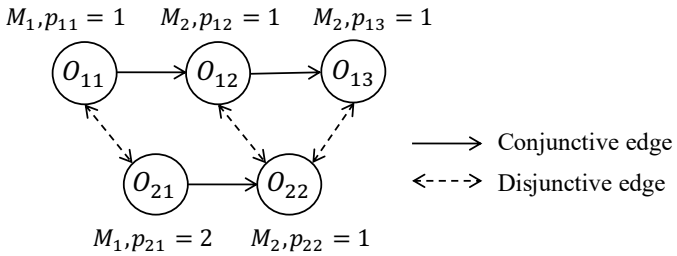


Figure 4.3: Disjunctive graph representation of a job-shop problem

On the basis of this disjunctive graph representation, the job-shop problem can be seen as a problem of determining the direction of disjunctive edges while keeping the resulting graph acyclic. This is equivalent to determining the ordering of the operations processed on the same machine, and such ordering yields a unique schedule, called a *semi-active schedule* [113], by sequencing operations as early as possible. Figure 4.4 shows two

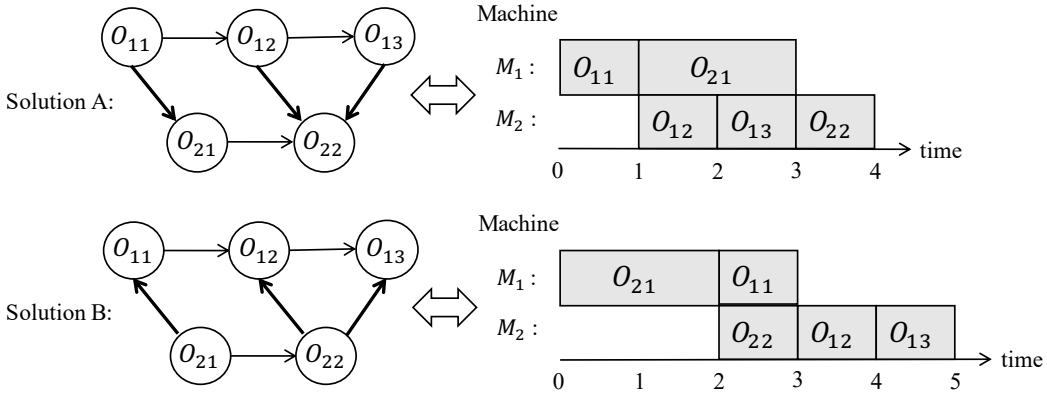


Figure 4.4: Two solutions to the job-shop problem in Fig. 4.3

solutions to the job-shop problem defined by the disjunctive graph depicted in Fig. 4.3. As shown, a different selection of the direction of the disjunctive edges results in a different solution. Among the operations $\{O_{12}, O_{13}, O_{22}\}$, we cannot select $\{(O_{22}, O_{12}), (O_{13}, O_{22})\}$

because it produces a cycle $O_{12} \to O_{13} \to O_{22} \to O_{12}$. In Solution A, the directed edge (O_{11}, O_{21}) determines the order of operations processed on machine M_1, and $\{(O_{12}, O_{22}), (O_{13}, O_{22})\}$ determine that on machine M_2.

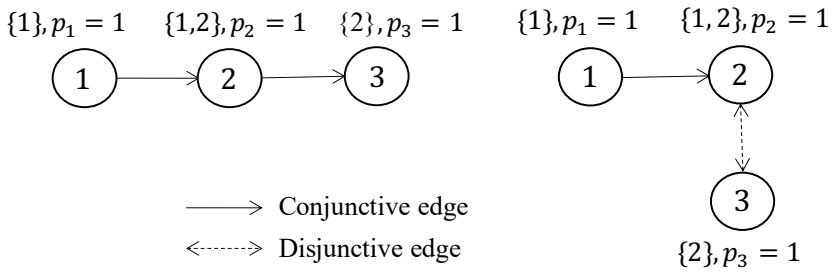
4.2.3 Disjunctive Graph Representation of Quantum Circuit Scheduling

Technically, quantum circuit scheduling can be seen as a special type of job-shop problem with the following properties: (1) one job has one operation, (2) a precedence constraint is given as a partial ordering among all operations instead of total ordering of operations per job, and (3) multiple machines (i.e., qubits) can be occupied by a single operation at the same time. Those properties preserve enough conditions to represent the problem by a disjunctive graph $G = (V, C \cup D)$. For property (1), we need to define the problem on operations without jobs, but this does not change the fact that nodes V represent operations. For properties (2) and (3), we need to modify the definition of conjunctive edges C and disjunctive edges D, respectively as follows.

The circuit scheduling can be represented by a disjunctive graph $G = (V, C \cup D)$, where

- V is a set of nodes representing the quantum operations in a given circuit,
- C is a set of conjunctive edges representing dependencies among the operations, i.e., edges of the dependency graph, and
- D is a set of disjunctive edges representing pairs of operations that act on the same qubit and (possibly) commute with one another.

For each operation (i.e., node) $i \in V$, the processing time p_i and the acting qubits are attached. Note that conjunctive graph (V, C) is a directed acyclic graph (DAG) given as a dependency graph. Conjunctive edges C and disjunctive edges D still represent the precedence constraint and non-overlap constraint, respectively. It is known that the dependency graph for any circuit can be computationally constructed under several popular commutation rules [91]. When provided with the dependency graph of a quantum circuit, the disjunctive graph representation can be computationally constructed. In fact, it is possible to define disjunctive edges by all the pairs of nodes that acting on the same qubit. This definition is redundant, but there is no problem with including edges (pairs of operations) that do not commute with one another. The point is that it must include all of the commuting pairs. Starting from the redundant edges, we can define the minimal disjunctive edges by removing edges that have a path on the conjunctive graph. However, this comes at a significant computation cost. There is an in-between definition that picks up operations acting on any qubit and splits them into sets of operations commuting each other within each of the sets. We used this definition for the experiments discussed in Section 4.4. Although this cannot provide the minimal edges because the operations within a set may not commute each other when considering operations acting on the other qubits,



(a) Standard DAG (b) Extended DAG

Figure 4.5: Disjunctive graphs representing a quantum circuit scheduling problem for different dependency graphs

it has fewer edges than the most redundant definition and requires less computation cost than the minimal definition.

With the disjunctive graph representation of the circuit scheduling, we take the commutation of operations into account on the basis of the difference of the dependency graph. We call the dependency graph that considers only the trivial commutation between operations not sharing their acting qubits *standard DAG* and the dependency graph that considers commutation rules in addition to the trivial ones *extended DAG*.

Figure 4.5 shows disjunctive graphs representing two circuit scheduling problems that have the same operations but different dependency (conjunctive) graphs (standard DAG and extended DAG). In the case of extended DAG, we can select the order of O_2 and O_3, while in standard DAG, all the orders among operations are fixed.

It generally holds that if we consider a standard DAG, there are no disjunctive edges, i.e., $D = \emptyset$. This means there is a unique semi-active schedule because each ordering of the operations processed on the same qubit is uniquely determined by the conjunctive DAG. However, if we consider an extended DAG, we have fewer edges in C and some edges in D. This creates the room for selecting a better ordering of the operations processed on the same qubit, i.e., optimizing the schedule.

4.3 Formulation

We provide a Constraint Programming (CP) formulation and a Mixed Integer Programming (MIP) formulation for the circuit scheduling problem defined in the previous section. By solving them, we can find the optimal solution of circuit scheduling and analyze how much we can improve the resulting schedule. In this section, we assume that the disjunctive graph representation of circuit scheduling $G = (V, C \cup D)$ for a given circuit has already been constructed.

4.3.1 Constraint Programming Formulation

Let x_i be an interval variable describing the start and end time of operation $i \in V$, whose duration is fixed to its processing time p_i. Using functions commonly supported by CP solvers, e.g. **interval_var**, quantum circuit scheduling is formulated as a CP:

$$\begin{array}{ll} \text{minimize} & \max\{\textbf{end\_of}(x_i) \mid \forall i \in V\} \\ \text{subject to} & \textbf{end\_before\_start}(x_i, x_j),\ \forall (i,j) \in C, \\ & \textbf{no\_overlap}(x_k, x_l),\ \forall (k,l) \in D, \\ & x_i \equiv \textbf{interval\_var}(\text{duration} = p_i),\ \forall i \in V. \end{array}$$

Here **end_of**(x_i) takes the end time of x_i, so the objective is the minimization of the makespan. The constraint **end_before_start**(x_i, x_j) means the end time of x_i must precede the start time x_j, so it represents the precedence constraint. The constraint **no_overlap**(x_k, x_l) means the interval x_k must not overlap the interval x_l, so it represents the non-overlap constraint. Note that, for any operation pair $(i,j) \notin D$, the precedence constraint guarantees no overlap between them. All of **end_of**, **end_before_start**, **no_overlap**, and **interval_var** are supported in the IBM ILOG CP Optimizer.

4.3.2 Mixed Integer Programming Formulation

Let x_i be a variable representing the start time of operation $i \in V$ and p_i be a constant parameter representing its processing time. Let t be a makespan of the schedule defined by x. Let y_{kl} be an indicator (Boolean) variable that takes True (1) if operation k precedes operation l and False (0) if l precedes k. Using these, quantum circuit scheduling is formulated as a MIP:

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & x_i + p_i \leq x_j,\ \forall (i,j) \in C, \\ & y_{kl} \Rightarrow x_k + p_k \leq x_l,\ \forall (k,l) \in D, \\ & \neg y_{kl} \Rightarrow x_l + p_l \leq x_k,\ \forall (k,l) \in D, \\ & x_i + p_i \leq t,\ \forall i \in V, \\ & 0 \leq x_i \in \mathbb{R},\ \forall i \in V, \\ & y_{kl} \in \{0,1\},\ \forall (k,l) \in D. \end{array}$$

The inequality $x_i + p_i \leq x_j$ represents the precedence constraint. The two inequalities using $\Rightarrow$ represent the non-overlap constraint. Note that the constraints with indicator variables y can be translated into linear constraints by applying the so-called the big-M technique. However, recent MIP solvers have the capability to handle such indicator constraints very well, so we leave the formulation with indicator variables.

4.4 Experiment

We conducted two experiments. In the first one, we evaluate how much the consideration of commutation between operations improves the schedule in circuit scheduling. In the second one, we investigate how much the extent of improvement in scheduling can be affected by the optimization level in a previous task.

4.4.1 Common Experimental Settings

Both of the experiments were conducted in a real compiling environment. As the target quantum computing device for compilation, we used the ibmq_johannesburg, which has 20 qubits (see [114] for the details). We implemented our scheduling algorithms within Qiskit 0.18.0 (Terra 0.13.0), which is an open-source quantum computing software development framework [15].

We first transpiled all of the circuits to make them executable on the ibmq_johannesburg, i.e., we solved the circuit mapping problem to map given circuits onto the device topology. For this, we used the "transpile()" function in Qiskit and set the ibmq_johannesburg backend, fixed the seed_transpiler to 1, the optimization_level to 2, and left the other options as the default. During transpiling, all of the circuits were decomposed into the basis gates $\{u1, u2, u3, CX\}$, which are elementary gates supported by the backend. Here $u1, u2, u3$ are single-qubit gates and CX is a two-qubit gate. The execution time for each gate (gate length) is provided as the backend properties. Note that it can differ depending on which qubit(s) the gate acts on, e.g. the length of $u3(1)$ can be different from that of $u3(2)$. We used those as of April 11, 2020.

We then applied our scheduling algorithms to the transpiled circuits. We used the real processing time for each of the basis gates provided as backend properties. We considered three commutation rules on the basis gates—$u1(i) \leftrightarrow CX(i, j)$, $CX(i, j) \leftrightarrow CX(i, k)$, and $CX(i, k) \leftrightarrow CX(j, k)$—in the construction of the extended DAGs for the transpiled circuits.

We used the IBM ILOG CP Optimizer and CPLEX 12.9.0 to solve the scheduling problem based on the CP and MIP formulation described in Section 4.3, respectively.

4.4.2 Improvement by Considering Gate Commutation

In the first experiment, we quantified the significance of considering the commutation of operations in circuit scheduling. Specifically, we evaluated the improvement by comparing the best solutions (makespans) of the formulation constrained by standard DAG with those by extended DAG as shown in Table 4.1.

For this experiment, we used quantum circuits from the test dataset provided by Zulehner et al. [94], which originated from the RevLib benchmark [98]. We selected 16 circuits with 10–16 qubits and less than 1000 gates from among them. We used the as-soon-as-possible heuristic scheduling algorithm implemented in Qiskit to find the unique solutions from the standard DAG formulation (Std-DAG). We also used the CP solver with

Table 4.1: Comparison of makespans [dt] ($1\,dt = 2/9\,\mathrm{ns}$) obtained by the formulation based on standard DAG (Std-DAG) and those based on extended DAG (Ext-DAG) for 16 circuits from the RevLib benchmark. For the Ext-DAG formulation, the solutions by the Constraint Programming solver with the time limit of ten seconds are listed. The Qubits and Gates columns list the number of qubits and gates in the input circuits. The Δ column lists the improvement rate from Std-DAG to Ext-DAG.

Circuit name	Qubits	Gates	Std-DAG	Ext-DAG	Δ
mini_alu_305	10	173	24,940	24,308	2.53%
qft_10	10	200	24,358	24,074	1.17%
sys6-v0_111	10	215	30,604	30,114	1.60%
rd73_140	10	230	35,848	35,484	1.02%
ising_model_10	10	480	4,210	4,210	0.00%
wim_266	11	986	141,914	138,658	2.29%
sym9_146	12	328	50,458	50,170	0.57%
rd53_311	13	275	40,420	40,164	0.63%
ising_model_13	13	633	4,210	4,210	0.00%
0410184_169	14	211	40,356	39,074	3.18%
sym6_316	14	270	47,178	46,404	1.64%
rd84_142	15	343	39,812	38,490	3.32%
cnt3-5_179	16	175	19,630	19,366	1.34%
cnt3-5_180	16	485	69,854	68,326	2.19%
qft_16	16	512	50,674	50,088	1.16%
ising_model_16	16	786	4,370	4,370	0.00%

a 10-sec time limit to find the best possible solutions from the extended DAG formulation (Ext-DAG).

Looking at the Δ column in Table 4.1, i.e., the improvement rates from Std-DAG to Ext-DAG, we can see they were non-negative and varied depending on the circuit structures from 0.00% to 3.32% (median 1.26%). These results demonstrate that the commutation-aware formulation we proposed in Section 4.2.1 can improve the resulting schedule length in a practical situation. Although they may look marginal, they may be welcomed by those who have abundant time for compilation and need more optimization.

We also examined the MIP solver (with the same 10-sec time limit) to find solutions of the Ext-DAG; however, all of the solutions were slightly worse or equal to those by the CP solver. Hence, we omitted these results in Table 4.1. As for the solutions (i.e., makespans) from the standard DAG formulation, we verified that those by the CP and MIP solvers were exactly the same as those by the as-soon-as-possible heuristic scheduling algorithm implemented in Qiskit as expected.

4.4.3 Performance Variation by Optimization Level of Previous Task

In the second experiment, we investigated how a previous task affects the solution quality in the circuit scheduling task. To this end, as a previous task, we picked the gate decomposition task that decomposes gates with three or more qubits into those with one or two qubits.

We changed the optimization level in the gate decomposition task and observed how it affects the improvement rates of makespans from scheduling with standard DAG (Std-DAG) compared to those with extended DAG (Ext-DAG), as shown in Table 4.2.

Table 4.2: Difference in improvement rates (Δ column) of makespans from scheduling with standard DAG (Std-DAG) compared to those with extended DAG (Ext-DAG) using CP solver after applying a naive gate decomposition or optimized gate decomposition.

Circuit name	Naive gate decomposition			Optimized gate decomposition		
	Std-DAG	Ext-DAG	Δ	Std-DAG	Ext-DAG	Δ
Mod 5_4	6,328	5,984	5.44%	6,016	5,578	7.28%
VBE-Adder_3	19,126	18,852	1.43%	11,416	11,148	2.35%
CSLA-MUX_3	22,238	21,438	3.60%	19,774	18,778	5.04%
RC-Adder_6	28,606	27,564	3.64%	20,408	18,906	7.36%
Mod-Red_21	33,160	32,348	2.45%	28,320	27,494	2.92%
Mod-Mult_55	13,942	13,740	1.45%	14,478	14,070	2.82%
Toff-Barenco_3	12,388	12,388	0.00%	4,812	4,812	0.00%
Toff-NC_3	9,108	9,108	0.00%	3,818	3,818	0.00%
Toff-Barenco_4	15,992	15,582	2.56%	9,006	8,678	3.64%
Toff-NC_4	12,272	11,834	3.57%	8,016	7,626	4.87%
Toff-Barenco_5	21,600	21,378	1.03%	19,424	19,138	1.47%
Toff-NC_5	13,536	12,984	4.08%	9,262	8,920	3.69%
Toff-Barenco_10	90,940	89,248	1.86%	59,282	57,282	3.37%
Toff-NC_10	43,156	42,546	1.41%	28,720	28,322	1.39%
GF(2^4)-Mult	33,160	32,492	2.01%	33,646	33,148	1.48%
GF(2^5)-Mult	37,948	36,480	3.87%	44,702	44,030	1.50%
GF(2^6)-Mult	63,856	61,870	3.11%	64,784	64,180	0.93%

For this experiment, we used 17 circuits from the test dataset provided by Nam et al. [71]. To change the optimization level in the gate decomposition task, we used both their input circuit data (with "_before" suffix in their file names) and output circuit data after the heavy optimization proposed in [71] (with "_after_heavy" suffix) as our input circuits to be scheduled. Those correspond with the Naive gate decomposition column and Optimized gate decomposition column, respectively. Note that, in the Naive case, gates are decomposed by a simple rule-based algorithm implemented in Qiskit before scheduling. As in the previous experiment, we used the as-soon-as-possible heuristic scheduling algorithm implemented in Qiskit to find the unique solutions from the Std-DAG formulation and the CP solver with a 10-sec time limit to find the best possible solutions from the Ext-DAG formulation.

As shown in the two Δ columns in Table 4.2, we can observe clear improvement from Std-DAG to Ext-DAG no matter which gate decomposition algorithm we used before circuit scheduling: Min: 0.00%–Median: 2.45%–Max: 5.44% (Naive) and Min: 0.00%–Median: 2.82%–Max: 7.36% (Optimized). This again confirms that our commutation-aware formulation proposed in Section 4.2.1 can improve the resulting schedule length in a practical situation.

When we compare the improvement rates (Δ column) from scheduling after the naive gate decomposition with those after the optimized gate decomposition in Table 4.2, there are two key findings. First, they have a similar median: 2.45% (Naive) and 2.82% (Optimized). This suggests that, on average, our commutation-aware scheduling can stably improve the resulting schedule no matter how much circuits has been optimized in a previous task (at least in the gate optimization task). Second, the improvement rates for each individual circuit differs between Naive and Optimized. Specifically, they increase from Naive to Optimized for ten circuits and decrease for five circuits. This suggests that the optimization level of the previous task significantly affects the optimization gain in circuit scheduling.

Comparing the makespans in the Std- or Ext-DAG column under naive gate decomposition with those under optimized gate decomposition in Table 4.2, we can see that they decrease for 13 out of 17 circuits, as expected, but increase for four circuits. The latter four exceptional cases stem from negative interference among optimization tasks before scheduling, i.e., between gate decomposition and some tasks done within 'transpile()' in Qiskit, and they are not caused by any errors in the scheduling.

4.5 Related Work

The job-shop problem, also known as job shop scheduling, is a well known optimization problem in computer science and operations research, and many variations of it have been studied [115, 116]. See Section 4.2.2 for its definition. Algorithms to solve this problem include exact ones such as branch-and-bound based on a Mixed Integer Programming (MIP) formulation [117], heuristic ones such as shifting bottleneck [118], and meta-heuristic ones such as simulated annealing [119]. In this chapter, we mainly focus on the exact algorithm, as we want to determine the effect of optimizing quantum circuit scheduling.

Task scheduling, which is the scheduling of computational tasks on multiple classical processors, has been extensively studied [120, 121]. A special type of task scheduling, DAG (Directed Acyclic Graph) scheduling, which deals with heterogeneous processors [122, 123], is most similar to quantum circuit scheduling, but it differs in the way that resource constraints are handled. In DAG scheduling, every task can be executed on any processor with a different cost, i.e., the resource constraint is soft, while in quantum circuit scheduling, every quantum operation has fixed qubit operands that are not interchangeable, i.e., the resource constraint is hard.

Quantum circuit mapping is a task that transforms a given circuit into an equivalent circuit so that all two-qubit operations in it can be executed on limited pairs of qubits. Schedule length can be approximated by circuit depth, which is the schedule length when assuming all operations have the same unit processing time. Therefore, circuit mapping with the objective of minimizing the circuit depth [93] or two-qubit gate depth [110, 124] can be viewed as approximate circuit scheduling with circuit mapping. Although the algorithms for this may be applicable to scheduling without circuit mapping, they provide

only approximate solutions, not the exact ones to circuit scheduling.

There are several studies on scheduling specialized for quantum computers based on ion trap technology [75, 76, 77]. These works consider a combination of scheduling and mapping under a hardware structure model, called macroblocks, and propose heuristic algorithms to solve it.

Several studies have considered the commutation of quantum operations in scheduling [81, 82, 83, 84]. Venturelli et al. [83] examined the scheduling of quantum operations as a sub-problem of circuit mapping. They proposed an exact method using a temporal planner and showed it works well for QAOA circuits, which have many commuting gates. Although their method is applicable to circuit scheduling without circuit mapping, our methods discussed in Section 4.3 are simpler and perform sufficiently well for the specific scheduling problem considered in this chapter. Guerreschi and Park [81] proposed a two-step solution that decomposes the problem with qubit routing and solves circuit scheduling (without qubit routing) in the first step. They provide a list scheduling heuristic algorithm using upward ranking but not any exact algorithm for scheduling. Other studies have considered circuit scheduling as a subtask of circuit mapping [84] or control optimization [82]. While they provide practical heuristic algorithms for solving the task that includes scheduling, the exact algorithm for scheduling is not discussed.

4.6 Discussion

The basic version of the job-shop problem as a decision problem is known to be NP-complete [125]. Since quantum circuit scheduling is a special variant of the job-shop problem, as discussed in Section 4.2.3, it is not necessary for it to be NP-complete. Identifying the theoretical complexity of circuit scheduling would be an interesting avenue for future work.

Throughout this chapter, we have investigated how to minimize the overall execution time of the resulting schedule. Although this certainly contributes to obtaining computational results with higher fidelity, there should be more direct approaches that attempt to maximize the output fidelity by considering gate-dependent errors. In fact, such approaches have recently proposed in quantum circuit mapping [126, 127, 128]. Utilizing techniques like this for circuit scheduling is also left for future work.

The two formulations (CP/MIP) discussed in Section 4.3 are useful for the theoretical best case analysis because their solvers implement exact algorithms that can find the optimal solution in the long run. They may also be sufficient for certain practical applications, since CP/MIP solvers usually implement problem-agnostic heuristic algorithms to find the best possible solution within a limited time. However, for the use cases where the compilation time is too critical to use CP/MIP solvers, it is worth considering heuristic algorithms specialized for circuit scheduling. As an example, we provide a heuristic algorithm based on the Heterogeneous-Earliest-Finish-Time (HEFT) algorithm for task

scheduling in Appendix A.3. Such a heuristic algorithm can complement the CP/MIP-based approach.

4.7 Summary

We investigated quantum circuit scheduling, the problem of scheduling operations in a given circuit with the shortest total execution time. We demonstrated that quantum circuit scheduling can be interpreted as a special type of job-shop problem where we consider the commutation between quantum operations to make room for optimization. We provided a Constraint Programming formulation and showed through experiments with real circuits and a compiler that solving quantum circuit scheduling independently improved the schedule length by the modest rate up to 7.36%.

Chapter 5

Conclusion

Throughout this thesis, we have considered optimization problems arising in quantum compiler tasks for noisy superconducting quantum computers. We focused on two of them, quantum circuit mapping and quantum circuit scheduling, both of which are essential to improve the quality of computational results. We provided their formulations as classical optimization problems taking into account gate commutation, which had not been fully considered in previous studies. We also provided exact and heuristic algorithms for solving them. We examined how the consideration of gate commutation in our algorithms improves the quality of solutions by computational experiments in practical settings.

In Chapter 3, we investigated quantum circuit mapping that minimizes the number of additional CNOT gates. Our proposed solution is to use gate commutation rules and gate transformation rules in the form of SWAP and Bridge gates. We developed exact and heuristic algorithms that take advantage of such rules by employing dynamic programming technique and look-ahead scheme, respectively. Comparing them with the state-of-the-art circuit mapping algorithms, we demonstrated that our proposed algorithms can find dramatically better mappings with much fewer additional CNOT gates for the circuits of a commonly used benchmark dataset.

In Chapter 4, we investigated quantum circuit scheduling that minimizes the total execution time. We demonstrated that the problem can be considered as an optimization problem only when we consider gate commutation, and it can be interpreted as a special type of job-shop problem. We provided Constraint Programming and Mixed Integer Programming formulations as well as a heuristic algorithm based on the HEFT algorithm well known in classical task scheduling. We showed through experiments with real circuits and a compiler that solving quantum circuit scheduling independently improved the schedule length. We also showed that, on average, our commutation-aware scheduling can stably improve the resulting schedule no matter how much the circuits have been optimized in a previous task (at least in the circuit synthesis task).

Our future work on optimization in quantum compilers is described as follows.

- Comparing optimization of one large task with that of multiple small tasks.

In this thesis, we decomposed an entire large compilation task into as many small tasks as possible. For example, we split the quantum circuit mapping task into two smaller tasks (qubit assignment and qubit routing) in Chapter 3. We also dealt with the quantum circuit scheduling task independent of the circuit mapping task in Chapter 4. This approach, handling multiple small tasks, makes it relatively easy to design, implement and maintain efficient algorithms for optimizing the individual tasks. On the other hand, considering a large problem that optimizes multiple tasks at once would be better than considering small sub-problems that optimize each task independently in the case when we are able to find the optimal solution of the large problem. This suggests that it is worth considering whether we can develop any efficient algorithm for the large problem that outperforms the combination of algorithms for the small sub-problems.

- Designing objective functions that approximate fidelity better.
  Considering objective functions that reflect the extent of errors that occur in quantum computing devices during the computation of a given circuit would be another important future direction. Although CNOT gate counts or schedule length, as we considered in this thesis, approximates the real errors fairly well, neither of them is sufficient to account for the qubit coherence errors proportional to the schedule length (e.g., T1/T2 error) or the extra errors caused by the parallel execution of gates (e.g., cross-talk error). For quantum circuit mapping, several approaches have attempted to maximize the output fidelity by considering instruction-dependent errors [126, 127, 128] or cross-talk errors [129]. Here, *instruction* means the pair of an operation and its acting qubits. However, these are not fully successful yet, partially because of the temporal variation in the error rate of each instruction, as reported in [130]. Conversely, this direction would be worth pursuing all the more because it seems to converge into the fundamentally hard problem of how to construct a sufficiently accurate and efficiently computable error model of a target hardware.

- Taking quantum error correction into consideration.
  In this thesis, we focused on compilers for noisy quantum computers and did not address error correction. However, in order to achieve the fault-tolerant quantum computing, compilers must have embedded processes for quantum error correction. Such processes include encoding of a logical qubit onto physical qubits, transformation of operations into those on the codes, and detection and correction of errors. The details of those processes differ depending on the quantum error-correcting code used, and how many basis gates are required in the resulting circuit differs as well. Therefore, research on synthesis, mapping, or scheduling that takes error correction into account needs to be conducted for each error-correcting code. Although there have already been many studies on this topic, it is still an active area of research because of rapid advances in research on quantum error-correcting codes.

Progress in software research for quantum computers, including the above topics on compilers, as well as in hardware research, would accelerate the realization of fault-tolerant quantum computers in the future.

Appendix A

Appendix

A.1 Construction of dependency graph

We give the algorithm for constructing dependency graph in Algorithm 3. Because it is specifically tailored to the commutation rules shown in Fig. 3.5, the commutation rules are not stated as input to the algorithm.

Algorithm 3 Constructing dependency graph

IN: List of gates L in a given logical circuit
OUT: Dependency graph D

```
1: V_D ← {g | g ∈ L}      // node set of D
2: E_D ← ∅      // edge set of D
3: for all gate pair (g_i, g_j) such that i < j in L do
4:     for all common acting qubit b of g_i and g_j do
5:         S ← {s | symbol s between g_i and g_j on b}
6:         if S ⊈ {R_z, •} and S ⊈ {R_x, ⊕} then
7:             E_D ← E_D ∪ {(g_i, g_j)}
8:         end if
9:     end for
10: end for
11: return (V_D, E_D)
```

Note that the dependency graph obtained by the algorithm is redundant. If necessary, the minimal set of edges can be obtained by checking each of the edges in E_D and removing the edge (s, t) if there exists a path from s to t in the graph with reduced edge set excluding (s, t).

A.2 Experimental comparison of formulations with standard circuits

We give the results of experiments for comparing formulations using well-known standard circuits in Table A.1. We used 10 circuits with five qubits from the same dataset mentioned

Table A.1: Comparison of optimal numbers of additional SWAP and Bridge gates of our formulation (in the Proposed column) and those of fixed-layer and standard-DAG formulation for ten RebLib benchmark circuits with five qubits under ibmqx4 coupling architecture. The numbers of Bridge gates are stated in (·). The No-Bridge column lists the optimal numbers of SWAP gates from the original formulation [92].

Circuit name	#gates	Fixed-layer	Std-DAG	Proposed	No-Bridge
4mod7-v1_96	164	6 (0)	6 (0)	6 (0)	6
aj-e11_165	151	7 (2)	7 (2)	6 (1)	6
one-two-three-v0_98	146	6 (2)	6 (2)	6 (1)	6
one-two-three-v1_99	132	6 (4)	6 (4)	6 (4)	6
4_49_16	217	7 (0)	7 (0)	7 (1)	7
mod10_171	244	7 (0)	7 (0)	7 (0)	7
hwb4_49	233	8 (0)	8 (0)	8 (0)	8
one-two-three-v0_97	290	8 (1)	8 (1)	8 (1)	9
mini-alu_167	288	10 (0)	10 (0)	10 (0)	10
alu-v2_31	451	13 (4)	13 (4)	13 (4)	15

in Section 3.5.2, i.e. Zulehner et al.'s RevLib dataset [94]. For all the circuits, we computed the mappings to the ibmqx4 coupling architecture [131]. The other settings of experiments are same as those for the experiments in Section 3.5.1.

In contrast to the results using random circuits, the difference among formulations is not so large but we can see our formulation (Proposed) yields strictly better solution quality bound than two formulations with less commutation rules (Fixed-layer and Std-DAG) in aj-e11_165, and than one without Bridge gates (No-Bridge) in one-two-three-v0_97 and alu-v2_31.

A.3 HEFT algorithm for quantum circuit scheduling

We show how the Heterogeneous-Earliest-Finish-Time (HEFT) algorithm for task scheduling can be used for quantum circuit scheduling with a slight modification. The original HEFT algorithm is designed for scheduling with a soft resource constraint, i.e., every operation can be executed on any processor with a different cost. We adjust it here so that it can work with a hard resource constraint, i.e., every operation has fixed qubit operands that are not interchangeable.

The original HEFT algorithm consists of two phases: an *operation prioritizing phase* for computing the priorities of all operations based on upward ranking and a *processor selection phase* for scheduling the highest priority operation at the moment on the processor, which minimizes the operation's finish time [120]. In the processor selection phase, the algorithm considers the possibility of inserting an operation in the earliest idle time-slot between two already scheduled operations. Only the idle time-slots that preserve precedence constraints, i.e., that comply with the dependency graph, are considered in this phase. This *insertion-based policy* allowing the insertion in the idle time-slot characterizes the HEFT algorithm.

While keeping this insertion-based policy, we adjust the HEFT algorithm so that every operation is assigned to the fixed qubits (i.e., processors in the original term), which means we no longer need to select qubits in the processor selection phase. Note that it is necessary for the adjusted HEFT algorithm to maintain scheduled time-slots across qubits, whereas the original algorithm simply maintains the time-slots by processors. The process flow of the HEFT algorithm for quantum circuit scheduling is shown in Algorithm 4.

Algorithm 4 HEFT algorithm for quantum circuit scheduling

1: $G = (V, C)$: dependency graph of a quantum circuit scheduling problem
2: Compute upward rank $r(u)$ for each operation $u \in V$ by

$$r(u) = d(u) + \max_{v \in succ(u)} r(v)$$

where $succ(u)$ is the set of immediate successors of u, $d(u)$ is the duration of u, and $r(e) = d(e)$ for any exit operation e.
3: $ready\_time(u) = 0$ for all $u \in V$.
4: **for all** $u \in V$ in descending order of $r(u)$ **do**
5: Insert u at the start time t of the earliest idle time-slot (whose duration $> d(u)$) after $ready\_time(u)$.
6: **for all** $v \in succ(u)$ **do**
7: $ready\_time(v) = \max(ready\_time(v), t + d(u))$.
8: **end for**
9: **end for**

We conducted experiments to check the solution quality of the adjusted HEFT algorithm with the same benchmark sets and experimental settings as used in Section 4.4. For all of the instances under the formulation with extended DAG (Ext-DAG), the HEFT algorithm always succeeded in finding solutions slightly worse than or equal to those by the CP solver. The medians of improvement rates from Std-DAG to Ext-DAG with the HEFT algorithm (the CP solver) were 1.16% (1.26%) for circuits by Zulehner et al. [94] used in Table 4.1 and 1.67% (2.45%) and 2.35% (2.82%) for circuits by Nam et al. [71] used in Table 4.2 with naive and optimized gate decomposition, respectively. This suggests that the HEFT algorithm is a good option for adding a bit more optimization in cases where not much compilation time is available.

All the results of these latter two experiments are listed in Table A.2. We can see a negative improvement at Toff-arenco_3 using optimized gate decomposition and the HEFT algorithm. This can happen because considering further commutation in the formulation with Ext-DAG yields a broader search space for algorithms, and it provides an opportunity to find not only a better solution than Std-DAG but also a worse one. However, this is not a big issue in practice in cases where we can afford to select the better of the solution by the as-soon-as-possible algorithm with Std-DAG and that by the HEFT algorithm with Ext-DAG.

Table A.2: Makespans and their improvement rates (Δ) from scheduling with standard DAG (Std-DAG) compared to those with extended DAG (Ext-DAG) using HEFT algorithm or CP solver after applying a naive gate decomposition or optimized gate decomposition.

Circuit name	Naive gate decomposition			Optimized gate decomposition		
	Std-DAG	Ext-DAG		Std-DAG	Ext-DAG	
		HEFT (Δ)	CP (Δ)		HEFT (Δ)	CP (Δ)
Mod 5_4	6,328	6,076 (3.98%)	5,984 (5.44%)	6,016	5,670 (5.75%)	5,578 (7.28%)
VBE-Adder_3	19,126	18,972 (0.81%)	18,852 (1.43%)	11,416	11,148 (2.35%)	11,148 (2.35%)
CSLA-MUX_3	22,238	21,444 (3.57%)	21,438 (3.60%)	19,774	18,914 (4.35%)	18,778 (5.04%)
RC-Adder_6	28,606	27,584 (3.57%)	27,564 (3.64%)	20,408	19,136 (6.23%)	18,906 (7.36%)
Mod-Red_21	33,160	32,640 (1.57%)	32,348 (2.45%)	28,320	27,570 (2.65%)	27,494 (2.92%)
Mod-Mult_55	13,942	13,822 (0.86%)	13,740 (1.45%)	14,478	14,270 (1.44%)	14,070 (2.82%)
Toff-Barenco_3	12,388	12,388 (0.00%)	12,388 (0.00%)	4,812	4,998 (−3.87%)	4,812 (0.00%)
Toff-NC_3	9,108	9,108 (0.00%)	9,108 (0.00%)	3,818	3,818 (0.00%)	3,818 (0.00%)
Toff-Barenco_4	15,992	15,786 (1.29%)	15,582 (2.56%)	9,006	8,678 (3.64%)	8,678 (3.64%)
Toff-NC_4	12,272	11,834 (3.57%)	11,834 (3.57%)	8,016	7,698 (3.97%)	7,626 (4.87%)
Toff-Barenco_5	21,600	21,388 (0.98%)	21,378 (1.03%)	19,424	19,220 (1.05%)	19,138 (1.47%)
Toff-NC_5	13,536	13,002 (3.95%)	12,984 (4.08%)	9,262	8,920 (3.69%)	8,920 (3.69%)
Toff-Barenco_10	90,940	89,418 (1.67%)	89,248 (1.86%)	59,282	57,678 (2.71%)	57,282 (3.37%)
Toff-NC_10	43,156	42,600 (1.29%)	42,546 (1.41%)	28,720	28,494 (0.79%)	28,322 (1.39%)
GF(2^4)-Mult	33,160	32,508 (1.97%)	32,492 (2.01%)	33,646	33,160 (1.44%)	33,148 (1.48%)
GF(2^5)-Mult	37,948	36,734 (3.20%)	36,480 (3.87%)	44,702	44,464 (0.53%)	44,030 (1.50%)
GF(2^6)-Mult	63,856	62,174 (2.63%)	61,870 (3.11%)	64,784	64,278 (0.78%)	64,180 (0.93%)

Bibliography

[1] Richard P. Feynman. Simulating physics with computers. *Int. J. Theor. Phys*, 21(6/7), 1982.

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

[3] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC96)*, pages 212–219, 1996.

[5] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15):150502+, 2009.

[6] Stephen Jordan. Quantum algorithm zoo. https://quantumalgorithmzoo.org/ (Accessed: 2020-11-07).

[7] IBM Quantum. IBM Quantum Experience. https://quantum-computing.ibm.com/ (accessed 2020-11-27).

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

[9] Wikipedia. Quantum computing: Physical realizations. https://en.wikipedia.org/wiki/Quantum_computing#Physical_realizations (Accessed: 2020-11-27).

[10] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. Experimental implementation of fast quantum searching. *Physical review letters*, 80(15):3408, 1998.

[11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[12] Microsoft. Azure Quantum. https://azure.microsoft.com/services/quantum/ (Accessed: 2020-11-27).

[13] Amazon. Braket. https://aws.amazon.com/braket/ (Accessed: 2020-11-27).

[14] Quantum AI team and collaborators. Cirq, October 2020. https://github.com/quantumlib/Cirq (Accessed: 2020-11-07).

[15] Qiskit. Qiskit: An open-source framework for quantum computing, 2019. https://www.qiskit.org/ (Accessed: 2020-11-07).

[16] Rigetti. Forest SDK. https://pyquil-docs.rigetti.com/en/stable/ (Accessed: 2020-11-07).

[17] Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. Toward an architecture for quantum programming. *The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics*, 25(2):181–200, 2003.

[18] Krysta M. Svore, Alfred V. Aho, Andrew W. Cross, Isaac Chuang, and Igor L. Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83, 2006.

[19] N. Cody Jones, Rodney Van Meter, Austin G. Fowler, Peter L. McMahon, Jungsang Kim, Thaddeus D. Ladd, and Yoshihisa Yamamoto. Layered architecture for quantum computing. *Physical Review X*, 2(3):031007, 2012.

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

[21] Thomas Häner, Damian S. Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018.

[22] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Abhari-Javadi, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 527–540. IEEE, 2019.

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

[24] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: A retargetable compiler for NISQ devices. *Quantum Science and Technology*, 2020.

[25] Microsoft. Quantum Development Kit. https://www.microsoft.com/en-us/quantum/development-kit (Accessed: 2020-11-07).

[26] Xanadu. Strawberry Fields. https://strawberryfields.ai/ (Accessed: 2020-11-07).

[27] Amazon. Braket SDK. https://github.com/aws/amazon-braket-sdk-python (Accessed: 2020-11-07).

[28] Alwin Walter Zulehner. *Design Automation for Quantum Computing*. PhD thesis, Universität Linz, 2019.

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

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

[31] Simon J. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581, 2006.

[32] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

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

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

[35] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[36] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[37] Andrew J. Landahl, Daniel S. Lobser, Benjamin C. A. Morrison, Kenneth M. Rudinger, Antonio E. Russo, Jay W. Van Der Wall, and Peter Maunz. Jaqal, the quantum assembly language for QSCOUT. *arXiv preprint arXiv:2003.09382*, 2020.

[38] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. Quantum programming languages. *Nature Reviews Physics*, pages 1–14, 2020.

[39] Wikipedia. Quantum programming. https://en.wikipedia.org/wiki/Quantum_programming (Accessed: 2020-11-07).

[40] Ken Matsumoto and Kazuyuki Amano. Representation of quantum circuits with Clifford and $\pi/8$ gates. *arXiv preprint arXiv:0806.3834*, 2008.

[41] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Asymptotically optimal approximation of single qubit unitaries by Clifford and T circuits using a constant number of ancillary qubits. *Physical review letters*, 110(19):190502, 2013.

[42] Neil J. Ross and Peter Selinger. Optimal ancilla-free Clifford+T approximation of z-rotations. *Quantum Information & Computation*, 16(11-12):901–953, 2016.

[43] Mehdi Saeedi, Mona Arabzadeh, Morteza Saheb Zamani, and Mehdi Sedighi. Block-based quantum-logic synthesis. *Quantum Information & Computation*, 11(3):262–277, 2011.

[44] Marc G. Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi, and Costin Iancu. Towards optimal topology aware quantum circuit synthesis. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 223–234. IEEE, 2020.

[45] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013.

[46] Luke E. Heyfron and Earl T. Campbell. An efficient quantum compiler that reduces T count. *Quantum Science and Technology*, 4(1):015004, 2018.

[47] Vivek V. Shende, Stephen S. Bullock, and Igor L. Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, 2006.

[48] M. Mottonen and Juha J. Vartiainen. Decompositions of general quantum gates. *arXiv preprint quant-ph/0504100*, 2005.

[49] Charles H. Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.

[50] Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages, and Programming*, pages 632–644. Springer, 1980.

[51] Mehdi Saeedi and Igor L. Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys (CSUR)*, 45(2):1–34, 2013.

[52] Dmitry V. Zakablukov. On asymptotic gate complexity and depth of reversible circuits without additional memory. *Journal of Computer and System Sciences*, 84:132–143, 2017.

[53] Dmitri Maslov. Optimal and asymptotically optimal NCT reversible circuits by the gate types. *Quantum Information & Computation*, 16(13-14):1096–1112, 2016.

[54] Sergey Bravyi, Theodore J. Yoder, and Dmitri Maslov. Efficient ancilla-free reversible and quantum circuits for the Hidden Weighted Bit function. *arXiv preprint arXiv:2007.05469*, 2020.

[55] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147, 1996.

[56] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184*, 2004.

[57] Thomas G. Draper. Addition on a quantum computer. *arXiv preprint quant-ph/0008033*, 2000.

[58] Thomas G. Draper, Samuel A. Kutin, Eric M. Rains, and Krysta M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Quantum Information & Computation*, 6(4):351–369, 2006.

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

[60] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Physical Review A*, 54(2):1034, 1996.

[61] Rodney Van Meter and Kohei M. Itoh. Fast quantum modular exponentiation. *Physical Review A*, 71(5):052320, 2005.

[62] Yasuhiro Takahashi and Noboru Kunihiro. A fast quantum circuit for addition with few qubits. *Quantum Information & Computation*, 8(6):636–649, 2008.

[63] Igor L. Markov and Mehdi Saeedi. Faster quantum number factoring via circuit synthesis. *Physical Review A*, 87(1):012310, 2013.

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

[65] Vivek V. Shende and Igor L. Markov. On the CNOT-cost of Toffoli gates. *Quantum Information & Computation*, 9(5):461–486, 2009.

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

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

[68] Martin Plesch and Časlav Brukner. Quantum-state preparation with universal gate decompositions. *Physical Review A*, 83(3):032302, 2011.

[69] Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home, and Matthias Christandl. Quantum circuits for isometries. *Physical Review A*, 93(3):032318, 2016.

[70] Dmitri Maslov, Gerhard W. Dueck, D. Michael Miller, and Camille Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008.

[71] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018.

[72] Frank Leymann and Johanna Barzen. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology*, 5(4):044007, 2020.

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

[74] Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *arXiv preprint arXiv:2002.09783*, 2020.

[75] Naser Mohammadzadeh, Morteza Saheb Zamani, and Mehdi Sedighi. Improving latency of quantum circuits by gate exchanging. In *Proceedings of 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 67–73. IEEE, 2009.

[76] Tayebeh Bahreini and Naser Mohammadzadeh. An MINLP model for scheduling and placement of quantum circuits with a heuristic solution approach. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 12(3):29, 2015.

[77] Xin-Chuan Wu, Dripto M. Debroy, Yongshan Ding, Jonathan M. Baker, Yuri Alexeev, Kenneth R. Brown, and Frederic T. Chong. TILT: Achieving higher fidelity on a trapped-ion linear-tape quantum computing architecture. *arXiv preprint arXiv:2010.15876*, 2020.

[78] Dmitri Maslov, Sean M. Falconer, and Michele Mosca. Quantum circuit placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):752–763, 2008.

[79] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. On the complexity of quantum circuit compilation. In *Eleventh Annual Symposium on Combinatorial Search*, 2018.

[80] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[81] Gian Giacomo Guerreschi and Jongsoo Park. Two-step approach to scheduling quantum circuits. *Quantum Science and Technology*, 3(4):045003, 2018.

[82] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1031–1044. ACM, 2019.

[83] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Temporal planning for compilation of quantum approximate optimization circuits. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 89–101, 2017.

[84] Tzvetan S. Metodi, Darshan D. Thaker, Andrew W. Cross, Frederic T. Chong, and Isaac L. Chuang. Scheduling physical operations in a quantum information processor. In *Quantum Information and Computation IV*, volume 6244, pages 210–221. International Society for Optics and Photonics, SPIE, 2006.

[85] Pranav Gokhale, Ali Javadi-Abhari, Nathan Earnest, Yunong Shi, and Frederic T. Chong. Optimized quantum compilation for near-term algorithms with openpulse. *arXiv preprint arXiv:2004.11205*, 2020.

[86] Petar Jurcevic, Ali Javadi-Abhari, Lev S. Bishop, Isaac Lauer, Daniela F. Bogorin, Markus Brink, Lauren Capelluto, Oktay Günlük, Toshinari Itoko, Naoki Kanazawa, et al. Demonstration of Quantum Volume 64 on a superconducting quantum computing system. *arXiv preprint arXiv:2008.08571*, 2020.

[87] Felix Motzoi, Jay M. Gambetta, Patrick Rebentrost, and Frank K. Wilhelm. Simple pulses for elimination of leakage in weakly nonlinear qubits. *Physical review letters*, 103(11):110501, 2009.

[88] Max Werninghaus, Daniel J. Egger, Federico Roy, Shai Machnes, Frank K. Wilhelm, and Stefan Filipp. Leakage reduction in fast superconducting qubit gates via optimal control. *arXiv preprint arXiv:2003.05952*, 2020.

[89] Andre R. R. Carvalho, Harrison Ball, Michael J. Biercuk, Michael R. Hush, and Felix Thomsen. Error-robust quantum logic optimization using a cloud quantum computer interface. *arXiv preprint arXiv:2010.08057*, 2020.

[90] Nicolas Wittler, Federico Roy, Kevin Pack, Max Werninghaus, Anurag Saha Roy, Daniel J. Egger, Stefan Filipp, Frank K. Wilhelm, and Shai Machnes. An integrated tool-set for control, calibration and characterization of quantum devices applied to superconducting qubits. *arXiv preprint arXiv:2009.09866*, 2020.

[91] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration*, 70:43–50, 2020.

[92] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. Quantum circuit compilers using gate commutation rules. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 191–196. ACM, January 2019.

[93] Debjyoti Bhattacharjee and Anupam Chattopadhyay. Depth-optimal quantum circuit placement for arbitrary topologies. *arXiv:1703.08540*, 2017.

[94] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD)*, 2018. Implementation with test data set is available at https://github.com/iic-jku/ibm_qx_mapping.

[95] Marcos Siraichi, Vinicius Fernandes Dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. Qubit allocation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–12, 2018.

[96] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014. ACM, April 2019.

[97] Robert Wille, Oliver Keszocze, Marcel Walter, Patrick Rohrs, Anupam Chattopadhyay, and Rolf Drechsler. Look-ahead schemes for nearest neighbor optimization of 1D and 2D quantum circuits. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 292–297. IEEE, 2016.

[98] Mathias Soeken, Stefan Frehse, Robert Wille, and Rolf Drechsler. RevKit: An open source toolkit for the design of reversible circuits. In *Proceedings of the International Workshop on Reversible Computation*, pages 64–76. Springer, 2011. RevLib is available at http://www.revlib.org.

[99] IBM Quantum. IBM Q 16 Rueschlikon V1.0.0 (ibmqx3). https://github.com/Qiskit/ibmq-device-information/blob/master/backends/rueschlikon/V1/version_log.md (Accessed: 2020-10-10).

[100] Atsushi Matsuo and Shigeru Yamashita. Changing the gate order for optimal LNN conversion. In *Proceedings of the International Workshop on Reversible Computation*, pages 89–101. Springer, 2011.

[101] Yuichi Hirata, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima. An efficient conversion of quantum circuits to a linear nearest neighbor architecture. *Quantum Information & Computation*, 11(1&2):142–166, 2011.

[102] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing*, 10(3):355–377, 2011.

[103] Amlan Chakrabarti, Susmita Sur-Kolay, and Ayan Chaudhury. Linear nearest neighbor synthesis of reversible circuits by graph partitioning. *arXiv:1112.0564*, 2011.

[104] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures. In *Proceedings of the 50th Design Automation Conference*, page 41. ACM, 2013.

[105] Robert Wille, Aaron Lye, and Rolf Drechsler. Optimal SWAP gate insertion for nearest neighbor quantum circuits. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 489–494. IEEE, 2014.

[106] Md. Mazder Rahman and Gerhard W. Dueck. Synthesis of linear nearest neighbor quantum circuits. *arXiv:1508.05430*, 2015.

[107] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. Qubit placement to minimize communication overhead in 2D quantum architectures. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 495–500. IEEE, 2014.

[108] Aaron Lye, Robert Wille, and Rolf Drechsler. Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 178–183. IEEE, 2015.

[109] Daniel Ruffinelli and Benjamín Barán. Linear nearest neighbor optimization in quantum circuits: a multiobjective perspective. *Quantum Information Processing*, 16(9):220, 2017.

[110] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography*, 2019.

[111] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Proceedings of the 56th Design Automation Conference*, page 142. ACM, 2019.

[112] Toshinari Itoko and Takashi Imamichi. Scheduling of operations in quantum compiler. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 337–344. IEEE, October 2020.

[113] Takeshi Yamada and Ryohei Nakano. Job-shop scheduling. In *Genetic algorithms in engineering systems*, chapter 7, pages 134–160. The Institution of Electrical Engineers, 1997.

[114] Antonio D. Córcoles, Abhinav Kandala, Ali Javadi-Abhari, Douglas T. McClure, Andrew W. Cross, Kristan Temme, Paul D. Nation, Matthias Steffen, and J. M. Gambetta. Challenges and opportunities of near-term quantum computing systems. *Proceedings of the IEEE*, 2019.

[115] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1–33, 1996.

[116] Jian Zhang, Guofu Ding, Yisheng Zou, Shengfeng Qin, and Jianlin Fu. Review of job shop scheduling research and its new perspectives under Industry 4.0. *Journal of Intelligent Manufacturing*, 30(4):1809–1830, 2019.

[117] Alan S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

[118] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[119] Peter J. M. van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992.

[120] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[121] Oliver Sinnen. *Task scheduling for parallel systems.* John Wiley & Sons, 2006.

[122] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of DAG scheduling heuristics. In *Grid Computing*, pages 73–84. 2008.

[123] Christos Valouxis, Christos Gogos, Panayiotis Alefragis, George Goulas, Nikolaos Voros, and Efthymios Housos. DAG scheduling using integer programming in

heterogeneous parallel execution environments. In *Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2013)*, pages 392–401, 2013.

[124] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. Circuit transformations for quantum architectures. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography*, 2019.

[125] Michael R. Garey and David S. Johnson. *Computers and intractability.* W. H. Freeman and Company, 1979.

[126] Swamit S. Tannu and Moinuddin K. Qureshi. Not all qubits are created equal: a case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 987–999, 2019.

[127] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1015–1029, 2019.

[128] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. Extracting success from IBM's 20-qubit machines using error-aware compilation. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(3):1–25, 2020.

[129] Prakash Murali, David C. McKay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1016, 2020.

[130] Ellis Wilson, Sudhakar Singh, and Frank Mueller. Just-in-time quantum circuit transpilation reduces noise. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 345–355, 2020.

[131] IBM Quantum. IBM Q 5 Tenerife V1.0.0 (ibmqx4). https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md (Accessed: 2020-10-10).

List of Publications

Journals

1. Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration*, 70:43–50, 2020.

International Conferences with Review

1. Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. Quantum circuit compilers using gate commutation rules. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASP-DAC2019)*, pages 191–196. ACM, January, 2019.

2. Toshinari Itoko and Takashi Imamichi. Scheduling of operations in quantum compiler. In *Proceedings of the International Conference on Quantum Computing and Engineering (QCE2020)*, pages 337–344. IEEE, October, 2020.

Unpublished Manuscripts

1. Petar Jurcevic, Ali Javadi-Abhari, Lev S. Bishop, Isaac Lauer, Daniela F. Bogorin, Markus Brink, Lauren Capelluto, Oktay Günlük, Toshinari Itoko, Naoki Kanazawa, et al. Demonstration of Quantum Volume 64 on a superconducting quantum computing system. *arXiv preprint arXiv:2008.08571*, 2020.