Towards Effective Qubit Mapping for Noisy Intermediate-Scale Quantum Devices

by

Chi Zhang

Bachelor of Engineering, Xidian University, China, 2014

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Chi Zhang

It was defended on

December 2nd 2021

and approved by

Dr. Youtao Zhang, Department of Computer Science, University of Pittsburgh

Dr. Shi-Kuo Chang, Department of Computer Science, University of Pittsburgh

Dr. Xulong Tang, Department of Computer Science, University of Pittsburgh

Dr. Wonsun Ahn, Department of Computer Science, University of Pittsburgh

Dr. Jun Yang, Department of Electrical and Computer Engineering, University of Pittsburgh

Towards Effective Qubit Mapping for
Noisy Intermediate-Scale Quantum Devices

Chi Zhang, PhD

University of Pittsburgh, 2021

Today, quantum devices are comprised of qubits ranging from dozens to hundreds in number. There is a class of problems that cannot be solved using classical computing. Whereas, it's expected that these tasks can be solved by the current and the future quantum devices achieving supremacy over classical computing. However, the current state-of-the-art quantum computing is riddled with qubit mapping problem. CNOT gates, that take two logical qubits as input, can only be mapped to a physical qubit pair, adjacent to each other. But the physical layout of the current quantum hardware is normally irregular. This renders executing quantum programs directly on current quantum devices infeasible. Moreover, the available quantum error correction is not adequate. This leads the state of quantum computing to the Noisy Intermediate-Scale Quantum (NISQ) era, yielding the need of smaller sized quantum circuits to be mapped onto the hardware. This requires thorough investigations into designing solutions to achieve effective mapping strategy between logical and physical qubits.

In this proposal, we are looking into four strategies to partially address the qubit mapping problem namely – (i) a depth-aware SWAP insertion scheme, (ii) a slack-based qubit mapper, (iii) the first time-optimal qubit mapping solution, and (iv) a crosstalk-aware decoherence-mitigating qubit mapping compilation framework.

Table of Contents

List of Tables

List of Figures

Preface

Many thanks to my family, my advisors and my friends.

1.0 Introduction

Quantum computing is a promising method to speed up important applications. These applications include factoring large numbers [22], searching a database [7], and simulating quantum systems [18]. With around 100 reliable qubits, quantum computers can already solve useful problems that are out of reach for classical computers.

Recently quantum systems with 49-72 qubits have been announced by IBM, Google and Intel. A number of quantum computers with around or less than 20 qubits are available to the public [2] through IBM Q experience. Programmers can run programs on these quantum computers. Despite the existence of certain tools such as IBM Qiskit [1], code must be written with respect to low-level specifications. A complete quantum compiler tool-chain needs to be built such that programmers can develop quantum algorithms that take full advantage of the potentially disruptive computing paradigm without having to worry about low level machine details.

The compilation of a quantum program is decomposed into two levels of translation. First, it converts an algorithm into a logical circuit composed of universal gates. These circuits are formulated independently of the hardware implementation. Second, it converts a logical circuit into a physical circuit with respect to hardware constraints. The problems in the first abstraction layer have been extensively [7, 18, 22] studied by theoreticians. However, less attention has been paid to the second abstraction layer, which is critical to the efficient execution of programs on real quantum computers.

This dissertation addresses the second level of translation: from a logical circuit to a hardware-compliant physical circuit. In particular, we tackle the qubit mapping problem. In realistic architecture, it is not possible to establish direct interactions between every pair of qubits. In the superconducting quantum computers, qubits operate in the nearest neighbor manner, in which direct interactions form a bounded degree graph. An example is shown in Fig. 1 (a).

However, a logical circuit independent of hardware implementation assumes an unrestricted architecture, where every two qubits are connected. A logical circuit must be modified to

account for the coupling constraints in quantum hardware. The common practice is to dynamically remap logical qubits to physical qubits via SWAP gates such that each two-qubit gate is applied to two physically connected qubits. An example is shown in Fig. 1. If two logical interacting qubits q_1 and q_4 are respectively mapped to Q_1 and Q_4, one of them has to be "moved" closer to the other. For instance, by swapping q_2 with q_4, we can move q_4 closer to q_1.

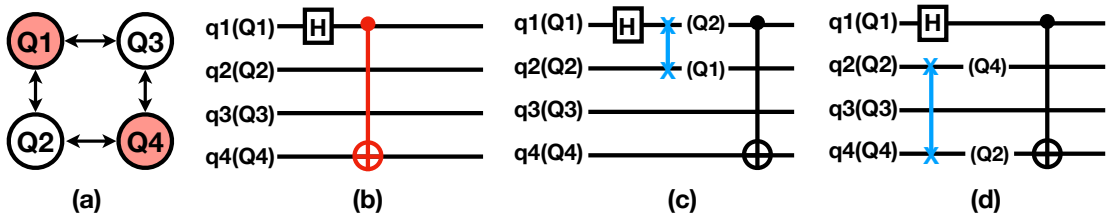

Figure 1: (a) Hardware coupling graph, (b) the logical circuit which cannot run due to qubit coupling constraints, (c) one possible way to make the circuit executable by swapping Q_1 with Q_2, and (d) another possible way to make the circuit executable by swapping Q_2 with Q_4. Upper case Q represents a physical qubit while lower case q represents a logical qubit.

1.1 Gate Optimality v.s. Time Optimality

Most previous studies on the qubit mapping problem [13, 24, 25, 28, 34, 35] have focused on *gate-optimal* solutions. They minimize the number of inserted SWAP gates. Some [13, 35] enhance the parallelism among the inserted SWAP gates while aiming to minimize the gate count. Zulehner *et al.* [35] proposed an algorithm using the A-star paradigm to minimize the number of swap gates for a local layer of concurrent CNOT gates. Li *et al.* [13] formulate a multi-objective function for exploiting the trade-off between different swap insertion strategies. The study by Siraichi *et al.* [24] models the swap-insertion problem as a subgraph isomorphism problem. Wille *et al.* [28] propose a model for global gate-optimal mapping using the SAT solver. A number of studies [16, 27] note the variability of qubit error rates in the IBM quantum computers and develop variability-aware qubit mapping strategies.

There are very few studies focusing on time-optimal qubit mapping. A time-optimal solution minimizes the depth of the entire transformed circuit rather than the depth or the number of the inserted SWAPs. OLSQ [26] is the only study that solves for optimal depth of the *entire* circuit. It is based on a constraint solver and its mapping overhead depends on the how far the optimal depth is from the ideal depth assuming every two qubits are connected. The work by Childs *et al.* [3] forms the foundation of IBM Qiskit's qubit mapper, however, it uses a heuristic approach to minimize the depth of the inserted SWAPs rather than that of the entire circuit. Lao *et al.* [12] introduces a framework that considers gate selection, qubit mapping, and classical control constraints. Their qubit mapping heuristic aims to improve the depth of the entire circuit by overlapping swaps with original gates in the circuit. But it does not guarantee an optimal mapping solution.

To see why it is important to consider the interaction between inserted swap operations and the original circuit, we use an example in Fig. 1. The original circuit in Fig. 1 (b) is not executable. The transformed circuits in (c) and (d) are both gate optimal. However, only one of them is time optimal. That is because the solution in Fig. 1 (c) inserts a swap involving the slowest qubit, and as a result delays the execution of the entire circuit, while the solution in Fig. 1 (d) does not.

A time-optimal solution not only reduces the execution time, but also improves the reliability of the transformed circuit. It mitigates the *decoherence* effect. Qubits are error prone. A qubit decoheres over time. It gradually lose its state information. The longer a qubit operates, the less reliable it is. A time-optimal solution minimizes the impact of decoherence for the qubits in the circuit, and results in higher fidelity of the circuit as a whole.

Time-optimal qubit mapping is in general challenging. There are many possible permutations of SWAPs to achieve the same desired qubit mapping and there are many possible qubit mappings that can satisfy two-qubit gates in a given circuit. Even properly modeling the problem is a challenge.

1.2 Crosstalk-Awareness

The states of qubits in NISQ devices, however, are quite volatile. A quantum system can be affected by various types of quantum errors including decoherence error, readout error, gate error, crosstalk error, among which the impact from crosstalk error has been reported to be significant. Crosstalk exists due to unwanted coupling between concurrently qubit operations, including both single-qubit and two-qubit gates. However, crosstalk for single-gate is negligible compared to two-qubit gates. Therefore, crosstalk commonly refers to the interference between concurrent two-qubit gates, which is also the focus of this work, shown in Figure 2.

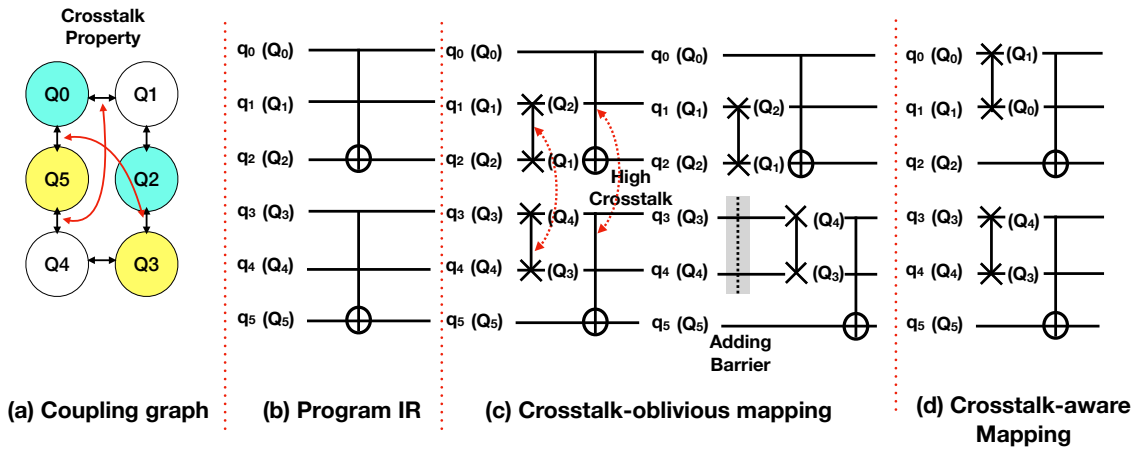

Figure 2: Crosstalk example: (a) Hardware coupling graph (b) Program IR, and initial qubit mapping; (c) Crosstalk-oblivious hardware mapping of the program IR, where at least one two-qubit gate needs to be delayed to avoid crosstalk; (d) Crosstalk-aware hardware mapping, where no delay of gate is necessary.

An example in Fig. 2 (a) shows two pairs of qubits (q_1, q_2) and (q_3, q_4) have crosstalk, which means it will have higher error rate if two two-qubit gates run simultaneously than independently on these two pairs of qubits. Among all error sources, crosstalk error dominates the overall fidelity of the circuit.

Previous studies aiming at crosstalk mitigation can be categorized into two approaches: *(i) Hardware approach*: Ding *et al.* [6] tune the frequency of each qubit in a program-aware

manner to alleviate the crosstalk. However this method relies on hardware to be capable of tuning the frequency of qubits while not every quantum computer has such capability, for instance, most IBM quantum computers use fixed frequency qubits [6]; *(ii) Software approach*: Murali *et al.* [16] avoid running two gates that cause crosstalk at the same time by delaying one of them to a later time step. Their approach uses a SMT solver to find the best timing for each gate in the circuit to maximize the overall fidelity. However, their approach performs gate scheduling only after the program has been compiled into a hardware-compliant circuit. We find that we can mitigate crosstalk at a much earlier stage, the program compilation stage, where hardware mapping and routing is determined for a given program IR. It can already reduce a lot of crosstalk errors before it comes to the scheduling process.

We give an example in Fig. 2 to better demonstrate the idea of compile-time crosstalk mitigation. Fig. 2 (a) shows an architecture with 6 physical qubits, and Fig. 2 (b) a program IR with two CX gates. Two pairs of links are prone to crosstalk error, which are $\{(Q_0, Q_1), (Q_4, Q_5)\}$ and $\{(Q_1, Q_2), (Q_3, Q_4)\}$ marked in Fig. 2 (a). As shown in the program IR, there are two CX gates *CX(q_0, q_2)* and *CX(q_3, q_5)*, which cannot run immediately due to the qubit coupling constraint that only two neighboring hardware qubits can communicate, and due to the initial mapping $q_i \rightarrow Q_i$ (i=0 to 5). After performing hardware mapping and routing, there are two candidate hardware-compliant circuits in Fig. 2 (c) and (d) which are both executable. In Fig. 2 (c), it performs *SWAP(q_1, q_2)* and *SWAP(q_3, q_4)*, and in Fig. 2 (d) it performs *SWAP(q_0, q_1)* and *SWAP(q_3, q_4)*. However, for Fig. 2 (c), *SWAP(q_1, q_2)* and *SWAP(q_3, q_4)* are on two links that are prone to crosstalk, and hence one of them has to be delayed with an instruction barrier, no matter how good the runtime scheduler is in the next stage. But, for Fig. 2 (d), *SWAP(q_0, q_1)* and *SWAP(q_3, q_4)* are not on crosstalk-prone links, and hence neither gate needs to be delayed. Overall the hardware mapping in Fig. 2 (d) suits better for crosstalk mitigation than that in Fig. 2 (c).

Hardware mapping is an important component in the workflow of quantum program compilation and execution as shown in Fig. 3. A quantum program is compiled into a quantum IR (in terms of quantum circuit) and then the IR mapped to a hardware compliant circuit with proper mapping and routing through swap gates insertion (as of Qubit Mapping). Current qubit mapping techniques [13,30–32,35] are crosstalk-oblivious. Crosstalk is typically

postponed to the runtime scheduling stage of the quantum program, and hence misses crosstalk optimization opportunities across the compilation stack.

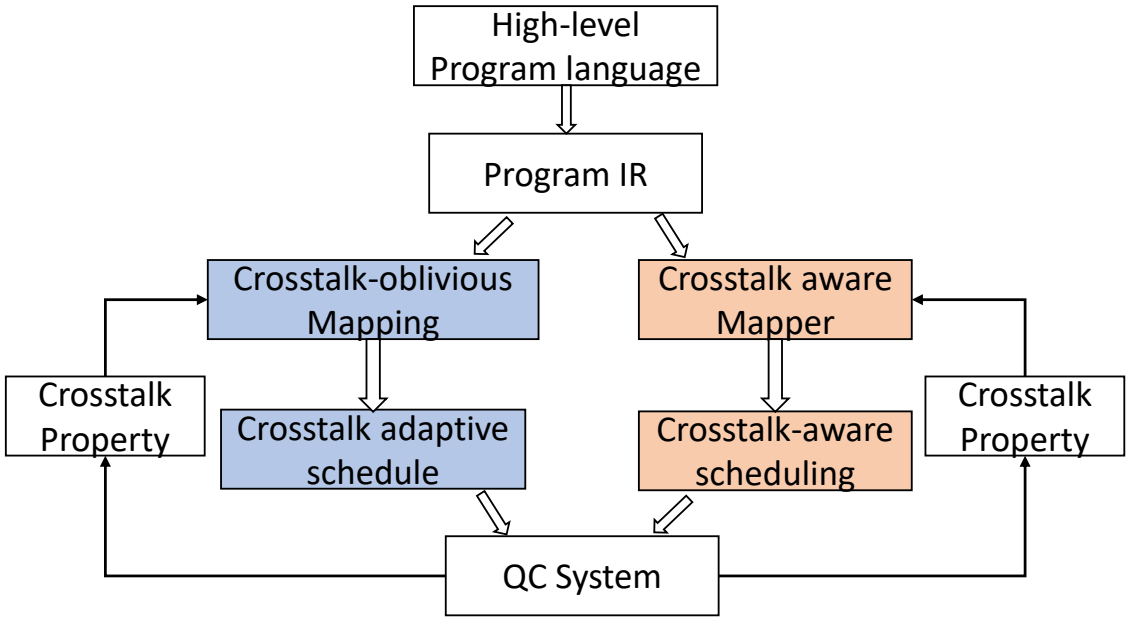

Figure 3: Quantum Program Compilation Workflow

1.3 Overview

In this dissertation, we aim to address the problems of time-aware and crosstalk-aware qubit mapping. The dissertation can be partitioned into four major works:

- First, a depth-aware SWAP insertion scheme that yields promising results in reducing the depth of generated circuit. Our results show we can reduce the depth of the transformed circuit by up to 30% compared with two best known qubit mappers [13, 28], and in the meantime, have on average less than 3% additional gates over a large set of representative benchmarks.
- Then, a slack-ware qubit mapping solution. We discover the key is to find intervals with slack in the circuit and to use the slack to hide the latency of inserted swap

operations. Our implemented qubit mapper named SlackQ automatically searches for dynamic qubit mappings given an input program on a quantum architecture with arbitrary qubit connectivity. The experiments show that SlackQ improves performance by up to 2.36X, by 1.62X on average, over 106 representative benchmarks from RevLib [29], IBM Qiskit [20], and ScaffCC [9]

- Furthermore, a time-optimal qubit mapper which guarantees its optimality. We present the first theoretical model for time-optimal qubit mapping without any implicit constraints. The theoretical model can be flexibly extended to practical algorithms. e discovered time-optimal solutions for quantum fourier transformation (QFT) on both 1D and 2D nearest neighbor architecture (using our search framework). Our solution for 1D nearest neighbor architecture is the same as a manual solution reported by Maslov [15]. But our optimal solution for 2D architecture is for the first time reported.
- Lastly, a crosstalk-aware qubit mapping framework that takes not only circuit depth but also circuit fidelity by reducing the crosstalk of the generated circuit. We propose an optimal solution and a reasonable heuristic solution to tackle the crosstalk problem.

2.0 Background and Related Work

2.1 Quantum Computing Basics

2.1.1 Qubit

A quantum bit or qubit, is the counterpart to classical bit in the realm of quantum computing. Different from a classical bit that represents either '1' or '0', a qubit is in the coherent superposition of both states. It is considered as a two-state quantum system that exhibits the peculiarity of quantum mechanics [17]. An example is the spin of the electron that the two states can be spin up and spin down.

The mathematical form of a qubit is represented as a combination of $|1\rangle$ and $|0\rangle$, which is shown as $|\Phi\rangle = \alpha\,|1\rangle + \beta\,|0\rangle$. α and β are the probability amplitudes and are both complex numbers, which have to comply with the constraint: $|\alpha|^2 + |\beta|^2 = 1$. With the increasing of the number of qubits, the total number of states that can be represented grow exponentially. For n qubits, there are at most 2^n states that can be represented at the same time.

2.1.2 Quantum Gates

There are two types of basic quantum gates. One type of basic gates is the single-qubit gate, a unitary quantum operation that can be abstracted as the rotation around the axis of the Bloch sphere [17] which represents the state space of one qubit. A single qubit-gate can be parameterized using two rotation angles around the axes. There are several elementary single-qubit gates including the Hadamard (H) gate, the phase (S) gate, and the $\pi/8$ (T) gate [17]. The other type of basic gates is the multi-qubit gate. However, all complex quantum gates can be decomposed into a sequence of single qubit gates H, S, T, and the two-qubit CNOT gate. Thus we only focus on the two-qubit CNOT gate. The CNOT gate operates on two qubits which are distinguished as a control qubit and a target qubit. If the control qubit is 1, the CNOT gate flips the state of the target qubit, otherwise, the target qubit remains the same.

2.1.3 Quantum Circuit

Quantum circuit is composed of a set of qubits and a sequence of quantum operations on these qubits. There are various ways to describe the quantum circuits. One way is to use the quantum assembly language called OpenQASM [5] released by IBM. Another way is to use the circuit diagram, in which qubits are represented as horizontal lines and quantum operations are the different blocks on those lines. In Fig. 4 (b), we show a simple example of quantum circuit diagram. A single-qubit gate is denoted as a square on the line, and one CNOT gate is represented by a line connecting two qubits and a circle enclosing a plus sign.

2.2 Noisy Intermediate-scale Quantum Hardware

The concept of Noisy Intermediate-scale Quantum (NISQ) device was first proposed by John Preskill [19], that represents the kind of quantum computer with limited number (dozens to hundreds) of qubits and also with limited reliability due to the physical qubits being implemented without quantum error correction. NISQ devices may be able to exhibit quantum supremacy over classical computing on certain computation tasks, but the lack of reliability would make the circuit difficult to scale. The NISQ technology may not change the world immediately, but it's a good first step towards the more powerful quantum devices in the future.

The industry has been deeply involved in the development of NISQ devices. Google, Microsoft, IBM all released their NISQ devices [8,10,11], with qubits number ranging from 14 to more than 70. Among these devices, IBM's QX series stand out as are the first quantum chips available to public via cloud access. It has been intensively used by more than 100,000 users since 2017 [28]..

2.3 Qubit Mapping Problem

To execute a quantum circuit on a real machine, logical qubits must be mapped to physical qubit on the target hardware. When applying a two-qubit gate, the two participating logical qubits must be mapped to physically connected qubits. Due to the bounded degree connectivity of physical qubits on current devices, it is generally considered impossible to find an initial qubit mapping that satisfies all two-qubit gates in the entire circuit.

A common practice is to dynamically map the logical qubits by inserting SWAP gates. A swap gate exchanges the states of the two operand qubits. There are different ways to implement a SWAP. A typical way to implement a SWAP is to use 3 CNOT gates if the links between physical qubits are bidirectional. Our model does not impose constraints on how a SWAP is implemented. Rather, we set the latency of a SWAP as a parameter in our model. We use various SWAP latencies according to the architectures we are evaluating.

The qubit mapping problem takes a logical circuit and a hardware coupling graph as input, outputs a transformed circuit. Only swap operations are allowed to be added into the transformed circuit. After transformation, all two-qubit gates must be hardware compliant. If a two-qubit gate g is performed on logical qubits q_1 and q_2, when running g, q_1 and q_2 must be physically connected.

One logical qubit can be mapped to different physical qubits at different points of circuit execution. Once a two-qubit gate is completed, both of its logical qubits can be remapped to some other physical qubits in the future. An example of qubit mapping is shown in Fig. 4 (c) and (d).

2.4 Parallelism in Quantum Circuit

Like in classical computers, parallelism is also important in quantum computers. Parallelism comes from independent operations on different qubits. Gates on the same qubit have to run sequentially. For instance, if a and b are two consecutive gates on the same qubit, and a is before b in the program, then gate b depends on a. Gates that do not share a single

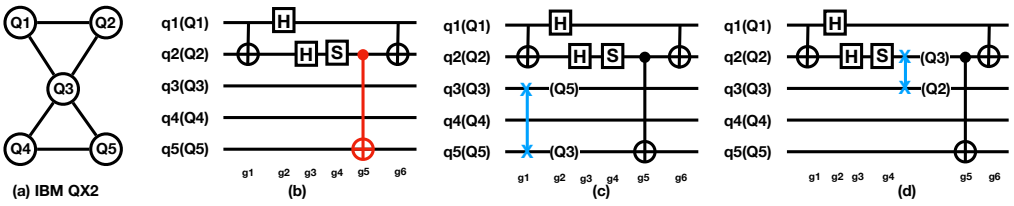

Figure 4: (a) Physical qubit connectivity; (b) the original logical circuit (logical qubits q1, q2, q3, q4, q5 are mapped to physical qubits Q1, Q2, Q3, Q4, Q5); The gate marked in red is the CNOT gate that cannot be executed due to a connectivity constraint (as Q2 and Q5 are not physically connected). (c) uses 1 swap but the execution time of the circuit is not increased; (d) uses 1 swap but the execution time of the circuit is increased. The operations marked in blue are swap operations. We assume a swap operation can be decomposed into three CNOT gates and each gate takes 1 cycle in this example.

qubit are independent. A two-qubit gate depend on up to two gates since it involves two qubits. A dependence graph can be built with respect to the partial order between gates mentioned above. It is a directed acyclic graph (DAG) as the edges have directions and there are no cycles in the graph.

In a transformed circuit, the parallelism could be between the gates in the original circuit (as g2 and g3 in Fig. 4 (b)), between the gates that are inserted to the original circuit,or between a gate in the original circuit and a newly inserted gate (as swap 3,5 and g1 in Fig. 4(c)). A good qubit mapping algorithm should consider all types of parallelism. However, most existing studies only consider the first types of parallelism. Our work is the first one that systematically exploits all type of parallelism and builds a formal model for the overall performance of the transformed circuit.

Noisy Intermediate-Scale Quantum (NISQ [19]) represents the standard of current mainstream quantum computers with the capacity of from dozens to hundreds of qubits. This type of QC lacks the ability to self-correct its bit errors, which makes it vulnerable to the ambient noise and unstable to maintain its integrity.

NISQ bottlenecks	
Low fidelity	High error rate of quantum gates: 2% for CNOT gates, 0.3% for single-qubit gate on average
Time-sensitive	Short qubit lifetime (couple of ∼us), unsuitable for long-run quantum circuit
Unreliable	Need large number of trials for consolidating final result
Error-prone	Error in one quantum gate can lead to incorrectness of the entire circuit
Unscalable	Unable to scale to large quantum circuits

Table 1: The bottlenecks of current NISQ devices

The NISQ devices exhibit traits of being unreliable, low fidelity, time-sensitive and error-prone, unscalable (explained in table 1). A qubit is volatile and error prone. It gradually decays over time and may have phase and bit flip errors. It may completely lose its state after a certain period of time, called *coherence time*. Quantum error correction (QEC) codes can detect error syndromes and fix them. However, QEC needs to use a large number of redundant physical qubits. A realistic QEC circuit may need more than 10,000 physical qubits, which is not possible for today's NISQ device. Without QEC, a program must terminate within a threshold amount of time. IBM proposed the metric of *quantum volume* [4] for evaluating the effectiveness of quantum computers which accounts for not only the width of the circuit (the number of qubits), but also the depth, how much time the circuit can execute. It's of great urgency to build a compilation system that we can map the logical circuit to physical quantum device in a way that targets less execution time. An execution-time aware mapping strategy can make the program less vulnerable to the drawbacks of the NISQ devices.

2.5 Circuit Error Model and Crosstalk

In quantum computer system, one of the major different between QC system and classical system is that quantum computer suffers a lot from different type errors, we will introduce them one by one.

Single-qubit gates and decoherence error: In superconducting transom systems, single-qubit gates are implemented by driving the target qubit via the microwave. For example, Rx and Ry totations can be implemented by sending microwave voltage singals. However all these implementation might lead to some errors which is called single qubit error. Another important qubit property is that the qubit is not stable and can have decay. Such decay can happen in two ways: i) T1 relaxation (i.e. spontaneous loss of energy causing decay from $|1>$ to $|0>$, and ii) T2 dephasing time (i.e. loss of relative quantum phase between $|0>$ and $|1>$. We can model both decays in a combined decoherence error: $q(t) = (1 - e^{-t/T1})(1 - e^{-t/T2})$, where t is qubit life time, and T1, T2 are constants characterizing the speed of the decays, for some qubit q.

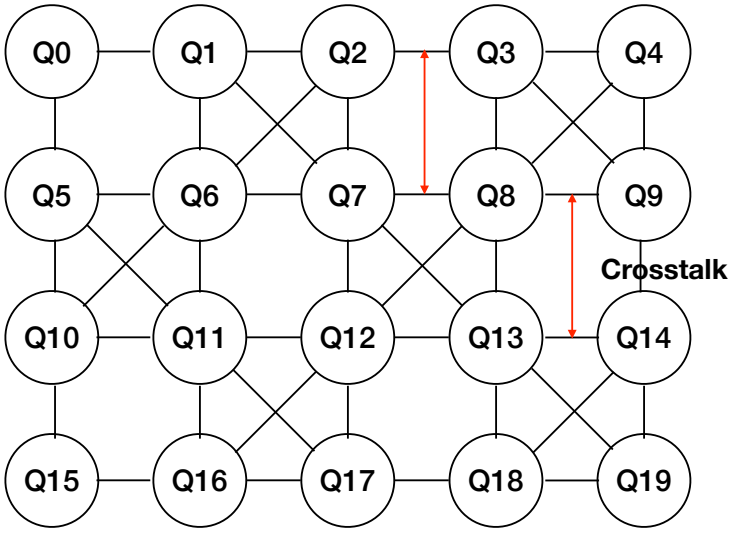

Figure 5: Tokyo architecture and some crosstalk errors.

Two-qubit gates error and cross-talk error: Two-qubit gates play important roles in quantum computation, as they implement entangling operations, that is, transformations of one qubit conditioned on the state of the other qubit. Some commonly used two-qubit

gates include CNOT (controlled-not) gate and SWAP gate. Since two qubit gate needs two corresponding qubits and it cause more errors, typically two qubit gate in most cases are the major error in the whole quantum program, and there are plenty of studies to mitigation two qubit error in both mapping and scheduling. Two-qubit gates are implemented by tuning the frequencies of the two interacting qubits to the interaction frequencies. Then, the qubits are held at that frequency for a duration of time t. Crosstalk occurs when a pair of two-qubit gates (on connected qubits simultaneously) happened to use very close interaction frequencies.

2.6 Quantum Program Transpile Process

From high level programming language to quantum gates which ready to be executed on hardware quantum machine, there are several critical process in the middle. The process is shown in Fig.2, we first need to transfer the high level program language into intermediate representation (IR) shown in Fig.2. To enable the execution of a quantum circuit, the logical qubits in the circuit must be mapped to the physical qubit on the target hardware. When applying a two-qubit gate, the two participating qubits need to be physically connected to each other. Due to the bounded degree connectivity of physical qubits on current devices, it is generally considered impossible to find an initial mapping that makes the entire circuit hardware-compliant.

The qubit mapping process takes a logical circuit and a hardware coupling graph as input, outputs a transformed circuit. Only swap operations are allowed to be added into the transformed circuit. After transformation, all two-qubit gates must be hardware compliant. The common practice is to remap the logical qubits is inserting SWAP gates. A swap gate exchanges the states of the two input qubits. There are different ways to implement a SWAP. One way is to insert three CNOTs to implement a SWAP. However, prior study uses crosstalk-oblivious mapping for mitigating the crosstalks. We are first try to mitigate crosstalk errors at this process which can have multiple advantages. For the example we shown in Fig.2, there are several different choices to insert swaps, instead of using the swap-oblivious way, we try to use the swaps which does not cause or cause tiny crosstalk errors, and we

called this method crosstalk-aware mapping approach.

After the mapping process, it comes to gate scheduling process, based on qubit dependency, quantum gates can only be executed if and only if the qubits meet the dependency constraint. The crosstalk-adaptive schedule will choose to delay the gates which cause the crosstalk errors. And our approach has two benefits, as we talked before, we mitigate the crosstalk errors prior to mapping process, so it will mitigate some crosstalk errors caused by swaps, then we don not need to worry about these gates. Another benefit is that we can set some crosstalk tolerance, find the trade-off between crosstalk errors and depths overhead, which try to maximize the overal fidelity instead of just delaying the gates. And we recursively do this process until all the gates have been scheduled and we push into QC systems.

3.0 A Depth-Aware SWAP Insertion Scheme

3.1 Introduction

In this chapter, we propose the one of the first depth-aware qubit mapping schemes for quantum circuits running on arbitrary qubit connectivity hardware. Our depth-aware qubit mapper searches for the mapping that minimizes the transformed circuit depth and keeps the gate count within a reasonable range. Our results show we can reduce the depth of the transformed circuit by up to 30% compared with two best known qubit mappers [13, 28], and in the meantime, have on average less than 3% additional gates over a large set of representative benchmarks.

3.2 Insight and Design

3.2.1 Metric

As our work is a depth-aware SWAP insertion scheme, we first precisely define the metric for characterizing the depth of a circuit. In order to fully explain the metric, we need to introduce the concepts of *dependency graph* and *critical path*.

The dependency graph represents the precedence relation between quantum gates in a logical quantum circuit. The definition is below:

Definition 1. *Dependency Graph : The dependency graph of a quantum circuit C with a set of gates Ψ is a Directed Acyclic Graph $G_\psi = (\Psi, E_\psi)$, $E_\psi \subseteq \psi \times \psi$. A directed edge from node ψ_1 to node ψ_2 exists if and only if the output of gate ψ_1 is (part of) the input of gate ψ_2 in the quantum circuit C.*

The critical path is referred to as the longest path in the dependency graph. And the definition is below:

Definition 2. *Critical Path : Given a dependency graph $G_\psi = (\Psi, E_\psi)$ of a quantum circuit. The critical path is $CP = Max(Path(\psi_1, \psi_2))$ s.t. $\psi_1, \psi_2 \in E_\psi$ and $\psi_1 \neq \psi_2$*

Algorithm 1: Calculate the Critical Path of a Circuit

```
Input  : The circuit's dependency graph G(V, E)
Output : The critical path CP

earliest_start = {};
CP = 0;
for n ∈ V in topological order do
    temp = 0;
    for p ∈ V's predecessors do
        if temp < earliest_start[p] + latency[p] then
            temp = earliest_start[p] + latency[p];
        end
    end
    earliest_start[n] = temp;
    if CP < temp + latency[n] then
        CP = temp;
    end
end
return CP ;
```

The depth is characterizing the number of execution steps of a quantum circuit, which is tantamount to the critical path length of the circuit. The longest path in the dependence graph describes the minimal number of steps the circuit needs in order for every gate's data dependence be resolved. In Algorithm 1, we show how we calculate the critical path.

We first sort the nodes in the directed acyclic graph in topological order. Then we process the nodes in that order. For each node, we check the earliest start time for each of its predecessors, and add it by the latency of that predecessor, then we choose the maximum and use it as the earliest start time of this node. The maximum of all nodes' earliest start time added by their latency is the critical path length.

We use the critical path length as the metric for ranking different swap insertion options.

3.2.2 Framework Design

With the metric precisely explained in previous section, now we continue to explain the work flow of our framework and the intuitions behind it.

Before delving into the details of this framework, we need to define the *layer* and the *coupling graph*.

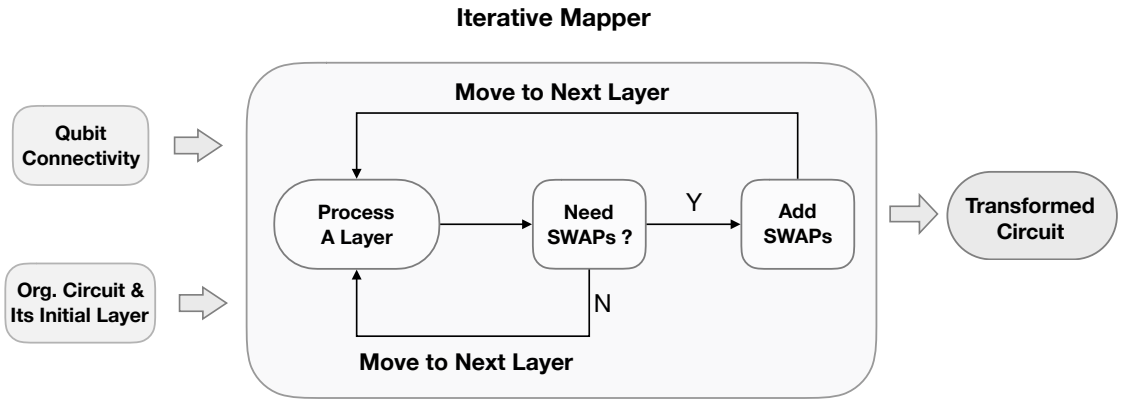

Figure 6: The Qubit Mapping Framework

Definition 3. *Coupling Graph : The coupling graph of a quantum architecture X with a set of physical qubits Q is a directed graph $G = (Q, E)$, $E \subseteq Q \times Q$. The edge $E_x = (Q_1, Q_2) \in E$ if and only if a $CNOT$ gate can be applied to Q_1 and Q_2 in X with Q_1 being the control qubit and Q_2 being the target qubit.*

We can divide the set of quantum gates in a circuit into layers, so that all gates in the same layer can be executed concurrently. The formal definition of a layer is:

Definition 4. *Layer : A quantum circuit C can be divided into layers $L = l_1, l_2, l_3, ..., l_m$, while $\bigcup_{i=1}^{m} l_i = C$ and $\bigcap_{i=1}^{m} l_i = \emptyset$. The set of gates at layer l_i can run concurrently and act on distinct sets of qubits.*

To divide a circuit into layers, we group the gates that have the same *earliest start time* (defined in Algorithm 1) into the same layer. The order of the layers is thus determined by

the order of the *earliest start times*.

We use an iterative process to find the mapping. Our framework is depicted in Fig. 6. And this iterative process is explained as below. We start the framework by taking the input of the coupling graph (also denoted as *Qubit Connectivity*) and the original circuit's initial layer.

We process the circuit layer by layer. Given a layer, we perform the following steps.

- We check the layer to see if it is hardware-compliant based on the coupling graph and the qubit mapping before current layer is scheduled.
- If YES, we move on to next layer.
- If NO, we invoke our mapping searcher to search for (the set of) swaps that are necessary to solve the current layer. We consider depth-awareness during the selection of the set of swap gates – the resulted mapping of which generates the smallest critical path length (described in Section 3.2.3). After we find a hardware-compliant mapping, we move to the next layer.

After all layers are processed, the mapping terminates.

3.2.3 Circuit Mapping Searcher

Here we describe the specific mapping searcher we use to overcome the coupling constraint for a given layer.

We build our method upon the *A-star* algorithm for finding valid mappings that minimize the number of only the inserted SWAP gates [35]. We extend it by changing the ranking metric and allowing it to search for feasible mappings that do not necessarily have the smallest SWAP gate counts. It will help us search in a way that minimizes the depth while not significantly increasing the gate count.

We rank the swap options by the increase in the critical path length. Since it is an iterative process that handles the gates layer by layer, it is tempting to consider only minimizing the depth of the already processed circuit when deciding which swaps to use.

But the example in Fig. 7 shows that not only the processed circuit, but also the remaining circuit can help overlap the SWAPs with existing gates in the circuit without affecting the

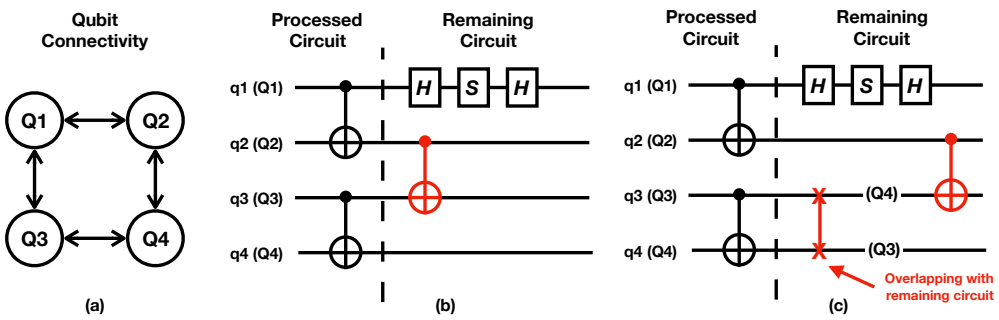

Figure 7: (a) Layout of an example architecture with 4 physical qubits (b) Example of a quantum circuit, the dashed line separates the processed circuit and the remaining circuit (c) Inserted SWAP overlaps with remaining circuit instead of existing processed circuit

critical path. As shown in Fig. 7, for the CNOT gate (in red), there is no way it can overlap the necessary SWAPs with the processed circuit (dubbed as the circuit before the dashed line). But when we look after the dashed line, the three single-qubit gates can overlap with inserted SWAP. And this renders less impact to the depth of the resulting circuit, compared to if we insert the SWAP on Q_1 and Q_2.

Based on this intuition, we design our scheme of choosing the SWAP candidate as in Fig. 8. For each of the hardware-compliant remapping candidates that we acquire from the *A-star* searcher, we calculate the critical path after merging the candidate (set of) swap(s) with both the processed circuit and the not-processed circuit. We choose the mapping that yields the shortest critical path.

3.2.4 Optimizations

We use two ways to optimize our proposed solution. One is to expand more nodes during the *A-star* search, and another one is to search into deeper levels.

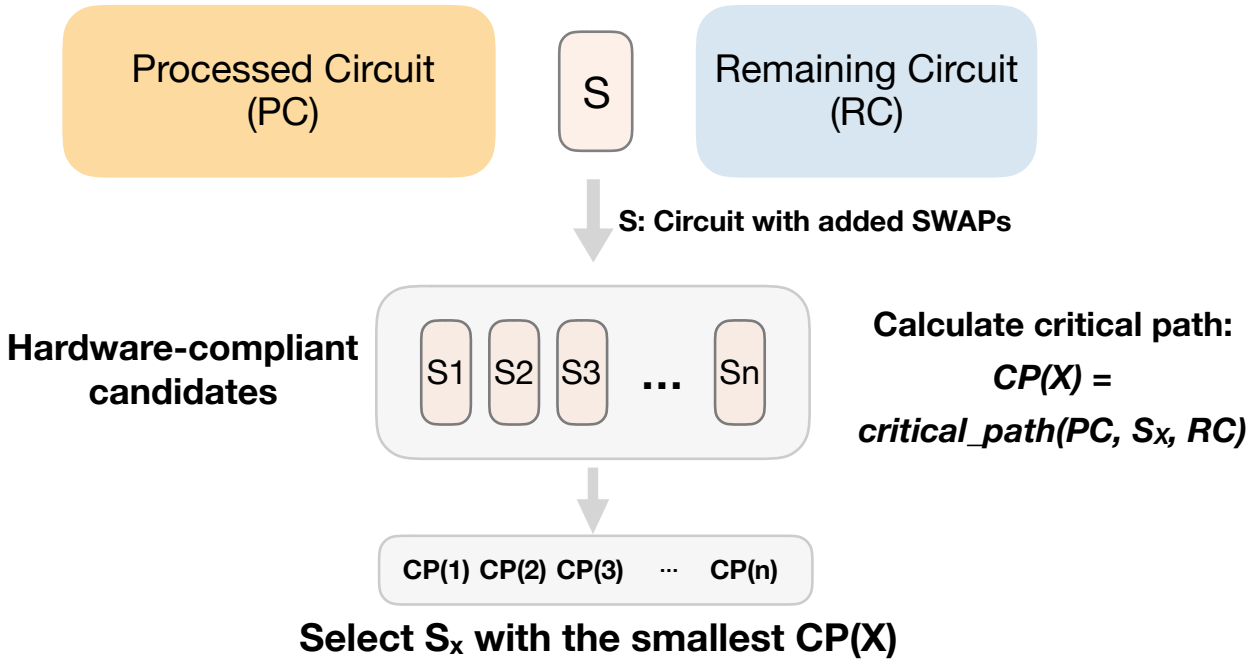

Figure 8: Choose SWAP Candidates

3.2.4.1 Expand More Nodes

In the search process for *A-star*, the normal routine is to expand the one node of least cost at each step. Here, we can expand more than one node at each step and increase the search space. The number of nodes that can be expanded at a time can go from 1 to larger number.

3.2.4.2 Deeper Search

We increase the depth of the A-star search tree. In normal case, the search process ends when it finds the first node that minimizes the number of SWAPs, which is reflected as a certain level of the A-star tree. To this end, the second optimization that we applied here is to continue the search into a deeper level of the A-star tree. We can specify and tune the parameter of the deeper search.

By tuning these parameters, there are more possible nodes added into our search space.

With a larger search space, we have a larger possibility to jump out of one local optima and go to the global optima.

3.3 Evaluation

Table 2: Summary of Experiment results

Benchmark		**Total Gate #**			**Depth**	**Depth-delta**			**Improvement**	
name	n	Zulehner	Sabre	**DPS**	Original	Zulehner	Sabre	**DPS**	Min	Max
4gt5_75	5	131	122	119	47	44	44	29	1.52	1.52
mini-alu_167	5	435	396	432	162	131	125	119	1.05	1.10
mod10_171	5	361	328	298	139	117	89	39	2.28	3
alu-v2_30	6	804	717	795	285	261	241	201	1.20	1.30
decod24-enable_126	6	533	476	509	190	187	150	141	1.06	1.33
mod5adder_127	6	849	780	858	302	256	256	222	1.15	1.15
4mod5-bdd_287	7	94	94	94	41	18	23	17	1.06	1.35
alu-bdd_288	7	126	117	135	48	36	36	30	1.2	1.2
majority_239	7	915	780	885	344	265	194	182	1.06	1.46
rd53_130	7	1619	1508	1619	569	529	482	384	1.26	1.38
rd53_135	7	419	410	422	159	116	112	109	1.03	1.06
rd53_138	8	186	183	174	56	37	40	21	1.76	1.90
cm82a_208	8	899	944	1007	337	219	295	213	1.03	1.38
qft_10	10	266	263	281	63	47	96	44	1.07	2.18
rd73_140	10	347	329	338	92	84	79	67	1.18	1.25
dc1_220	11	2868	2685	3129	1038	820	697	681	1.02	1.20
wim_266	11	1505	1415	1511	514	431	450	311	1.39	1.45
z4_268	11	4453	4477	4972	1644	1162	1492	1076	1.08	1.39
cycle10_2_110	12	9143	8666	10115	3386	2467	2640	2421	1.02	1.09
sym9_146	12	493	454	472	127	118	138	86	1.37	1.60
adr4_197	13	5299	5017	5530	1839	1439	1599	1210	1.19	1.32
rd53_311	13	467	413	446	124	138	157	87	1.59	1.80
cnt3-5_179	16	325	238	286	61	79	59	43	1.37	1.84

We compare the total gate count generated. For depth, we compare the increased depth for each benchmark, denoted as "Depth-delta" here. The improvement represents the ratio of a baseline's depth-delta divided by DPS's depth-delta. Min/Max represents the improvement over the best/worst baseline.

In this section, we evaluate our **dep**th-aware **s**wap insertion scheme (denoted as DPS) and compare it with the two state-of-the-art qubit mappers. The experiment setup is listed below:

- **Benchmarks**: We use the quantum circuits from RevLib [29], IBM Qiskit [20], and ScaffCC [9].

- **Hardware Model**: We use IBM's 20-qubit Q20 Tokyo architecture, which was used in [13]'s work. The qubit connectivity graph is shown in Fig. 9.
- **Evaluation Platform**: The mapping experiments are conducted on a Intel 2.4 GHz Core i5 machine, with 8 GB 1600 MHz DDR3 memory. The operating system is MacOS Mojave. We use IBM's Qiskit [20] to evaluate the depth of the transformed circuit.
- **Baselines**: We compare our work with two best know qubit mapping solutions, the work by Zulehner and others [35] (denoted as *Zulehner*), the Sabre qubit mapper from [13] (denoted as *Sabre*), and IBM's stochastic mapper in Qiskit. Since IBM's Qiskit mapper is significantly worse in terms of gate count and depth than all other mappers we evaluate, as also evidenced in the work by Zulehner *et al.* [35], we do not present Qiskit results.
- **Metrics**: We are comparing the depth and gate count of the transformed circuit circuits for all different strategies.

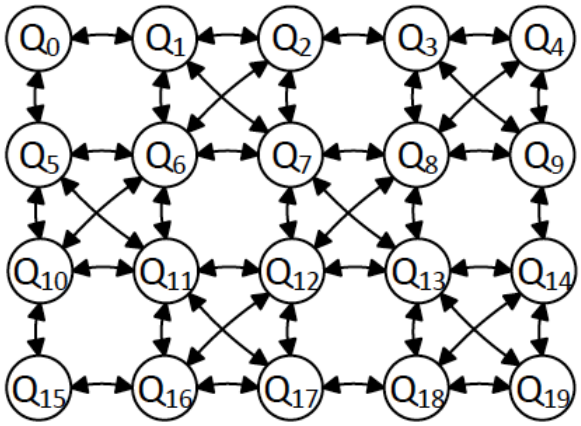

Figure 9: IBM Q20 Tokyo Physical Layout [13]

Table. 2 shows a summary experimental results. For gate count, we compare the total gate count generated in the transformed circuit. For depth, we compare the increased depth for each benchmark, denoted as "Depth-delta" in Table 2. The improvement columns provides the ratio between one of the two baseline's *depth-delta* and our *depth-delta*. We use the term *minimum improvement* to denote the improvement over the best of the two baselines, and the term *maximum improvement* to denote the improvement over the worse of the two baselines.

We discuss our findings from the following three aspects: depth reduction, gate count change, and the trade-off between gate count and depth.

3.3.1 Depth Reduction

For depth reduction, as shown in Table 2, our proposed solution outperforms the two baselines *Zulehner* and *Sabre*. Comparing *depth-delta*, the added depth of the circuit, our approach outperforms the better of the two baselines by more than 20% and up to 3X. For five out of the twenty-three benchmarks, our improvement on *depth-delta* is less than 20% compared with the better of the two baselines. However, for these cases, our approach still achieves considerable improvement over the worse of the two baselines. In these cases, it is possible that one of the two baselines happen to achieve very good depth in the transformed circuit and there is not much potential to improve. But our approach is still able to find a good mapping for these benchmarks and the performance is on par with the better of the two baselines.

3.3.2 Gates Count Changes

The primary goal of our depth-aware qubit mapper is to minimize the depth of the circuit. However, we discover that our qubit mapper can sometimes reduce the gate count. We discover that four out of the twenty three (17%) benchmarks, our qubit mapper yields the smallest number of gates among all three versions of qubit mappers. For 57% of these benchmarks, our method is ranked among top-2 of the three qubit mappers in terms of gate count. For the benchmarks where our method yields the largest gate count, the increased gate count percentage is negligible. On average, our depth-aware qubit mapper adds 3% gate count. From the experiment results, we can see that our solution does not greatly increase the number of gates while reducing the depth of the circuit.

3.3.3 Trade-off between Gate Count and Depth

While all previous works focus on reducing the total gate count (and the depth among the inserted gates themselves) after qubit mapping transformation, it is crucial to think about the trade-off between the resulted gate count and depth. Sometimes the choice made during the search process that favors the reduced gate count, might adversely affect the critical path. In Table 2, the *Sabre* mapper reduces the number of gates for 10-qubit QFT by 1.1% compared with *Zulehner*'s mapper, but increases the depth by 44.5%. For the *sym_9_246* benchmark, *Sabre* reduces the gate count by 3.8% compared with our approach, but increases the depth by 25.5%. Therefore a small reduction in the gate count may not be worthwhile if it increases the circuit depth significantly.

4.0 SlackQ: Slack-Aware SWAP Insertion Scheme

4.1 Introduction

A good time-aware qubit mapper needs to yield a hardware-compliant circuit while having optimal or near-optimal execution time. In this chapter, we discover the key is to find intervals with slack in the circuit and to use the slack to hide the latency of inserted swap operations. We present important considerations for detecting and exploiting slack in the circuit. Our implemented qubit mapper named SlackQ automatically searches for dynamic qubit mappings given an input program on a quantum architecture with arbitrary qubit connectivity. The experiments show that SlackQ improves performance by up to 2.36X, by 1.62X on average, over 106 representative benchmarks from RevLib [29], IBM Qiskit [20], and ScaffCC [9].

4.2 Insight and Design

To improve the parallelism between the inserted SWAP operation and the gates in the original circuit, we discover that it is important to exploit the slack intervals in the circuit. The slack represents the idle time in the original circuit for a given set of qubits. The key is to hide the latency of inserted swap operations by using the qubits that are idle at that time of the circuit execution. This forms the main idea of this paper and we insert SWAP operations such that they leverage slack in the circuit as much as possible.

4.2.1 Slack

We define slack as the idle time between two consecutive gates on the same qubit(s) and can be used to perform SWAP operation without affecting the total execution time of the entire circuit. The slack time is usually caused by dependence between gates and/or variation

of gate count on individual qubits.

The slack time due to dependence between gates only occurs when there are two-qubit gates in the circuit. Recall that a CNOT gate depends on up to two other gates, since CNOT is a two-qubit gate. If the qubits are running at different speeds, one of the other qubits might be ready earlier than the other. The faster qubit thus needs to wait for the slow qubit to finish before the CNOT gate can be executed. On the other hand, if a circuit has a number of qubits, and the number of gates on each qubit is different (even if they are all independent), then some qubits will inevitably be idle at some point of the execution. The slack intervals can be used for inserting swap operations that resolve qubit mapping constraints. An example of slack in the circuit is shown in Fig. 10.

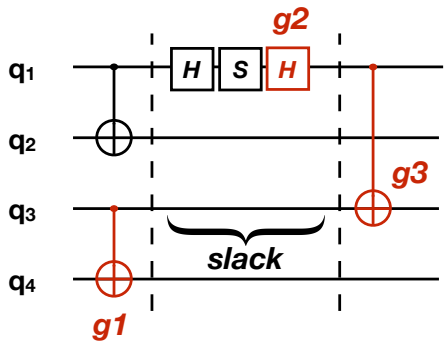

Figure 10: Slack in the circuit: Since g3 depends on g1 and g2, g1 finishes earlier than g2, therefore, for qubit q3, there is a slack interval of three cycles (assuming each gate takes one cycle) between g1 and g3 in the circuit.

There are two types of slacks in the circuit. One type does not require the rescheduling of the gates, and we define it as fixed slack. The other type of slacks may have variable number of cycles, and we denote it as flexible slack. A good qubit mapper needs to search globally and exploit both fixed and flexible slack.

An example of fixed slack is shown in Fig. 11 (b). Assuming each gate takes one cycle, there is a fixed slack between g1 and g2 on qubit q2. Here it cannot delay g2 or start g1 early if the total execution time needs to remain unchanged. If qubit q2 is used to perform another gate such as the swap operation during the three cycles, it will not affect the execution time of the entire circuit. In this case, the number of cycles that can be used on q2 between g1

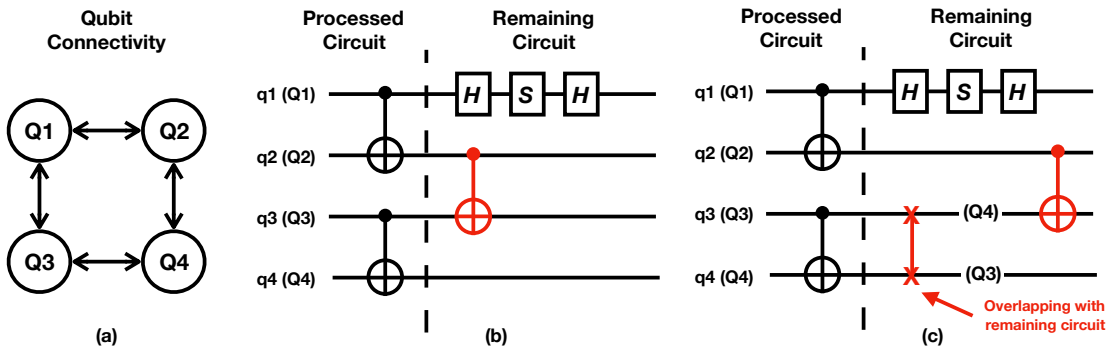

Figure 11: (a) Qubit coupling graph; (b) An example of fixed slack in the quantum circuit; (c) and (d) Examples of flexible slacks, where g1 can be moved within a time window without affecting the circuit execution time, the slack before g1 can be either 1 cycle or 2 cycles assuming every gate takes one cycle. Note the slack between g1 and g3 may also vary due to scheduling of g1.

and g2 is fixed.

Sometimes fixed slack does not always exist. It is necessary to move the gates in order to create slacks for latency hiding purpose. We show an example Fig. 11 (c) and (d), where slack can be created by moving $g1$. Let's say cnot$(q1, q2)$ and cnot(q_3, q_4) are scheduled on cycle 1. The three single-qubit gates on Q_1 are scheduled on cycle 2,3,4 respectively. With this going on, $g3$ expects to be executed on cycle 5 at the earliest. $g3$ depends on $g1$. $g1$ can be scheduled at the second cycle, the third cycle (Fig. 11 (c)) or the fourth cycle (Fig. 11 (d)) without delaying $g5$. To this end, a slack with zero, one or two cycles can be created between $g2$ and $g1$, depending on when $g1$ is scheduled. And this type of slack between $g2$ and $g1$ is flexible. On the other hand, since $g1$ is not directly executable due to the connectivity constraint in Fig. 11 (a), the more slack intervals before $g1$ there are, the better it is for hiding the swap latency. In Fig. 13, we show that by moving $g1$ forward, q2 and q3 can have more slack intervals before $g1$, and swap(3,4) is inserted which utilizes the slack, resulting in a total circuit time of 6 cycles only, which is optimal in this case.

It is worth mentioning flexible slack could be cascading as the rescheduling of one gate

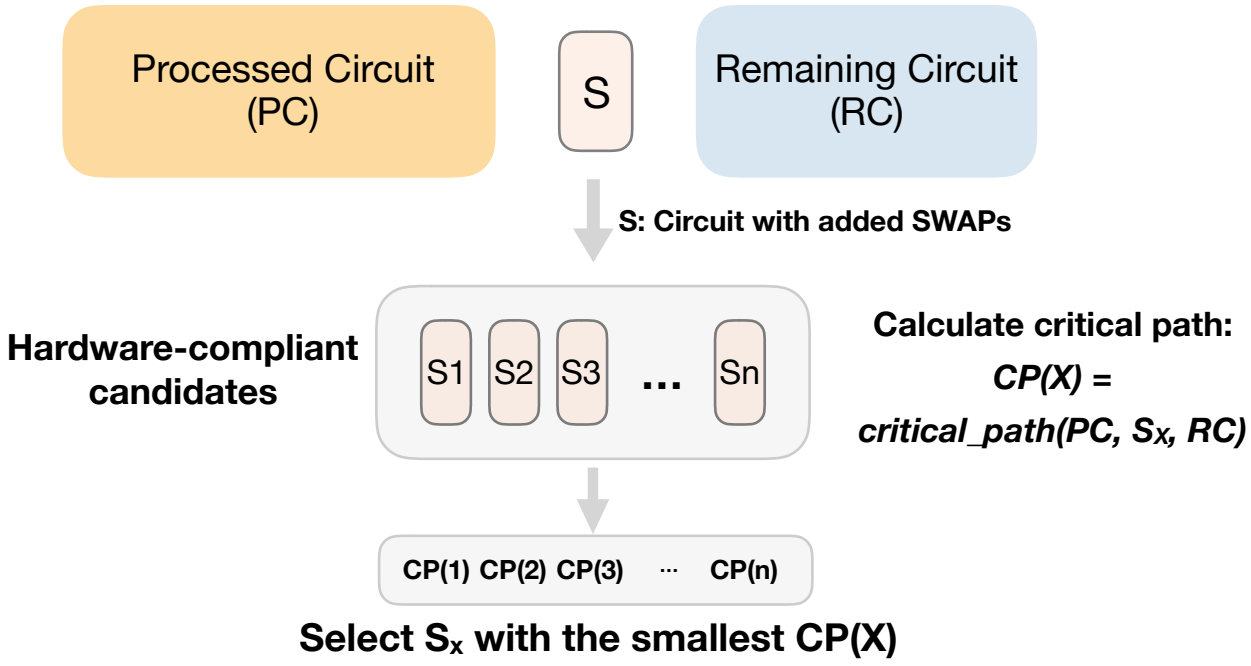

Figure 12: Choose SWAP Candidates

might affect its descendants or predecessors. For the fixed slack, the gates involved cannot be delayed without affecting the circuit time. Flexible slack allows one or multiple gates to delay start within reasonable time window(s). Flexible slack are more complicated than fixed slack. It is necessary to analyze and exploit flexible slack in a systematic way.

4.2.2 Dynamic Gate Scheduler

We model the resolution of qubit mapping conflicts as a dynamic scheduling process. Gates in the circuit are scheduled as soon as their dependencies are resolved. When a gate cannot be scheduled due to a connectivity problem, we insert a (combination of) swap(s) to change the qubit mapping so that the gate can be executed on the physical device. All the gates that have already been scheduled at one point of scheduling are called the Processed Circuit, and the gates that still await scheduling are called the Remaining Circuit.

Fig. 13 shows an example of how the scheduling works. With initial mapping of

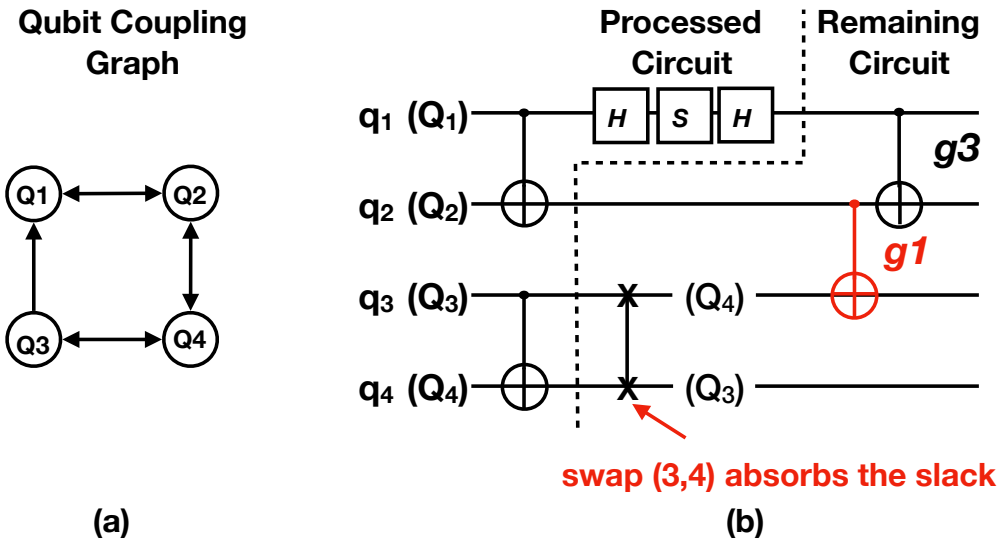

Figure 13: Scheduling gates to create more slacks: gate g1 can be moved forward such that swap Q3, Q4 can absorb the longer slack on q3

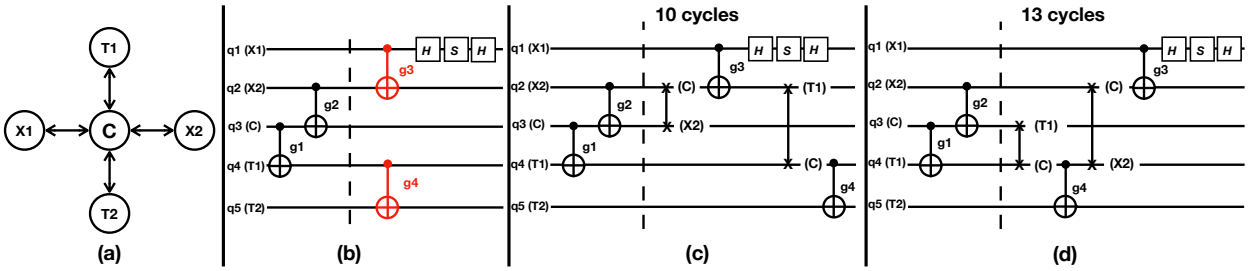

Figure 14: (a) Qubit Connectivity for a five-qubit machine (b) Original Circuit before qubit mapping (c) **Strategy One:** Resolving gates on critical path first (d) **Strategy Two:** Resolving gates on non-critical path first.

$\{q_1, q_2, q_3, q_4\} \rightarrow \{Q_1, Q_2, Q_3, Q_4\}$, the first two CNOTs cnot(q_1, q_2) and cnot(q_3, q_4) and the three single-qubit gates on $Q1$ can be scheduled without remapping. At this point, those gates that are scheduled are part of the Processed Circuit. The remaining two CNOT gates ($g1$ and $g3$) that cannot be scheduled are part of the Remaining Circuit. Gate $g1$ cannot be

scheduled because Q2 and Q3 are not connected in the device. Gate $g3$ cannot be scheduled because $g1$ must be scheduled before $g3$ (write-after-read dependency on Q2). The dashed lines divide the circuit into processed part and remaining part.

To minimize the circuit time, we search for swap candidates for $g1$ that results in maximally hiding swap latencies using circuit slack. Fig. 12 shows the key idea behind the searching for optimal swap candidates. The search reveals multiple hardware-compliant candidates that utilize different sequences of swaps to achieve compliance. We choose the optimal candidate by calculating the *Slack Utilization* of each candidate, and choosing the one with the best utilization. In Fig. 13, we choose swap candidate swap(q3, q4) since it best hides the swap latency behind the 2-cycle slack shown in Fig. 11 (d). Now $g1$ is satisfied and scheduling can proceed.

4.2.3 Critical Gates

The gates in the remaining circuit pending scheduling whose dependences have been resolved but connectivity problems haven't been can be divided into two groups: those on the critical path and those that are not. We denote the gates on the critical path as critical gates, and the others as non-critical gates. In parallel computing, the critical path length is equal to the execution time when there is enough parallelism. In this case, the critical path is equal to execution time as the maximum parallelism (the maximum number of gates that can run concurrently) is at most the same as the number of qubits. Thus it is important to prioritize the scheduling of critical gates over non-critical gates.

To prioritize critical gates, what we need to do is to resolve the connectivity problems of critical gates as early as possible. Imagine a scenario where two gates have connectivity problems, one gate is critical and the other is non-critical gate. Their connectivity issues cannot be resolved at the same time. Under this situation, we should resolve the critical gate first, as resolving the non-critical gates can be likely delayed without affecting the overall execution time.

We use an example from Fig. 14 to show how criticality can play an important role in determining the overall circuit time. We use a five-qubit quantum machine, whose connectivity

is shown in Fig. 14 (a). This example circuit consists of 4 CNOTs and 3 single-qubit gates, with $g1$, $g2$ scheduled, and $g3$, $g4$ not yet scheduled due to connectivity issues. It's crucial to note that g_3 is on the critical path, while $g4$ is not. The two gates $g3$ and $g4$ cannot be resolved at the same time if both of them want to use only one swap, since qubit C is the on the path from $T1$ to $T2$, and from $X1$ to $X2$. Whether to prioritize $g3$ over $g4$ when using the hub qubit C for swap, makes a big difference in terms of circuit time. We show this discrepancy by illustrating two strategies and their resulting circuits.

- **Strategy One - Prioritizing critical gates** Resolve $g3$ first. Shown in Fig 14 (c), it is necessary to insert swap($q2$, $q3$) before $g3$. After $g3$ is resolved and scheduled, swap($q2$, $q4$) is inserted such that $g4$ can be resolved. swap($q2$, $q4$) can take advantage of the slack on logical qubits $q2$ and $q4$, as logical qubit $q1$ is processing three single-qubit gates. This strategy results in total circuit time of 10 cycles, assuming CNOT and single-qubit gate both have latencies of one cycle.
- **Strategy Two - Not distinguishing critical gates from non-critical path** Resolve $g4$ first. Shown in Fig 14 (d), it is necessary to insert swap($q3$, $q4$) before $g4$, and let $g4$ be scheduled. In the meantime, $g3$ has to wait, which results in the critical path being elongated due to the delay of the execution time of $g4$ and swap($q3$, $q4$). It is because when $g4$ is being executed, the mapping that allows $g4$ must be kept, which will delay all the remaining gates. In this case, it is not desirable to delay all the remaining gates as they are on critical path. Delaying gates that are critical will have a more detrimental impact than delaying gates not on critical path. After $g4$ is resolved, the fastest way to resolve $g3$ is to swap($q2$, $q4$) before $g3$. This strategy as a whole results in total circuit time of 13 cycles, which is 30% more than strategy one.

It can be seen from this example the later resolving of the non-critical gates are highly likely to overlap with the gates on the critical path, and result in less impact to overall circuit execution.

4.3 Implementation

Based on the design consideration on Section 4.2, we implement a slack-aware qubit mapping framework called SlackQ.

4.3.1 Overview of SlackQ

Our algorithm is an iterative gate scheduler which dynamically resolves the connectivity issues encountered during the scheduling process. Initially, a dependency graph of the circuit is built. Then we traverse the dependency graph of the circuit and schedule the gates one by one. We keep a frontier set of gates ready to be scheduled. When resolving the connectivity issues, we invoke a priority-queue based searcher for swap candidates. It returns hardware-compliant candidates. Among these hardware-compliant candidates, the one that has the best slack utilization is chosen, and the scheduling process proceeds. We describe the algorithm below with respect to the pseudo-code shown in Algorithm 2:

Step One - Initialization This step prepares for the searching process. It builds the dependency graph of circuit. It finds the gates that do not depend on any other gates. Then it places those gates into the frontier F. It also initializes the processed gate set P as empty set, and the remaining gate sets R as the entire circuit.

Step Two - Schedule Ready Gates This step goes through frontier list F. It finds all gates in F that can be scheduled immediately due to having no connectivity issues according to the current mapping π. It schedules all these gates. When finishing the scheduling of one gate, it finds the descendant gate and see if this descendant's other parent has also been scheduled. If this is the case, the descendant gate's dependency is resolved. It then places this descendant gate into F. This step is repeated until F contains no gate that can be scheduled with respect to the current qubit mapping.

Step Three - Resolve Qubit Mapping Conflicts We go through the frontier F again, finding the gates with resolved dependencies but are constrained by the current mapping and are on the critical path of the remaining circuit. Put those gates into a set called $F_{critical}$. Run a priority queue based searcher for hardware-compliant mappings. Our mapping searcher

here returns a list of hardware-compliant mappings candidates, called M. Among these candidates, it finds the one (call it m) with the best slack utilization. Then we use the swap sequence associated with m to update the mapping π and add the swap sequence into the processed circuit P.

Step Four Repeat Step Two and Three until all gates are scheduled in the circuit. Return transformed circuit.

In Sections 4.3.2 to 4.3.3, we describe a few important aspects of this algorithm.

4.3.2 Choosing the Best Mapping Candidate

With multiple hardware-compliant mappings, it is necessary to determine the candidate that has the best slack utilization. The best slack utilization means the inserted swap sequence makes best use of the slack currently existing in the circuit. To evaluate these mapping candidates, for each of them, we tentatively insert the associated swap sequence and monitor how the swap insertions affect the dependence graph and the critical path. We trace the nodes that are affected due to the inserted swaps and detect how much their start/ending time changes.

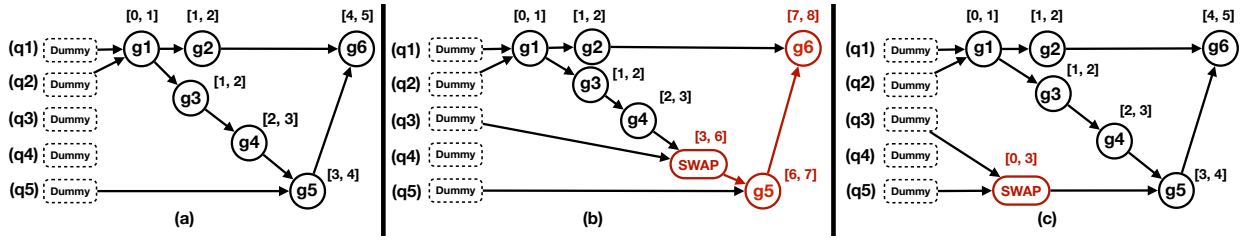

Figure 15: (a) The generated dependency graph from the example of Fig. 4. The numbers displayed on each gate refer to the start/end cycle of this gate. (b) One mapping candidate whose inserted swap results in two later gates delaying its start/end cycles. (C) Another mapping candidate whose inserted swap does not affect the start/end cycles of the entire circuit.

We again use the example circuit and qubit coupling graph in Fig. 4 to show how our evaluation approach works. We first show the original dependency graph in Fig. 15 (a). The

Algorithm 2: Dynamic Gate Scheduler

Input : Frontier F, initial mapping π, processed circuit P, remaining circuit R

Output : Transformed circuit T

```
while F not empty do
    E = getSchedulableGates(F, π);
    while E not empty do
        F.remove(E);
        P.add(E);
        R.remove(E);
        for g ∈ E do
            for d ∈ g.children do
                if d's dependency is resolved then
                    F.add(d);
                end
            end
        end
        E = getSchedulableGates(F, π);
    end
    F_critical = select_critical_gates (F);
    mapping_candidates = resolve_conflicts (F_critical, π);
    m = best_slack_utilization (mapping_candidates, P, R);
    π = update_mapping_with_swaps(m, π);
    P.add(m.swaps);
end
return P;
```

numbers on gates denote the start/end cycle of each gate. For instance, [0, 1] represents the start cycle as 0 and the ending cycle as 1. We assign a dummy gate node at the beginning of the circuit for each qubit $q1$ ˜ $q5$, for the sake of illustration. The dummy node starts at time

0 and takes 0 cycles. In this example, we have two possible mapping candidates whose swap sequences are to be inserted on different qubits. We need to choose the better one out of the two mapping candidates. The first mapping candidate shown in Fig. 15 (b) inserts one swap on logical qubits $q2$ (the qubit for $g4$) and $q3$ in between gates $g4$ and $g5$, corresponding to the circuit in Fig. 4 (d). It affects gates $g5$ and $g6$, which are marked in red. For each affected node in the dependence graph, to calculate its earliest start time, one needs to check each of its parent nodes' ending time, choose the maximum one, and use it as its own start time. In this example, added swap result in change in g5's start time as well as the change in g6'start time, and delays the entire circuit by 3 cycles. Here we assume each gate takes one cycle. We assume a swap is implemented using 3 CNOT gates and thus is 3 cycles.

The second mapping candidate shown in Fig. 15 (c) inserts one swap on logical qubits $q3$ and $q5$ placed right in front of $g5$. It results in no changes to the start/end cycles of the entire circuit, since there is slack on physical qubits $q3$ and $q5$. Obviously, it should choose the mapping candidate illustrated in Fig. 15 (c).

4.3.3 Navigating the Candidate Search Space

We use a priority queue based searcher for qubit mapping candidates. The search space consists of state nodes that represent possible mappings from logical qubits to physical qubits. A mapping can be represented as $\pi : \{q_1, q2, ..., q_n\} \rightarrow \{Q_1, Q2, ..., Q_n\}$. Applying swaps on top of a mapping can convert it into another mapping. Specifically, if we apply "$swap\ q_i, q_j$" on a certain mapping π_{old} and create the resulting mapping π_{new}, we will have $\pi_{new}[q_i] = \pi_{old}[q_j]$ and $\pi_{new}[q_j] = \pi_{old}[q_i]$. Given $F_{critical}$ and current mapping π, it starts searching the state space of all feasible mappings that satisfy $F_{critical}$. It picks a node to expand and enumerate all possible parallel one-step swaps as the node's successors.

We use a priority queue that is similar to that in [35]. Unlike the work by [35] where the search stops when the first state node that resolves all connectivity conflicts is retrieve from the priority queue, our search stops after m expansions since the mapping candidate with minimal swap count is found, or when the gate count of the mapping candidate that is just retrieve has less than or equal to k times more gates than the minimal swap count. We

set $m = 20$ and $k = 2$ such that the returned mapping candidate will have reasonable gate counts. After all mapping candidates have been retrieved, we rank them with respect to the metric of best slack utilization discussed above.

The priority queue based search returns a set of hardware-compliant mappings for which the gate count is bounded with respect to the minimal gate count by k times.

It will expand the state node of π by doing the following: It enumerates all possible one concurrent swap, two concurrent swaps, three concurrent swaps, all the wap up to $n/2$ concurrent swaps, where n is the number of qubits in the circuit; Each set of concurrent swaps lead to a new mapping, and the new mapping becomes a child of π. The total number of successors of π is up to $\binom{n}{1} + \binom{n}{2} + ... + \binom{n}{n}$. In this process, we only keep the nodes that at least have a swap that involves a logical qubit in $F_{critical}$ to prune the search space. All newly generated successors will be checked to see if it resolves qubit mapping conflicts in $F_{critical}$. Each successor of π can be further expanded in the same way. The search goes on until it finds a set of state nodes which meets our criteria. The state nodes would have the mapping that solves all conflicts. Exhaustive search in this space is prohibitive. We use a priority queue to guide the expanding of nodes in the frontier of the search tree. The nodes that need to be expanded are placed into the priority queue with respect to a cost function $f(x) = g(x) + h(x)$, with $g(x)$ representing the cost of the path from root node to x and $h(x)$ representing the estimate cost from x to target node. In our implementation, $g(x)$ is the number of swaps that have already incurred for state node x. The $h(x)$ estimates at least how many more swaps are still needed, which is approximated as the shortest physical distance of the qubits for at least one CNOT that is in the $F_{critical}$ set and hasn't be resolved. The heuristic function is similar to a classical A-star algorithm. Unlike A-star, our algorithm does not terminate when the mapping candidate with minimal cost function is achieve when d, i.e., when the mapping candidate with minimal swap gate count is achieved. Instead, we continue the search and keep expanding nodes from the priority queue up to a certain point when we have enough feasible mapping candidates. The output will be a list mapping candidates sorted with respect to the swap gate count in ascending order.

4.4 Evaluation

In this section, we evaluate our slack-aware swap insertion scheme (**SlackQ**) and compare it with the two state-of-the-art qubit mappers, respectively by [35] and [13].

4.4.1 Experiment Setup

Benchmark. We use 106 benchmarks from RevLib [29], IBM Qiskit [20], and ScaffCC [9]. **Baseline.** We compare our work with two best known qubit mapping solutions [35] (denoted as *Zulehner*) and the Sabre qubit mapper from [13] (denoted as *Sabre*). **Metrics.** We compare the execution time of the transformed circuits generated by different qubit mapping strategies. It is worth mentioning that our approach can take any gate latency as input parameters and generate transformed circuits based on the input. However, to make evaluation results as close to real machines as possible, we use the results from the study by [14]. In this study, different types of quantum architecture are investigated, and the studies reveal that two-qubit gates usually takes around twice as much time as single-qubit gates. Hence we assume single-qubit gates take 1 cycle and two-qubit CNOT gates take 2 cycles in our experiments. The time is reported as the total number of executed cycles. **Platform.** We use IBM's 20-qubit Q20 Tokyo architecture [13] as the underlying quantum hardware. The qubit mapping approach is implemented in C++.

4.4.2 Experiment Analysis

We categorize the 106 benchmarks into four categories. Benchmarks in the first category each has less than 200 gates, and we denote them as mini benchmarks. There are 22 mini benchmarks. The second category has benchmarks with 200 to 1,000 gates. We name this category as small benchmarks. There are 39 small benchmarks. The third category of benchmarks have 1,000 to 10,000 gates. We name it as medium benchmarks and there are 21 benchmarks in this category. The fourth category of benchmarks have 10,000 to 200,000 gates. We refer to it as large benchmarks and there are 24 benchmarks in this category. The results for mini, small, medium, and large benchmarks are presented in Fig. 16, Fig. 17, Fig.

18, and Fig. 19 respectively.

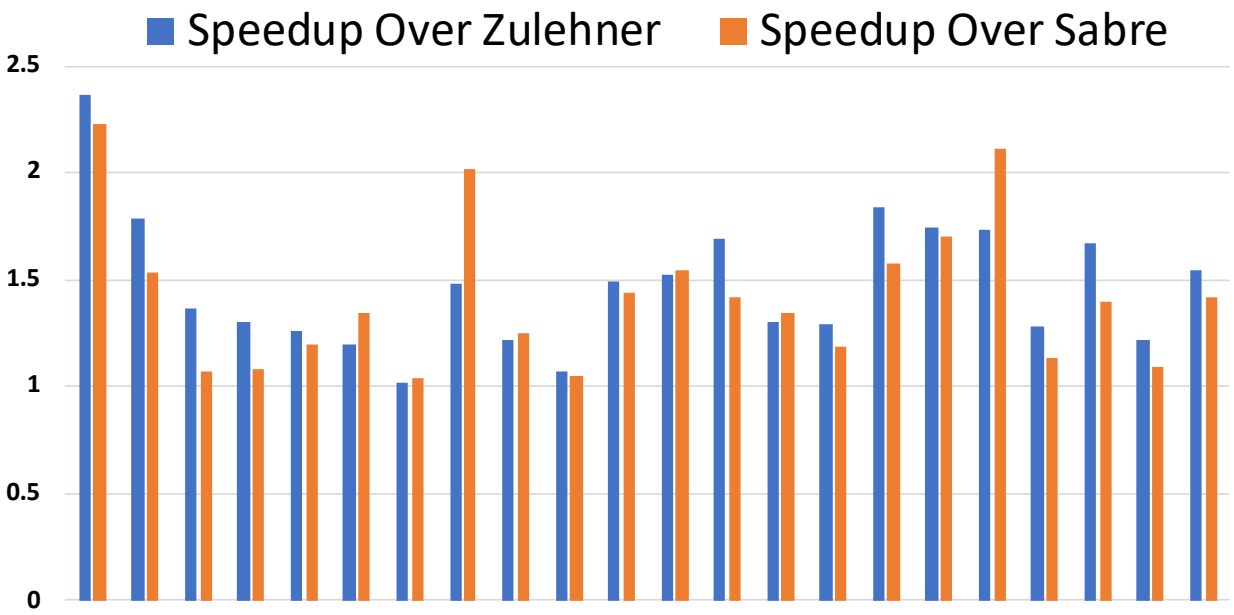

Figure 16: Speedup for Mini Benchmarks (< 200 gates)

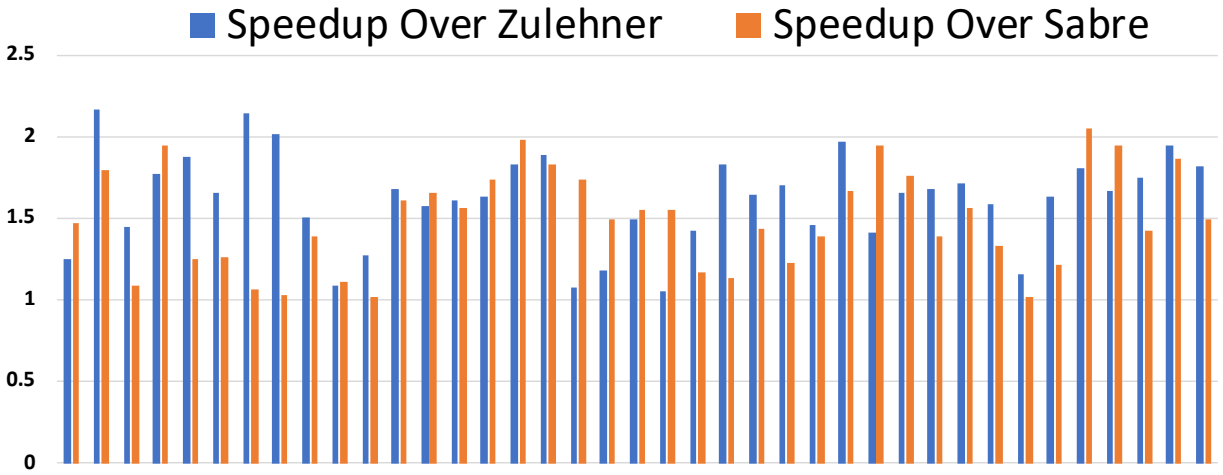

Figure 17: Speedup for Small Benchmarks (< 1000 gates)

It can be observed from the results that as the problem size scales, the performance improvement brought by SlackQ improves. For most benchmarks in the mini and small

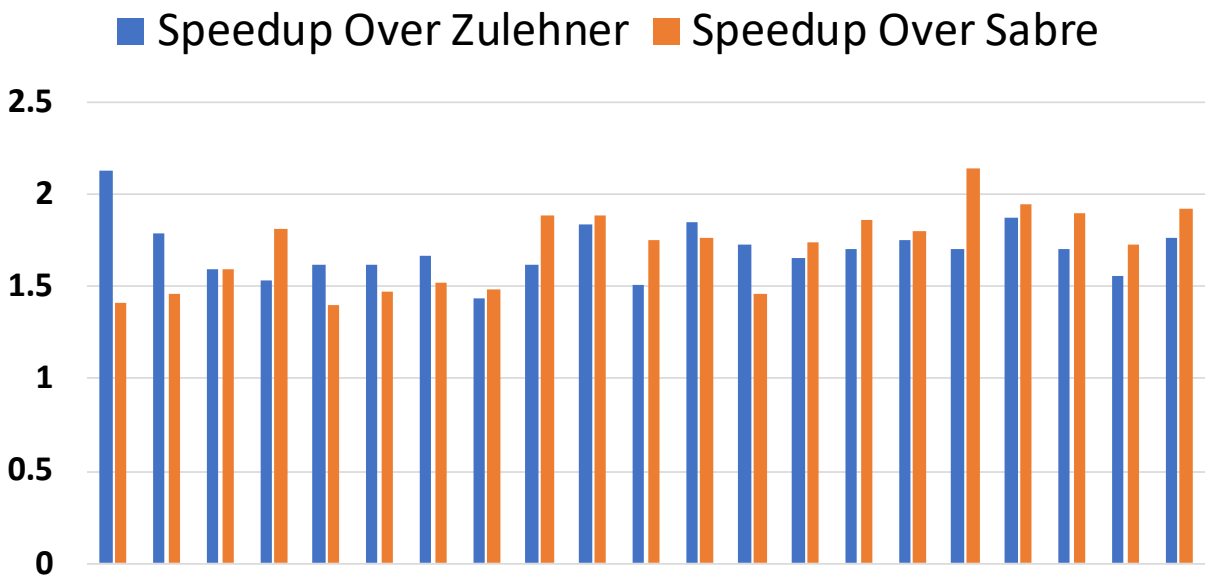

Figure 18: Speedup for Medium Benchmarks (< 10000 gates)

category, the speedup is between 1.1X and 1.5X. However, for the medium and large category, the speedup for most benchmarks is above or around 1.5X. The average speedup for mini benchmarks is 1.45X and for small, medium, and large benchmarks, the average speedup becomes 1.55X, 1.70X, and 1.86X respectively. The results show that our approach works well in general, and in particular for larger benchmarks.

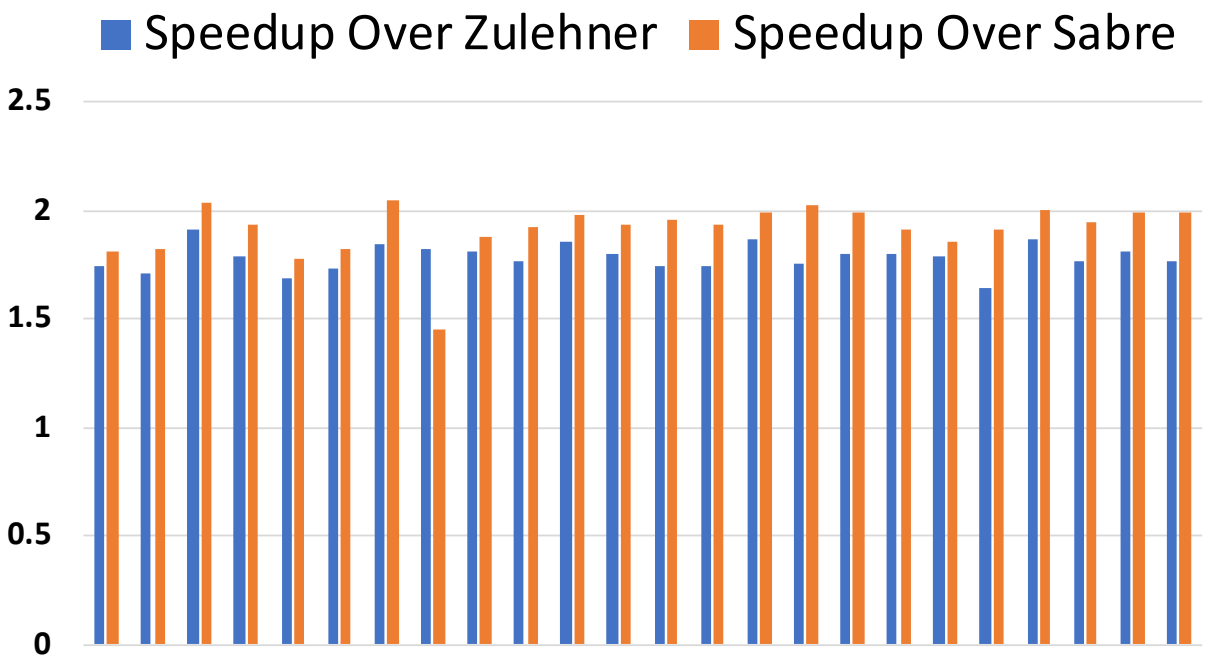

Figure 19: Speedup for Large Benchmarks ($<$ 200000 gates)

5.0 Time-Optimal Qubit Mapping

5.1 Introduction

In this section, we tackle the time-optimal qubit mapping problem. We first present a simple and effective model for representing the complete search space. The search space is constructed with respect to an input logical circuit and a bounded degree hardware coupling graph. We then propose a search algorithm that is complete and optimal. Our contributions are summarized as follows:

- We present the first theoretical model for time-optimal qubit mapping without any implicit constraints. The theoretical model can be flexibly extended to practical algorithms.
- We present a search framework based on our time-optimal model. It consists of space pruning, redundancy elimination, and comparative filtering. It significantly reduces the time complexity and makes time-optimal search feasible.
- We discovered time-optimal solutions for quantum fourier transformation (QFT) on both 1D and 2D nearest neighbor architecture (using our search framework). Our solution for 1D nearest neighbor architecture is the same as a manual solution reported by Maslov [15]. But our optimal solution for 2D architecture is for the first time reported. Interested readers for QFT solutions can refer to Section 5.7.1.1.
- We present a practical extension of our theoretical model. It prunes the branches in the search space unlikely to yield effective mapping solutions. Our practical implementation is not guaranteed to be optimal, however, it still outperforms state-of-the-art qubit mappers with speedups ranging from 0.99X to 1.36X, and on average 1.21x, over representative benchmarks from RevLib, IBM Qiskit, and ScaffCC.
- We implemented our search framework for both optimal and practical solutions. We open sourced the code at GitHub[1].

[1]Our implementation is made publicly available at the GitHub repository: https://github.com/time-optimal-qmapper/TOQM.

5.2 Motivation

It is non-trivial to achieve time optimality for the qubit mapping solution. As there are many possible permutations of circuit gates and inserted swaps, the search space is large even for small input. We use quantum fourier transform (QFT) to demonstrate the challenges in finding a time-optimal solution.

QFT is at the heart of integer factorization [23]. It serves as the basis for many important quantum algorithms. QFT is also one of the most challenging benchmarks for qubit mapping. It is because QFT requires all-to-all qubit connection. Each qubit needs to interact with every other qubit in the program.

The QFT circuit has a regular pattern. It has n qubits and $n(n-1)/2$ generic two-qubit gates. We follow the convention by Maslov *et al.* [15] for describing a QFT skeleton circuit. A concrete QFT circuit includes Hadamard (H) gates and controlled phase gates. Following this convention, a single-qubit is absorbed into a nearby two-qubit gate to form a generic two-qubit gate. Any two-qubit gate can be efficiently implemented and we assume they have the same latency as in [15]. Hence the entire circuit is described using generic two-qubit gates of the same latency. We denote the original two-qubit computation gate as GT gates for the simplicity of discussion.

Each logical qubit q_i interacts with every other logical qubit q_j, denoted as $GT(q_i, q_j)$ where $j \neq i$. The logical QFT circuit with 6 qubits is shown in Fig. 20 (b).

Let's look at the simple linear nearest neighbor (LNN) architecture as shown in Fig. 20 (a), where the capital case Q represents physical qubits. In this architecture, the physical qubits are arranged on a straight line. Each physical qubit only communicates with the qubit on its left or right. However, the logical QFT circuit requires each qubit to interact with every other qubit. The logical circuit in Fig. 20 (b) cannot directly run on Fig. 20 (a) no matter which initial qubit mapping is used.

Due to the all-to-all qubit interaction pattern in QFT, it is hard to obtain an effective qubit mapping on LNN, let alone an optimal solution. Using our search algorithm described in the next section, we find an optimally transformed QFT-6 circuit. The solution is shown in Fig. 20 (c). Although the circuit size is small, it is not difficult to see a butterfly pattern

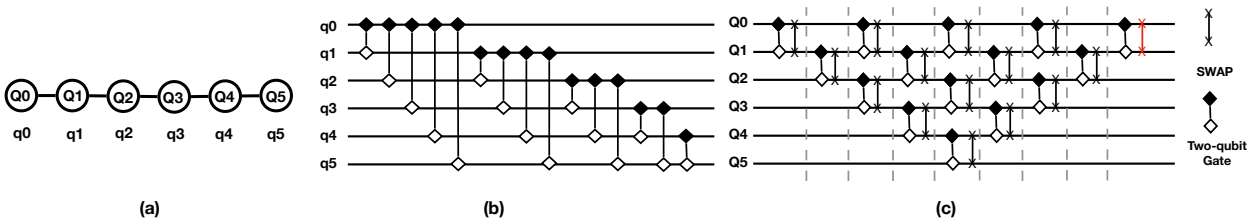

Figure 20: Adapting QFT logical circuit to LNN architecture: (a) 6-qubit LNN and initial mapping, (b) logical qft-6 circuit where each line represents a logical qubit q_i, and (c) physical qft-6 circuit where each line represents a physical qubit Q_i. The last swap gate in red in (c) is added for showing the symmetric pattern. Our solver does not have the swap in its returned solution.

in the transformed circuit represented using physical qubits. With further analysis in Section 5.7, this pattern can generalize to larger QFT circuits and ensure linear depths.

We note that our solution for QFT in LNN is the same as the manual solution by Maslov [15]. However Maslov [15] cannot find an optimal solution manually for 2D architecture due to the complexity of the architecture, while our mapper can, as described below.

Another prototypical hardware architecture is a structured two-dimensional lattice. Each qubit has up to 4 neighbors. For instance, IBM Melbourne architecture has a 2×N grid like topology, as shown in Fig. 21. Maslov [15] did not provide a manual solution for the 2D architecture. But the paper predicts a lower bound depth of $3n + \mathcal{O}(1)$. Using our algorithm, we found a generalizable pattern for arbitrary size QFT. Our generalized solution has $3n + \mathcal{O}(1)$ depth which matches the lower bound depth provided by Maslov [15]. The asymptotic term is only at the constant component. Hence our solution is not only confirmed to be optimal for small inputs but also for arbitrary size inputs. Our solution is presented in Section 5.7.

Time-optimal solutions, even for small size circuit, could be very useful. If an optimal solution for a logical circuit has recurring pattern, we can obtain the optimal solution for small-size inputs, and use that to deduce the generalized solution. The solutions to the

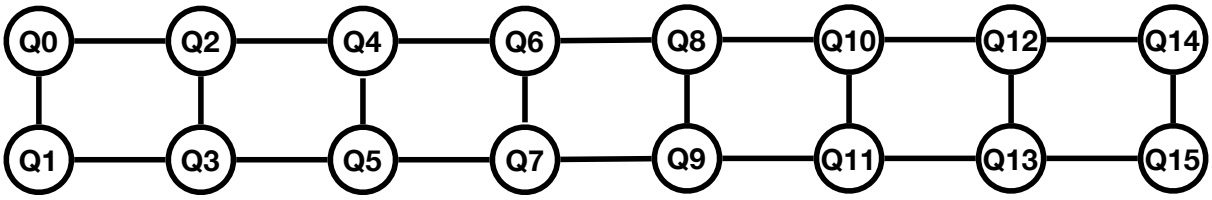

Figure 21: 2×N Qubit Coupling Graph

QFT program on two types of architectures demonstrate the effectiveness of our algorithm in discovering such patterns for circuit families. For finding a solution to QFT-6 on LNN, our mapper takes less than 1 second. For finding a solution to QFT-8 on 2D architecture, it takes less than 30 seconds. In the following, we will describe our model and search algorithm in details.

5.3 Time-Optimal Mapping Framework

In this section, we define a time-optimal model for the qubit mapping problem. We first define the search space. Then we present a guided search framework. We prove the optimality of the framework in Section 5.4 and Appendix 5.5.

5.3.1 Search Space

We discover that any valid execution of a circuit can be partitioned into a sequence of states. We refer to *cycle* as a unit of time. A circuit is decomposed with respect to its state at each cycle. A *state* of the circuit represents a qubit mapping and the specific busy/idle status of each qubit. The status of a qubit is represented as which gate it is executing if the qubit is busy, or which gate it has just completed if the qubit is idle. An example is shown in Fig. 22 (a).

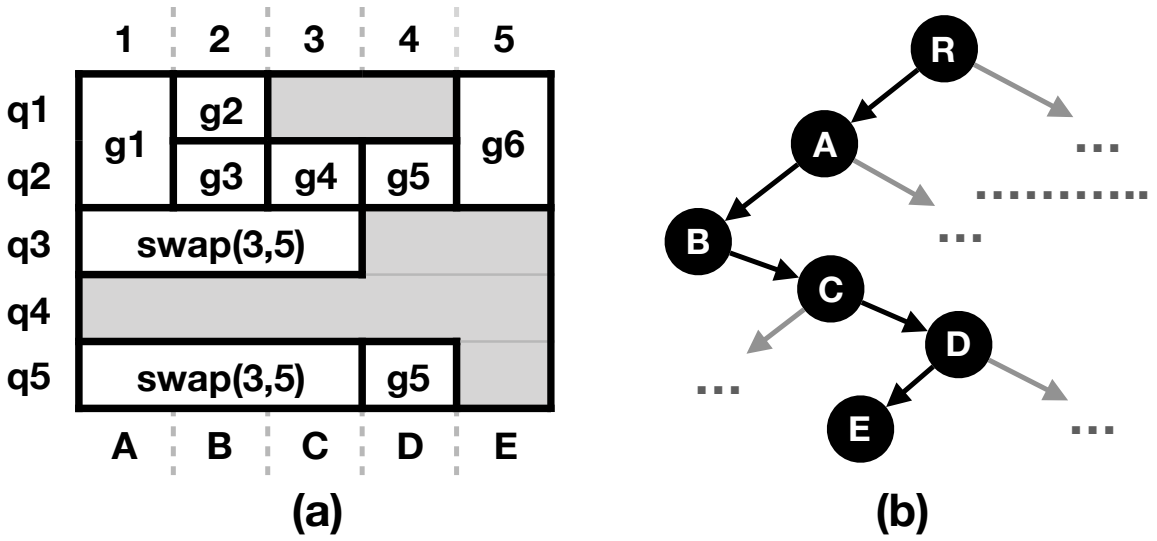

Figure 22: (a) Cycle-by-cycle partition of a 5-cycle circuit, (b) the search path representing the execution in (a). R is the root node. E is a terminal node.

We define our search graph. Each node in the search graph represents a state of the circuit (which also includes a cycle number).

A node expands into one or multiple children nodes. Each child node represents a possible state the circuit will be in at the next cycle. A child node's cycle number is its parent's cycle number + 1. The executing gate(s) of a child node in its own cycle must satisfy the constraints of the qubit coupling and gate dependence.

By enumerating all possible combinations of gates and swaps in next cycle, a node can determine its complete set of child node(s). The circuit makes progress when a gate in the original circuit executes. The circuit updates its qubit mapping when an inserted SWAP finishes its execution.

We let the *root node* be the initial state of the circuit at cycle 0 where all qubits are idle and no gate in the original circuit has been scheduled.

We say a node is *terminal*, if all logical qubits have completed their gates in the original circuit, and the last gate of all just finished at the cycle of this node.

The goal is to find a terminal node such that its path to the root node is the shortest.

There is only one root node but there could be more than one terminal node. An optimal solution corresponds to an optimal terminal node. The path from the root node to a terminal node represents a transformed circuit.

Fig. 22 (b) shows a valid path from the root node to a terminal node. It corresponds to the example circuit in Fig. 25 and the execution in Fig. 22 (a). The node E is a terminal node, and R is the root node.

5.3.2 Guided Search Framework

Since there is a finite number of gates and physical qubits, there is only a finite number of combinations of gates and swaps that can be executed simultaneously at any given moment. Thus each node in the search graph has only a finite number of child nodes. An exhaustive breadth-first search (BFS) algorithm is guaranteed to find the optimal terminal node in finite time. However, BFS search is not realistic as it is brute-force and search space is large even when the input is small.

We present a guided search framework. It uses a priority queue to keep track of the nodes that need to be expanded. We show our framework in Fig. 24. It first initializes the priority queue to be empty and then inserts the root node into the priority queue.

Next it extracts a node from the priority queue, expands it, and push new nodes into the queue. The three steps repeat until a terminal node is for the first time popped out of the priority queue. It is based on A* search. Each node is associated with a priority (cost). We set the priority (cost) function to be admissible to ensure the the A* search returns an optimal solution. The detailed definition of the priority function and the proof for optimality is shown in Section 5.4 and Appendix 5.5.

Using A* guarantees optimality. We also ensure efficiency by space pruning techniques. It is often possible that the circuit reaches the same state using different combinations of gates and swaps. If the only difference between two nodes is their timestamps, then the slower one of the two nodes can be dismissed without affecting optimality. We prune these nodes in the Expanding component and the Filtering component in Fig. 24.

The node expander is responsible for expanding the node extracted from the priority

queue. After expansion, the children nodes of the expanded nodes will be collected. We apply several restrictions on top of these children nodes. Below are the criteria that we want these children nodes to meet:

- **Coupling** One node needs to satisfy the current qubit coupling constraints. Thus we first filter out the nodes that cannot satisfy coupling constraint.
- **Dependency** The gates scheduled in a child node should also have their dependency resolved. This means all parent gates of any newly added active gate should be finished prior to the child node's cycle.
- **Redundancy Check** We check two kinds of redundancy. First we check that whether at least one active gate of the child node depend on some gate in the parent node's active gates (except that the active gate is the first gate on a qubit). This criteria is based on the fact that any non-depending gate in the child node could have been placed earlier in the parent's sibling nodes.

  Second we check if there are cyclic swaps, which means identical swaps applied consecutively to the same two qubits. It does nothing but cancel out the previous swap effect.

We have a filtering pass on the expanded nodes that meet the criteria we listed above. In this filtering pass, every expanded node has its hash value calculated based on its current qubit mapping (assuming all active swaps have taken effect). For each node, we look at its hash value and compare it to all the previous nodes (in the priority queue) that have the same hash value. When comparing one expanded node with all the previous nodes that have the same hash value, we check for equivalence and relative goodness.

- **Equivalence Check** We check if this expanded node is equivalent to any previous node. Equivalence means the two nodes will have finished the same set of gates assuming all active gates have taken effect. The active gates on same qubits will finish in the same exact cycle. The current cycle must be the same. If we find this equivalence, we filter out this expanded node. An example is shown in Fig. 23 (a).
- **Comparative Analysis** If there's no equivalence found, we then switch to comparative analysis between the expanded node and the previous node with same hash value. We

look at the projected finish cycle on each qubit (if the qubit is busy). If, for all qubits, the expanded node has fewer finished gates but longer projected finishing time, we know that this expanded node is less desirable than this previous node. We then filter out this expanded node as itself and its sub-tree will not lead to a better solution than one previous node. An example is shown in Fig. 23 (b).

5.4 Optimality Guarantee

5.4.1 Admissible Cost Function

We assign each search node v in the search graph a cost $f(v)$. The cost $f(v)$ consists of two parts $g(v), h(v)$ such that

$$f(v) = g(v) + h(v) \tag{1}$$

$g(v)$ represents the length of the path from the root node to v, which is the number of cycles already executed. $h(v)$ is a heuristic cost function as a lower bound on the number of cycles from v to any terminal node.

The circuit now contains two parts with respect to a search node v: (1) the part that has been scheduled due to v and v's ancestor nodes, and (2) the remaining circuit that hasn't been scheduled. The cost $g(v)$ is for part (1), and cost $h(v)$ is for part (2). An example is shown in Fig. 25.

The cost for part (1) is trivial as a search node contains a time stamp. The cost $h(v)$ for part (2) is more complex.

Our heuristic function $h(v)$ is defined with respect to the dependency graph G_{rem} and the qubit mapping π_{rem} of the remaining circuit. If the node v contain any active swap that hasn't been completed, we assume swap has taken effect for calculating π_{rem}. As it is not difficult to obtain remaining dependence graph from search node v and v's ancestor nodes, we omit the details here.

Let $G_{rem} = (V_{rem}, E_{rem})$ be the dependency graph of the gates in the remaining circuit. Thus V_{rem} consists of the gates that haven't been scheduled or have been scheduled but

executed in part, and G_{rem} is a directed acyclic graph. We define the heuristic function $h(v)$ via induction on G_{rem}.

Let $g_1, g_2, \cdots, g_n$ be a topological ordering on the vertices (gates) in G_{rem}. For each gate g,let $len(g)$ be the number of cycles that g needs to execute (if $len(g)$ is partially executed, $len(g)$ is equal to the length of the unexecuted part). We will define $t_{min}(g)$, which is a lower bound on the time when g begins to execute.

Base case: If g_i is a single-qubit gate that has no predecessors in G_{rem}, then $t_{min}(g_i) = 0$, meaning, g_i may begin to execute immediately.

Inductive case: Suppose gate g_i depends on gate $h_1, h_2, \cdots$, then g_i can only begin after these gates have finished. First let $u = \max_i t_{min}(h_i) + len(h_i)$. We must have $t_{min}(g_i) \geq u$. If g_i is a single-qubit gate, then we simply take $t_{min}(g_i) = u$.

If g_i involves two qubits, then we also have to consider the delay caused by inserting SWAP gates. Suppose that gate g_i involves qubits q_a and q_b. Let H_a be the set of gates (in G_{rem}) that involve q_a and are direct or indirect predecessors of g_i. Similarly we may define the set H_b. Let T_a be the sum of the number of cycles needed by the gates in H_a. Then $u - T_a$ represents the "slack" space that may be used by qubit q_a to perform SWAPs. Similarly we define T_b.

Now, in the current qubit mapping π_{rem}, let $d(a, b)$ be the shortest distance between qubit q_a, q_b. Then we need at least $d(a, b) - 1$ SWAPs in total on qubit q_a with some other qubit, and on q_b with some other qubit. We are only concerned with the delay on q_a and q_b but not the delay on the qubit which they have SWAP with. Suppose we place r SWAPs on q_a and s SWAPs on q_b, then the delay on q_a is $\max\{r \cdot len(SWAP) - (u - T_a), 0\}$, and the delay on q_b is $\max\{s \cdot len(SWAP) - (u - T_b), 0\}$. We have $r + s \geq d(a, b) - 1$. We let $r + s = d(a, b) - 1$ as it will not increase the cost, and enumerate all possible combinations of (r, s) with $r = 0 \ldots d(a, b) - 1$. We take the pair (r, s) that minimizes the larger of the two delays. Let u' be the minimized delay, then we take $t_{min}(g_i) = u + u'$.

Definition 5. *(Heuristic cost function $h(v)$) Having computed $t_{min}(g)$ for each gate g, we define the heuristic cost $h(v)$ to be*

$$h(v) = \max_{g \in V_{rem}} t_{min}(g) + len(g)$$

An example of calculating the cost function with respect to a CNOT gate is shown in Fig. 26. The state node F we focus on is in Fig. 26 (a), where it has already scheduled the $g1$ and started $SWAP\ Q_4, Q_5$. We assume each single original gate in the circuit takes 1 cycle and each swap takes 3 cycles.

The remaining dependency graph of the circuit is shown in Fig. 26 (c). Since g_1 is completed, s_{45} has executed in part, G_{rem} consists of the remaining gates and part of s_{45}. First we set $t_{min}(g_2) = t_{min}(g_3) = 0$ as they have no predecessors. $t_{min}(s_{45})$ is also 0. Then $t_{min}(g_4) = 1$ as it depends on the single-cycle gate g_3.

The qubit mapping π_{rem} after gate s_{45} is shown in Fig. 26 (b). g_5 is not immediately executable, as q_2 and q_5 are not adjacent to each other. The shortest path between them is $Q_2 \to Q_3 \to Q_4$ with a distance of $d = 2$. Now at least $d - 1 = 1$ total SWAP needs to be inserted in total on q_2 with some other qubit, and on q_5 with some other qubit.

Suppose g_5 could be executed immediately, then it would have got start time as $u = 2$. On the part of qubit q_2, gate g_5 depends on g_3 and g_4, $2 - 2 = 0$, so there's no slack space on q_2. On the part of qubit q_5 the predecessor is part of s_{45}, which is 2 cycles. Therefore, the slack space on qubit q_5 is 0 cycle too. Hence $t_{min}(g_5) = 2 + 3 = 5$, as inserting SWAP on either q_2 or q_5 introduces a delay of 3.

Finally, $t_{min}(g_6) = 6$ since q_1, q_2 are adjacent, and the cost for search node F is 8.

Another example is for the search node A in Fig. 26 (e) where g_1 and $swap(3, 5)$ is scheduled in cycle 1. The induced qubit mapping is in Fig. 26 (f). Since each CNOT in remaining circuit can be immediately executed, the cost of A is the critical path of the remaining circuit plus 1, which is 5. Therefore search node A is better than search node F as its cost is lower.

It is tempting to assume that, since two qubits can move towards each other by performing SWAPs simultaneously, the optimal position for two qubits to meet is always at exactly the middle of the shortest path between them. Hence one may simply need to insert dummy gates of length $(d - 1) \cdot len(SWAP)/2$. This is not true as it is oblivious to the slack in the original circuit that can potentially absorb SWAP overhead. An example is demonstrated in Fig. 27. Here we assume that, in the initial mapping, the distance between the two qubits is 5, so at least 4 SWAPs are needed. Suppose we let the two qubits meet in the middle.

Assume a SWAP takes 2 cycles. Then we need to insert a 4-cycle delay for each qubit. The length of the critical path is 8 cycles. However, suppose we let the first qubit do 1 SWAP, and let the second qubit do 3 SWAPs. Then the length of the critical path becomes 6 cycles. This demonstrates the necessity of trying all possibilities of splitting delay to two participating qubits when calculating the cost.

5.4.2 Optimality

Our heuristic cost function $h(v)$, in effect, computes a lower bound on the time needed by the remaining circuit through two different ways, and takes the larger of the two bounds. The first way considers the immediate predecessors of each gate. The second way considers the indirect predecessors and potential SWAPs. We prove rigorously it is a lower bound time of the remaining circuit in Appendix through Lemma 5.5.1. We relegate the details to the Appendix.

Lemma 5.4.1. *The cost function defined in Eq. 1, $f(v) = g(v) + h(v)$ is admissible.*

Proof. We've shown in Lemma 5.5.1 that the heuristic function $h(v)$ never overestimates the execution time of the remaining circuit. And also g(v) is the number of cycles already executed till node v. Therefore the cost function $f(P)$ is admissible. □

Theorem 5.4.2. *The A-star search algorithm we proposed here is complete and optimal.*

Proof. It has been shown that A-star search will find an optimal solution if the problem satisfies the following conditions:

1. Each node in the search graph only branches into a finite number of child nodes. This is true, because the set of gates that can possibly execute at any given moment is finite.
2. Each transition increases the cost of the path. This is true for our problem, because each gate takes at least one cycle to execute, and so increases the cost by at least 1.
3. The heuristic cost function is admissible. This is proven in the lemma above.

Thus our algorithm satisfies all conditions for optimality. □

5.4.3 Initial Mapping

Our optimal mapper works in two different modes: (1) It finds an optimal solution given an input initial mapping, and (2) It finds both optimal initial mapping and the transformed circuit.

For (1), it is trivial. Our previous technical description already addressed how to find the solution after initial mapping is given. For (2), we start with a random initial mapping. Then we allow at most d consecutive cycles of pure swaps before any original gate is scheduled, which represents a search for initial mapping. In the meantime, we modified our cost function such that the pure swap cycles at the beginning are not counted, as if the circuit starts at some initial mapping resulted from these consecutive pure swap cycles. The parameter d is defined as the maximum of the longest path (without going through any node twice) between any two qubits in the physical architecture. It is because one mapping layout could be transformed into another mapping layout with at most the number of cycles equivalent to the maximum longest path length in the graph, assuming swaps on disjoint qubits run in parallel.

For the pure swap cycles, our hash filter is also applied, ensuring that no unique initial mapping will appear twice in the priority queue. It is because one initial mapping can be achieved through different ways of swap combinations.

5.5 Proof of Optimality

Lemma 5.5.1. *For each state node P, the heuristic cost function $h(P)$ is a lower bound of the length of all paths from P to a terminal node in the search graph.*

Proof. Let $G_{rem} = (V_{rem}, E_{rem})$ be the dependency graph of the remaining circuit. For each gate $g \in V_{rem}$, let $t^*_{min}(g)$ be the length of the actual shortest path from P to any state node where gate g has already been scheduled to execute. In the next few paragraphs we will prove that $t_{min}(g) \leq t^*_{min}(g)$ for every gate g. Let X be any terminal node in the search graph, and let t^* be the distance between P and X. Since each gate must finish execution at state X,

for each gate g we have $t^* \geq t^*_{min}(g) + len(g)$. Together, for each gate g we have

$$t_{min}(g) + len(g) \leq t^*_{min}(g) + len(g) \leq t^*$$

Thus $h(P) = \max_g t_{min}(g) + len(g) \leq t^*$. This shows that $h(P)$ is a lower bound on the length from P to any terminal node.

Base case: If g is a single-qubit gate that has no predecessors, then $t^*_{min}(g) = 0$, since it can be scheduled to execute immediately. Note that our definition of G_{rem} includes gates which are only partially executed, so gates which have no predecessors do not need to wait current gates to finish.

Inductive case: Suppose gate g depends on a number of other gates. If g is a two-qubit gate then we also have to consider potential SWAPs before g, even if g has no apparent predecessor in G_{rem}. We will give two lower bounds on $t^*_{min}(g)$ and show that $t_{min}(g)$ is equal to the larger of the two bounds.

The first bound is derived from the immediate predecessors of g. If $h_1, h_2, \cdots$ are the immediate predecessors of g in G_{rem}, then g cannot be scheduled until all these gates have finished. Thus $t^*_{min}(g) \geq t^*_{min}(h_i) + len(h_i)$ for each gate h_i. This gives the first bound

$$t^*_{min}(g) \geq \max_i t^*_{min}(h_i) + len(h_i)$$

The second bound comes from gates which operate on the some same qubit as g. Suppose g involves two qubits q_a, q_b. Let H_a be the set of all gates on q_a which are direct or indirect predecessors of g. Since they all operate on the same qubit, they must be executed in the order they appear in the circuit. Suppose this ordering is $h_1, h_2, \cdots, h_n$. We have

$$t^*_{min}(h_2) \geq t^*_{min}(h_1) + len(h_1)$$

$$t^*_{min}(h_3) \geq t^*_{min}(h_2) + len(h_2) \geq t^*_{min}(h_1) + len(h_1) + len(h_2)$$

$$\vdots$$

By induction we have $t^*_{min}(g) \geq t^*_{min}(h_1) + \sum_i len(h_i) \geq \sum_i len(h_i)$. Now suppose we place r SWAPs before gate g on qubit q_a. They also cannot execute simultaneously with any of the gates in H_a. Thus

$$t^*_{min}(g) \geq r \cdot len(SWAP) + \sum_{h \in H_a} len(h) = u_1$$

Similarly we may define the set H_b, and if we place s SWAPs on q_b before g we have

$$t^*_{min}(g) \geq s \cdot len(SWAP) + \sum_{h \in H_b} len(h) = u_2$$

Let $d(a, b)$ be the distance between qubit q_a, q_b in the qubit mapping π_{rem}. Then at least $d(a, b) - 1$ SWAPs are needed before gate g, so $r + s \geq d(a, b) - 1$. Since u_1, u_2 increases linearly with r, s, we fix $r + s = d(a, b) - 1$ to minimize them. We choose the pair (r, s) such that $\max\{u_1, u_2\}$ is minimized. We take the minimized $\max\{u_1, u_2\}$ to be the second bound.

In the above we have derived two lower bounds on $t^*_{min}(g)$. If we compare them with the definition of $t_{min}(g)$, we see that $t_{min}(g)$ is exactly equal to the larger of the two bounds. Therefore $t_{min}(g) \leq t^*_{min}(g)$. This finishes the proof. □

5.6 Multiple Optimal Solutions

In certain benchmarks, multiple depth-optimal solutions exist. Our tool can be easily tuned to find all of them. Our algorithm waits until the first optimal solution is found. Typically in A*, one terminate the algorithm as long as the first solution is found. But it is not necessary. We then record the depth of the first optimal solution, and continue to run the extract-expand-push process as shown in Fig. 24 and report more solutions. We stop reporting solutions whenever an extracted node from the queue suggests a solution with a larger depth than the optimal one. At this time, our algorithm has found all solutions.

We need multiple optimal solutions because not all optimal solutions for small circuits have a recurring pattern. Hence, it is necessary to generate all optimal solutions and discover the one with recurring pattern to generalize to larger circuits. For instance, for QFT-8 on

2×4 architecture (without allowing CNOT and swap at the same cycle) only one solution among the eight optimal solutions shows the pattern in Fig. 32.

Another issue that arises when we try to manually generalize a solution is that the solution circuit might need slight transformation. It is possible that an optimal solution returned by our tool has cancelable swap gates (which usually involves multiple qubits and cannot be automatically found like cyclic swaps). However, it is easy to discover them when visualizing the small solution circuit.

Further, certain gates can be scheduled earlier or later without affecting the overall mapping or depth. We show an example in Fig. 33. As can be seen in step (5), the two-qubit gate for $\{q_2, q_3\}$ can be moved to step (6) without affecting the overall depth. Similarly, in step (11), the two-qubit gate or $\{q_4, q_5\}$ can be moved to step (12). This transformation is inferred from step (3) and (9). At this point, we are doing the generalization/inference of recurring patterns manually, but this could potentially done automatically. We leave it as our future work.

Last but not least, if a swap is followed by a two-qubit gate, the two-qubit can be moved in front of the swap by reversing the control and target, and the transformed circuit is equivalent. Similarly when a two-qubit is followed by a swap. We show a solution for QFT-6 on LNN in Fig. 34, where if in layers L2 to L8, the order of swap and two-qubit gate can be swapped to be consistent with L1 and L9, the entire solution is the same as we show in Fig. 20.

5.7 Analysis

5.7.1 Exact Analysis

When the number of qubits is small, our algorithm can find (a set of) exact optimal solution(s). This is helpful in the applications where optimal solution has a recurring pattern. We use a two-step approach. We first find optimal solution(s) for small inputs. Then we generalize the solution(s) to larger inputs. We demonstrate it using QFT program in Section

5.7.1.1. We also show the optimal results for small logical reversible circuits in Section 5.7.1.2.

5.7.1.1 Optimal QFT Mapping

Recall that QFT has a regular structure, with n qubits and $n(n-1)/2$ generic two-qubit gates (Section 5.2). QFT has an all-to-all qubit interaction pattern. Every two qubits need to interact. The QFT circuit with 6 qubits is shown in Fig. 20.

We present a different representation of the QFT circuit in Fig. 28 (a) to facilitate discussion. As gates that operate on non-intersecting qubits commute, a QFT program can run in linear depth if the underlying architecture is fully connected. Fig. 28 (a) is equivalent to the circuit in Fig. 20 (b) except that it is organized into parallel layers such that each layer consists of concurrent two-qubit gates, and the affine loop representation is shown in Fig. 28 (b).

Recall that in LNN, the physical qubits are arranged on a (conceptual) straight line, and each qubit may only interact with the qubit on its left or right. For QFT, the logical qubits are initially mapped according to its natural order such that logical qubit q_0 is mapped to the leftmost physical qubit and q_5 is mapped to the rightmost physical qubit as shown in Fig. 29 (step 0).

The LNN architecture is often considered as a good approximation to what a scalable quantum architecture may be. If a circuit can be adapted well to LNN, it typically can be adapted to other architectures represented by a bounded degree graph [15].

Our search algorithm finds an optimal solution[2] for QFT with 6 qubits, visualized in Fig. 29. We next show that the discovered pattern for 6-qubit QFT can be generalized to n-qubit QFT.

The qubit mapping changes every two consecutive steps (from step 0). In every two consecutive cycles, a set of GT gate(s) first operate on some qubits, then a set of swaps on exactly the same qubits. At the end of these steps, the layout of the logical qubits are

[2]There might be multiple optimal solutions, but not all of them have a recurring pattern. However, our algorithm can be adapted to find all optimal solutions such that the solution with recurring pattern can be easily obtained. On the other hand, even some solutions do not have a strict recurring pattern, but the pattern can be inferred by performing some transformation based on gate commutativity or cancel-able swaps. We demonstrate this in Appendix 5.6.

reversed such that q_5 is placed at the left end of the LNN, and q_0 is placed at the right end of the LNN. Each qubit first shifts to left until it hits the left end of the LNN, and then to right until it reaches its destination.

To generalize this, we assume a sequence of logical qubits on the chain at a cycle m where m is even and $m/2$ is also even, and we let $i = m/2$, the qubit placement from left to right is

$$q_{\frac{i}{2}}, q_{\frac{i}{2}+1}, q_{\frac{i}{2}-1}, q_{\frac{i}{2}+2}, ..., q_i, q_0, q_{i+1}, q_{i+2}, q_{i+3}, q_{i+4}, ..., q_{n-1}$$

At the step m, it is a parallel computation stage of two-qubit gates. We have GT(q_0, q_{i+1}), GT(q_1, q_i), ..., GT$(q_{\frac{i}{2}}, q_{\frac{i}{2}+1})$ scheduled. Then at the step $m+1$, we have SWAP(q_0, q_{i+1}), SWAP(q_1, q_i), ..., SWAP$(q_{\frac{i}{2}}, q_{\frac{i}{2}+1})$. For each pair of qubits, their subscript adds up to $i+1$, which is $m/2+1$.

At the step $m+2$ where m is even and now $(m+2)/2$ is odd, we still let $i = m/2$. The qubit placement from left to right is:

$$q_{\frac{i}{2}+1}, q_{\frac{i}{2}}, q_{\frac{i}{2}+2}, q_{\frac{i}{2}-1}, ..., q_{i+1}, q_0, q_{i+2}, q_{i+3}, q_{i+4}, ..., q_{n-1}$$

At step $m+2$, the operations are GT(q_0, q_{i+2}), GT(q_1, q_{i+1}), ..., GT$(q_{\frac{i}{2}}, q_{\frac{i}{2}+2})$. At step $m+3$, it performs parallel swaps: SWAP(q_0, q_{i+2}), SWAP(q_1, q_{i+1}), ..., SWAP$(q_{\frac{i}{2}}, q_{\frac{i}{2}+2})$. For each pair of qubits, their subscript adds up to $i+2$, which is $(m+2)/2+1$ or $\lfloor (m+3)/2 \rfloor + 1$.

Now the pattern is clear from Fig. 29. It repeats the two above sets of operations, and each pair of qubits that perform GT or swap their subscripts sum up to $\lfloor m/2 \rfloor + 1$ if m is the iteration number. The generalized strategy can be applied to QFT with arbitrary number of qubits. We describe it using an affine loop in Fig. 31 (a).

Note that the swap gate(s) in the last step is not necessary. But we add it because it fits into the pattern, and also that after the last swap, the physical connectivity graph of the logical qubits is isomorphic to the one right before step 0.

Our generalized solution for QFT on LNN is the same as that found by Maslov [15]. However, that solution is found manually and the author proved the solution is at most a constant factor away from the optimal solution. Our qubit mapper at least confirmed that this solution is optimal for small input size of QFT.

Now we describe our generalized solution for QFT on $2 \times N$ architecture (where $N = n/2$). Maslov [15] does not report a generalized solution for this, but predicts a lower bound of $3n + O(1)$ circuit depth. Our solution is $3n + O(1)$, which is the same (asymptotically at the constant component). This is discovered for the first time as far as we can tell for QFT on a 2D architecture.

We visualize the solution of QFT-8 on a 2x4 architecture in Fig. 30. We denote the physical qubit on the j-th column and the i-th row as $Q_{i,j}$. Initial placement is a column major order such that $q_{2j+i} \rightarrow Q_{i,j}$ as shown in step (1) of Fig. 30.

With *column major* order, it takes 17 cycles, and the pattern is generalizable. We also tried the row major order, it takes 21 cycles, and the pattern is not generalizable. We only present the result of column-major initial qubit placement here. It is worth mentioning that we also tried the version which does not allow concurrent swap and computation gates to run. It takes 19 cycles and is generizable too. The generalized solution is shown later in this section.

With some analysis, it can be shown that the $2 \times N$ solution is a non-trivial extended version of $1 \times n$. If we look at the qubit layout at steps (2), (5), (8), (11), (14), and (17) in Fig. 30, placement of pairs of qubits on ($Q_{0,i}$, $Q_{1,i}$) resembles placement of qubits on Q_i in LNN as the circuit makes progresses.

We let every three steps starting from step (2) in Fig. 30 form one iteration (iteration is indexed from 0). Let iteration number be i. If i is even, at the first step of iteration i and at the top row of the qubit layout, the placement of qubits is as follows:

$$q_{2(\frac{i}{2})}, q_{2(\frac{i}{2}+1)}, q_{2(\frac{i}{2}-1)}, q_{2(\frac{i}{2}+2)}, .., q_0, q_{2(i+1)}, q_{2(i+2)}, .., q_{2(N-1)}$$

We present it on purpose such that the subscript is a multiply of 2 and a number (the number need not to be integer).

The placement of logical qubits on the bottom of row is similar except that q's subscript increases by 1.

At the first step of the iteration i, swap and GT are performed simultaneously such that GT is on top row, and swap on the bottom row. GT is on even-index qubits, and swap on odd-index qubits.

For GTs, they are on $\{q_{2(\frac{i}{2})}, q_{2(\frac{i}{2}+1)}\}$, $\{q_{2(\frac{i}{2}-1)}, q_{2(\frac{i}{2}+2)}\}$, ..., $\{q_0, q_{2(i+1)}\}$. And for swaps, they are on $\{q_{2(\frac{i}{2})+1}, q_{2(\frac{i}{2}+1)+1}\}$,
$\{q_{2(\frac{i}{2}-1)+1}, q_{2(\frac{i}{2}+2)+1}\}$, ..., $\{q_{2*0+1}, q_{2(i+1)+1}\}$.

At the second step of iteration i, the placement of logical qubits on bottom row changes to the following:

$$q_{i+3}, q_{i+1}, q_{i+5}, q_{i-1}, .., q_{2(i+1)+1}, q_1, q_{2(i+2)+1}, .., q_{2(N-1)+1}$$

GT gates are performed between two qubits at the same column on $\{ q_i, q_{i+3} \}$, $\{ q_{i+2}, q_{i+1}\}$, ..., $\{q_0, q_{2i+3} \}$.

In the third step of the iteration i, swaps and GT are performed in parallel on top row and bottom row separately. This time, top row performs swaps and bottom row performs GT. Swaps are on $\{q_i, q_{i+2}\}, \{q_{i-2}, q_{i+4}\}$, ...$\{q_0, q_{2*(i+1)}\}$. GTs are on $\{q_{i+3}, q_{i+1}\}$, $\{q_{i+5}, q_{i-1}\}$, ...$\{q_{2i+3}, q_1\}$.

With these steps, the even iteration completes. It will go to an odd iteration $i+1$. Before operations at iteration $i+1$ start, the qubit layout at the top row now becomes

$$q_{i+2}, q_i, q_{i+4}, q_{i-2}, .., q_{2(i+1)}, q_0, q_{2(i+2)}, .., q_{2(N-1)}$$

The steps in the even iteration complete a part of GTs such for which the subscript summation of each pair of qubits is $2i+2$, and all GTs such that the subscript summation $2i+3$, and a part of GTs for which each pair's subscripts sum to $2i+4$.

The odd iterations are similar. It is just that the pairs of qubits should start from the second column from the left end of the grid when doing GT or SWAP for the step 1 and step 3 in the iteration, which we will not discuss in details here.

By repeating this pattern, all GT gates can be performed with respect to the loop in Fig. 28 (b). It is just that one parallel iteration in that loop might be split into two parts to be distributed into an even iteration and an odd iteration.

Asymptotically, every two parallel layers in Fig. 28 (b) take 3 cycles using the transformation pattern in Fig. 30. The total number of layers is $2n-3$ when the number of qubits is n. Our generalized solution then takes $3n + O(1)$ cycles.

Our generalized solution is presented in Fig. 31 (b).

In some scenarios, it does not allow concurrent swaps and two-qubit gates. Hence, we added the constraint that either GT gates or swap gates in each cycle (but not both). Under this constraint, we solve for an optimal solution. And we visualize such a solution of QFT-8 on a 2x4 architecture in Fig. 32.

Initial placement is column major as shown in Fig. 32 step (1).

Now a more elegant pattern shows and the depth is still $3n + O(1)$. We still form every three steps as one iteration (starting from step 1). We let iteration index be i, starting from 0. For each iteration, the first step only perform GTs and all GTs applied to qubits on the same column. The number of GT performed increases by 1 at each iteration until about half way of the iterations and then decreases by 1 at each iteration. The second step of each iteration only performs swap gates. All swaps only happen to qubits on the same row. The third step of each iteration performs GT gates only.

It is worth noting that the layout of these logical qubits form a graph that is isomorphic to that at the beginning (as if the layout is mirrored, similar to the case in LNN). This is a nice property of the structured QFT transformation methods we discovered.

We show the pseudocode of this solution in Fig. 31 (c).

5.7.1.2 Building Block Circuits

We show results for the building block circuits that include adders, modular function, and various counters in Table 3. These are also the benchmarks used in the work by Wille *et al.* [28] for gate-optimal qubit mapping. Our mapper finds time-optimal solution very fast, usually in less than one second. The quantum architecture is IBM's QX2. Note that for this set of benchmarks, both initial mapping and transformed circuit are determined optimally, while for QFT experiments in Section 5.7.1.1, we tried different initial mappings (row major and column major) just for discovering the patterns.

In this table, *ideal cycle* refers to how many cycles the original circuit would take on an ideal architecture where every two qubits are connected; *optimal cycle* is the number of cycles we found for the target architecture; *mapper overhead* is the time the mapper takes to find the optimal solution. We implemented the mapper using C++ and it was running on

Intel Xeon E5-2620v2 CPU.

5.7.1.3 Comparison with OLSQ

In Table 4 we show our results compared against OLSQ's depth-optimal results [26] on the benchmarks used in that paper. We correctly find the same optimal depths, but are able to do so around 9 to 1500 times faster depending on the benchmark. For this experiment, swaps were treated as having a latency of 3 cycles, and all other gates as having a latency of 1 cycle. When using our program we first tried to find an initial mapping that could satisfy all CNOTs in the circuit without swaps – if that failed, then we reran our mapper program with pure initial swaps allowed, and added together the times in our overhead column of Table 4. Our output circuit includes both a selected initial mapping and inserted swaps using the approach defined in Section 5.4.3.

5.7.2 Approximate Analysis

We relax our model to solve for large benchmarks. We aim to find a good solution within reasonable amount of time while not sacrificing the search quality too much.

We approximate it in the following ways. When an original gate is ready to execute with respect to dependence and coupling constraints, we immediately schedule it. Thus we eliminate the expanded nodes which do not schedule all ready original gates implied by their parent node (the state node).

We also reduce the number of expanded nodes by not allowing swaps that cause the executable gates on the CNOT frontier not executable. By executable we meant the coupling and dependence constraints are resolved. We rank the expanded nodes and only push the top-k into the priority queue. When the priority queue size reaches a threshold q, we cut it by a fixed number v through deleting the nodes that made the least progress in the circuit. If we need a tie breaker, we just rank them by the cost function. We choose the parameters k, q, and v as 10, 2000, and 1000.

We handle initial mapping on-the-fly in a greedy manner. Before calculating the cost of a node, we look at the qubits in each of its CNOT gates that are executable with respect to

dependence constraints: if one or both of its qubits are not yet mapped, then we pick an assignment that minimizes their physical distance. If at the end of the program there are any qubits that were never mapped (due to never being used in CNOT gates), then we assign them arbitrarily.

This method scales better than the optimal search method. It is non-optimal, but in practice it performs well.

5.8 Related Work

Early studies on qubit mapping problem focus on linear nearest neighbor architectures, that is when qubits are placed in a one dimensional grid, and one qubit has at most two neighbors. In this type of architecture, Shafei *et al.* [21] have modeled the qubit mapping problem as constraint solving problem and use satisfiability (SAT) solvers to solve for qubit mapping. It works well when the number of qubits is small and the search space is small. Maslov [15] has obtained and proved optimal qubit mapping for the quantum fourier transform (QFT) algorithm for LNN.

As quantum computers with more complex connectivity structure are built, a larger number of studies investigate the qubit mapping problem. However, most of these studies [13, 24, 25, 28, 34, 35] focus on minimizing the number of inserted swap gates and enhancing the parallelism among the swaps, but not the time of the entire circuit. Zulehner *et al.* [35] proposes a systematic A* algorithm for optimizing the number of swap gates for a fixed layer of CNOT gates that need to run concurrently. It pre-processes the circuit by partitioning the circuit into different layers, and solve the mapping problem layer by layer. Li *et al.* [13] use a frontier to keep track of the CNOT gates that cannot be scheduled on the fly and formulates a multi-objective function for ranking different SWAP insertion strategies. Li *et al.* [13] has briefly discussed the trade-off between the inserted SWAP number and the depth of the circuit, but not systematically addressed the time-optimal problem. Siraichi *et al.* [24] notes the similarity between the swap insertion problem and the subgraph isomorphism problem, which is essentially fitting a program qubit interaction graph into the physical qubit coupling

graph. But they do not provide optimal solutions. The studies [16,27] observed the variability of qubit error rates in IBM quantum computer and develop variability-aware qubit mapping strategies.

The study that is most relevant to ours is OLSQ by Tan *et al.* [26]. OLSQ is a constrained based solver. It solves for the time coordinate of each gate (including swap gate) and the qubit mapping at every time coordinate, and the objective function is the total depth. It is optimal with respect to a depth upper-bound. It tests different upper bounds of the circuit depth until it finds a solution. If the preset depth upper-bound is smaller than the actual optimal depth, it will not return a solution. They start from the the longest weighted path T in the DAG, since a circuit runs at least T cycles. If it does not return a solution with depth $\leq$ T, it changes the upper bound to T+1. If with T+1, it is still unsatisfiable, it goes to T+2, T+3, and so on until a feasible solution is found. While their method can find optimal solutions, it may suffer from scalability issues when the optimal circuit time is not close to T. Therefore they geometrically increase T each time by $(1 + \epsilon)$x until an optimal solution is found. Our model explicitly solves for an optimal solution and does not impose any constraints.

The study by Childs *et al.* [3] aims to minimize the depth. But it minimizes the depth of inserted swaps. Each set of co-running swaps is modeled as a graph matching (as no two parallel swaps share a qubit), then it tries to find a minimal sequence of matchings to achieve the desired permutation. Their theoretical model does not consider the parallelism between inserted swaps and original gates. Lao *et al.* [12] considers the parallelism between inserted swaps ad original gates, but their approach is not theoretically optimal.

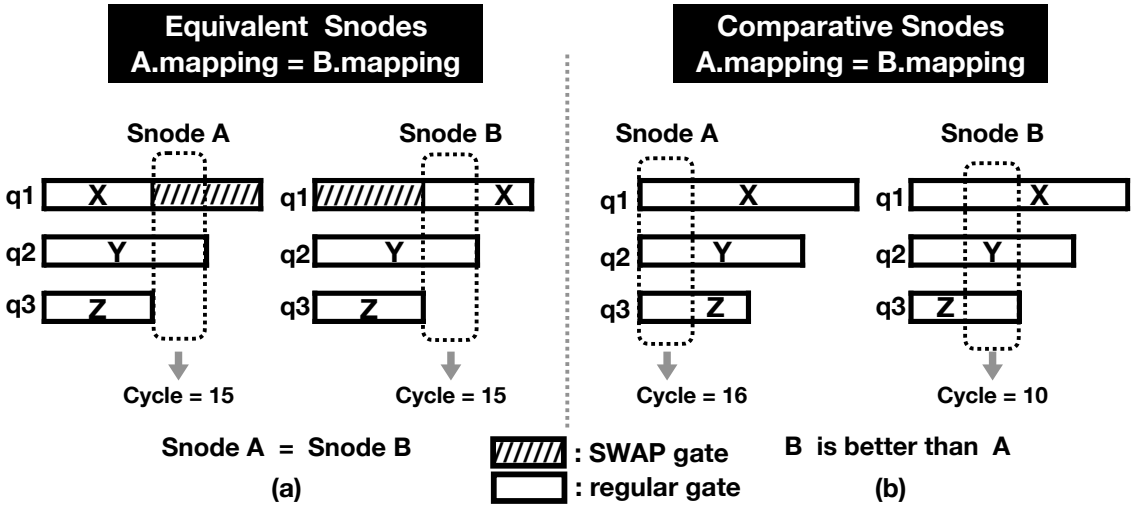

Figure 23: Filtering example (a) Two equivalent search nodes. Both A and B will finish the same number of original gates. They also have achieved the same mapping assuming all active swaps immediately take effect. But the order of the gates is different. (b) The two state nodes will have to take different number of cycles to achieve the same state. But B is faster, A can be safely pruned without affecting optimality.

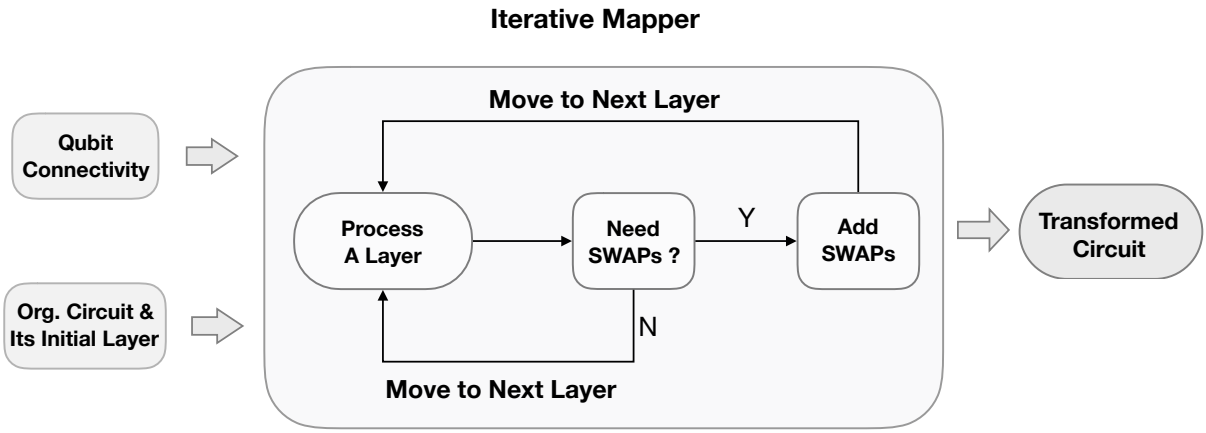

Figure 24: Our Optimal Search Framework

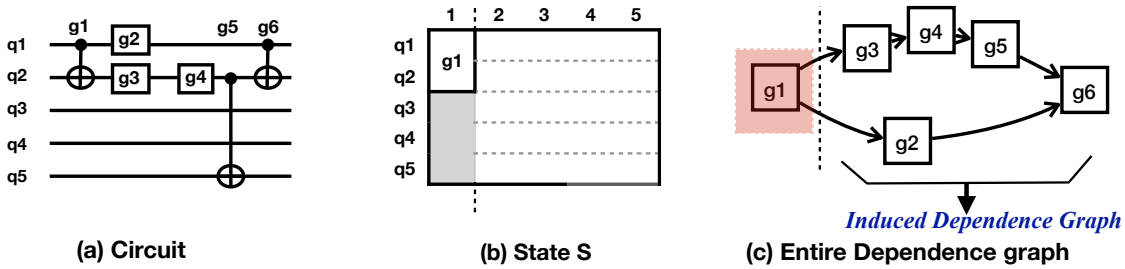

Figure 25: Two components of a circuit with respect to a search node: (a) logical circuit, (b) a search node S for cycle 1 indicating already scheduled gates by cycle 1, and (c) the dependence graph, where the part after the dashed line is remaining circuit to be executed.

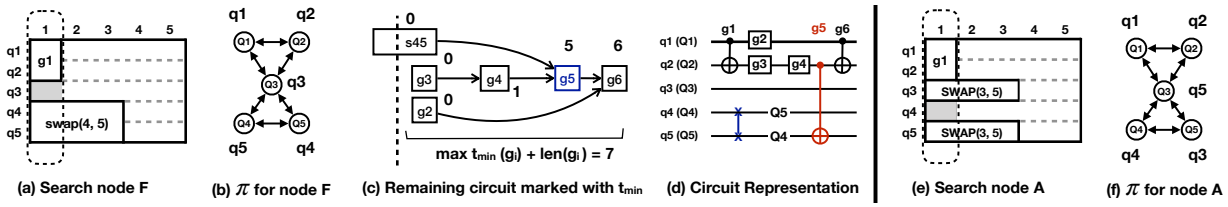

Figure 26: Cost calculation (a) Search node F, (b) qubit mapping for remaining circuit induced by F, (c) remaining dependency graph marked with t_{min}, and (d) circuit representation; (e) search node A, (f) qubit mapping for remaining circuit induced by A.

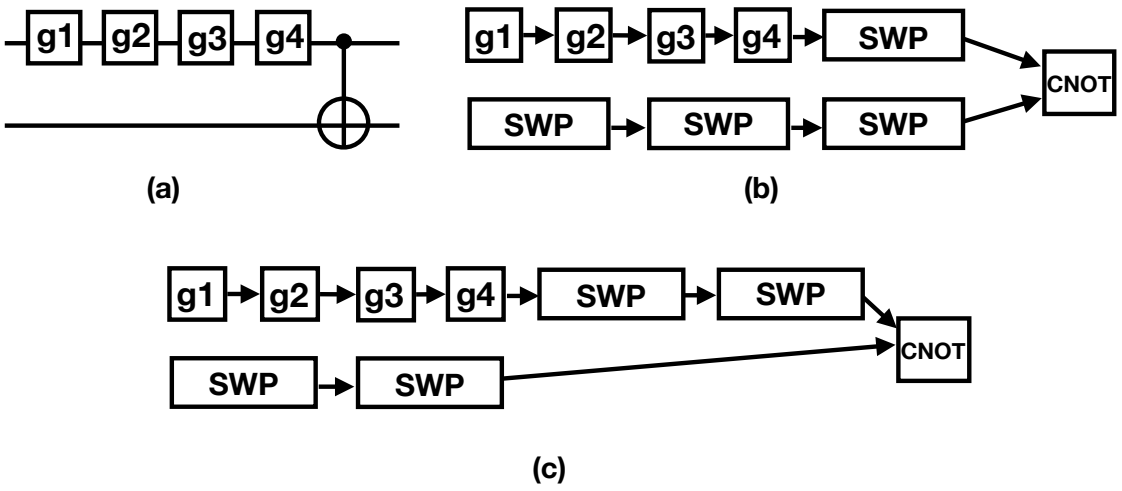

Figure 27: An example circuit, demonstrating a common fallacy in reasoning about the heuristic function. (a) Logical circuit, assuming that the distance between the two qubits is 5. (b) The dependency graph of the circuit, if we choose to place 1 SWAP on the first qubit, and 3 SWAPs on the second. (c) The dependency graph of the circuit, if we choose to insert two swaps on each qubit. Assuming each SWAP takes 2 cycles and each original gate takes 1 cycle in this example.

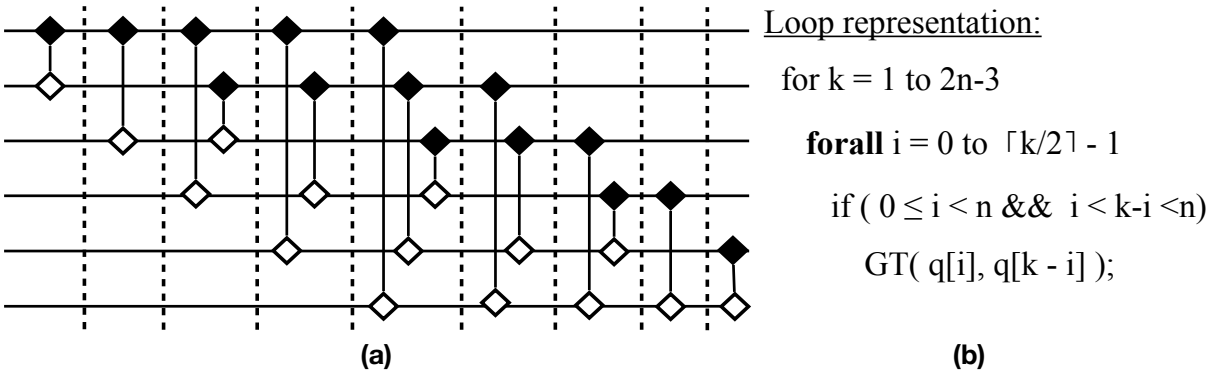

Figure 28: (a) QFT-6 circuit rearranged to exploit parallelism; (b) Affine loop representation of re-arranged n-qubit circuit.

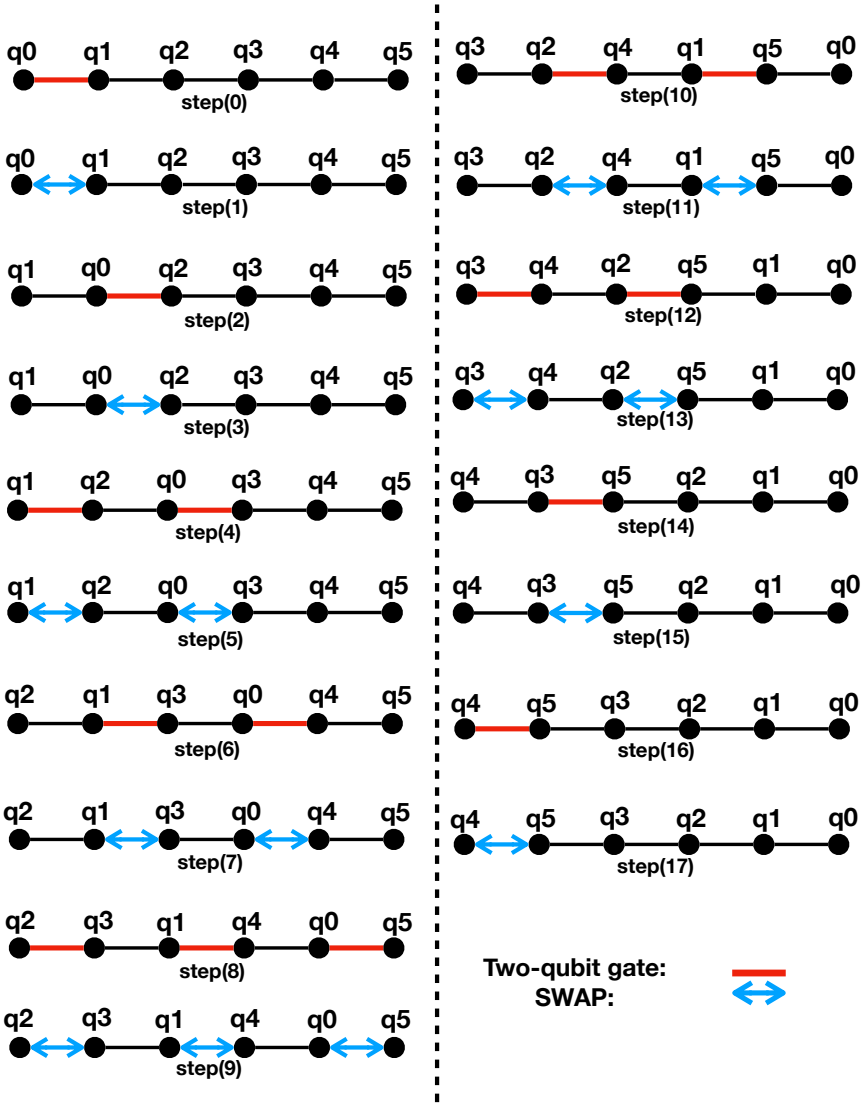

Figure 29: QFT-6 on LNN. Step (17) is not necessary. We are adding it to show the pattern.

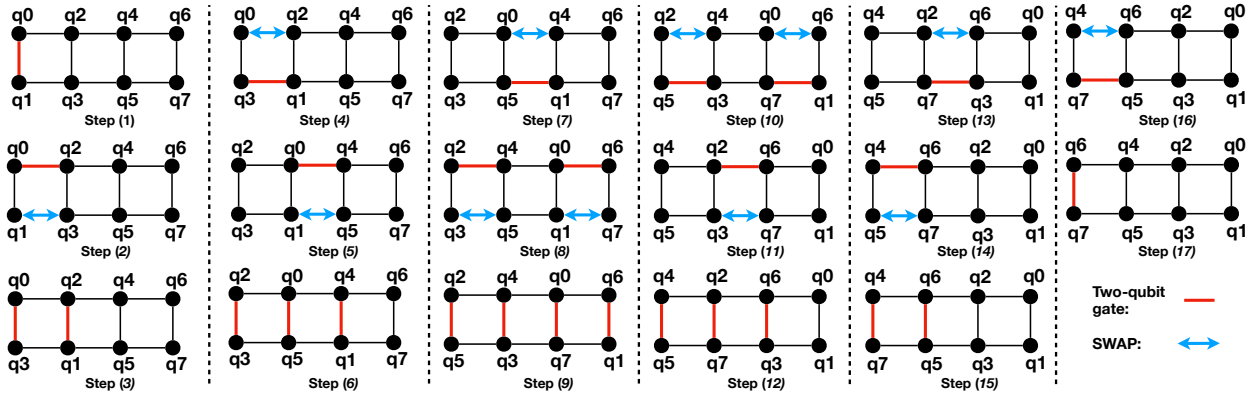

Figure 30: Optimal scheme for QFT-8 with 2×4 qubits. Each sub-figure represents a step of the execution, which takes one cycle. It also represents the state of the circuit at each cycle. There are in total 17 steps, and thus 17 cycles.

```
for m = 0; m+=2; m < 4n - 6 do
  k = (m/2)+1;
  for i= 0; i++; i < (k - i) do
    if  i < n && k - i < n, then
      GT( q[i], q[k - I] );
  for i = 0; i++; i < (k - i) do
    if  i < n &&  k - i < n, then
      SWAP( q[i], q[k - I] );
```

(a)

```
for i = 0; i++; 2*i+2 < 2n - 1 do
  for j = 0; j += 2; j < i  do
    if (2i - j) < n && j < n, then GT( q [j], q [2i-j] );
    if (2i -j + 1) < n && j+1 <n, then SWAP( q[j+1], q[2i - j + 1] );
  for j = 0; j += 2; j < 2*i+1, do
    if (2i+1-j) <n && j < n, then GT( q[j], q[2i+1-j] );
  for j = 1; j += 2; j <= i , do
    if (2i+2-j) <n, then  GT( q[j], q [2i + 2 - j] );
    if (2i+1-j) <n, then  SWAP( q[j-1], q[2i+1-j] );
```

(b)

```
for i = 0; i++; i <= n-2 do
  for j = 0; j++; j < i, do
    if  j < n && (2i-j) < n, then SWAP( q[j], q[2i-j] );
  for j = 0; j++; j < i do
    if  j < n && (2i-j) < n, then GT( q[j], q[2i-j] );
  for j = 0; j++; j < i+1 do
    if  j < n && (2i+1-j) < n, then GT( q[j], q[2i+1-j] );
```

(c)

Figure 31: Generalized solution for optimal schemes of QFT: (a) n-qubit QFT on LNN; (b) n-qubit QFT on 2×N architecture where $N = n/2$; (c) n-qubit QFT on 2×N architecture where swaps and CNOT cannot be mixed in one cycle.

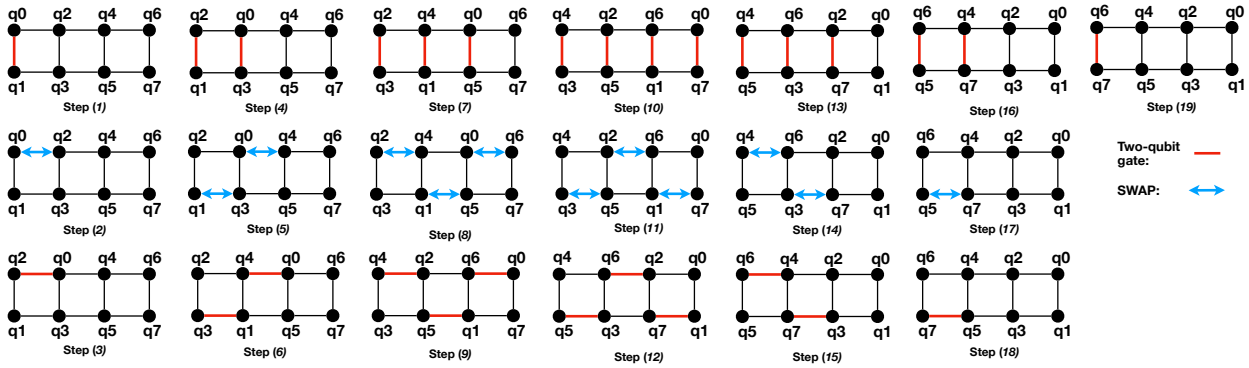

Figure 32: An alternative optimal scheme for QFT-8 with 2×4 qubits. Each sub-figure represents a step of the execution, which takes one cycle. It also represents the state of the circuit at each cycle. There are in total 19 steps, and thus 19 cycles.

Table 3: Summary of optimal analysis on Wille's [28] benchmarks for IBM QX2 architecture, with swap latency of 6 cycles and CX latency of 2 cycles; **n** denotes the number of qubits; Mapper Overhead, measured in seconds, is how long it took to generate the mapping.

Name	n	Gate Count	Ideal Cycle	Optimal Cycle	Mapper Overhead (s)
3_17_13	3	36	39	39	0.012
4gt11_82	5	27	38	40	0.044
4gt11_84	5	18	19	19	0.011
4gt13_92	5	66	64	64	0.014
4mod5-v0_19	5	35	37	45	0.075
4mod5-v0_20	5	20	21	27	0.052
4mod5-v1_22	5	21	22	28	0.053
4mod5-v1_24	5	36	36	42	0.085
alu-v0_27	5	36	35	40	0.043
alu-v1_28	5	37	37	42	0.029
alu-v1_29	5	37	36	41	0.052
alu-v2_33	5	37	36	41	0.036
alu-v3_34	5	52	53	59	0.314
alu-v3_35	5	37	37	42	0.038
alu-v4_37	5	37	37	42	0.038
ex-1_166	3	19	21	21	0.013
ham3_102	3	20	24	24	0.013
miller_11	3	50	52	52	0.016
mod5d1_63	5	22	24	34	0.076
mod5mils_65	5	35	37	46	0.115
qft_4	4	6	10	16	0.035
rd32-v0_66	4	34	36	41	0.045
rd32-v1_68	4	36	36	41	0.042

Table 4: Comparison of our results against OLSQ's depth-optimal results; We let each gate take 1 cycle as the setup of OLSQ [26]. Mapper overhead is measured in seconds. OLSQ is using a different CPU for qubit mapper implementation which is Intel Xeon E5-2699v3.

Name	Arch	Ideal Cycle	**OLSQ**		**Ours**	
			Cycle	Overhead	Cycle	Overhead
4gt13_92	ibmqx2	38	38	145.74	38	0.01
4mod5-v1_22	grid2by3	12	20	90.20	20	0.64
4mod5-v1_22	grid2by4	12	20	151.28	20	17.35
4mod5-v1_22	ibmqx2	12	15	21.60	15	0.03
adder	grid2by3	11	11	10.95	11	0.03
adder	grid2by4	11	11	13.45	11	0.01
adder	ibmqx2	11	15	39.71	15	0.06
mod5mils_65	ibmqx2	21	24	87.76	24	0.05
or	ibmqx2	8	8	3.55	8	0.01
qaoa5	ibmqx2	14	14	10.41	14	0.01
queko_05_0	aspen-4	5	5	68.89	5	0.01
queko_10_3	aspen-4	10	10	592.91	10	1.02
queko_15_1	aspen-4	15	15	4912.35	15	26.70

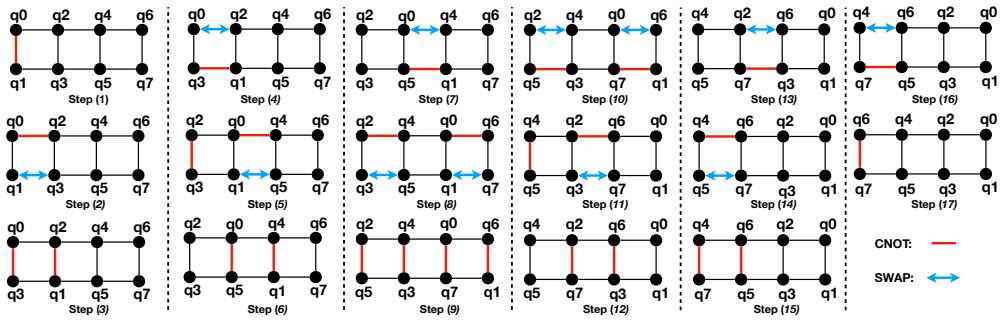

Figure 33: One possible solution for QFT-8 on 2×4 allowing two-qubit gate and swap on one cycle.

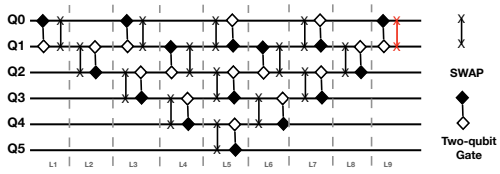

Figure 34: One possible solution for QFT-6 on LNN.

6.0 Crosstalk-aware Qubit Mapping

6.1 Introduction

We propose an optimal solution to tackle the crosstalk problem. The contributions are summarized as follows:

1. We reveal that crosstalk can be mitigated as early as at the hardware mapping process. Based on this finding, we build a crosstalk-aware hardware mapping framework to address crosstalk.
2. We consider two methods with different tolerances of crosstalk. One method allows zero crosstalk while the other allows crosstalk to some extent. We aim to achieve a good tradeoff between crosstalk tolerance and overall circuit fidelity. Zero crosstalk tolerance may lead to unnecessarily long circuit and result in decoherence errors, which lead to low overall circuit fidelity, while tolerating a small amount of crosstalk may improve circuit depth significantly, and hence improve overall fidelity.
3. We evaluate our method in fidelity, depth, and compilation overhead. The result not only shows the promising improvement in fidelity for an average of 20% but also only takes 10% of the compilation time.

6.2 Framework Overview

In this section we introduce our crosstalk-aware program compilation framework. The framework contains a swap insertion component, and a scheduling component. We take a quantum program as input and produce a hardware-compliant circuit with timing information as output. We show the framework in Fig.35.

We first analyze the dependency of the input program since a quantum gate can only be executed if and only if its dependency has been resolved. We construct a directed acyclic graph (DAG) for all quantum gates.

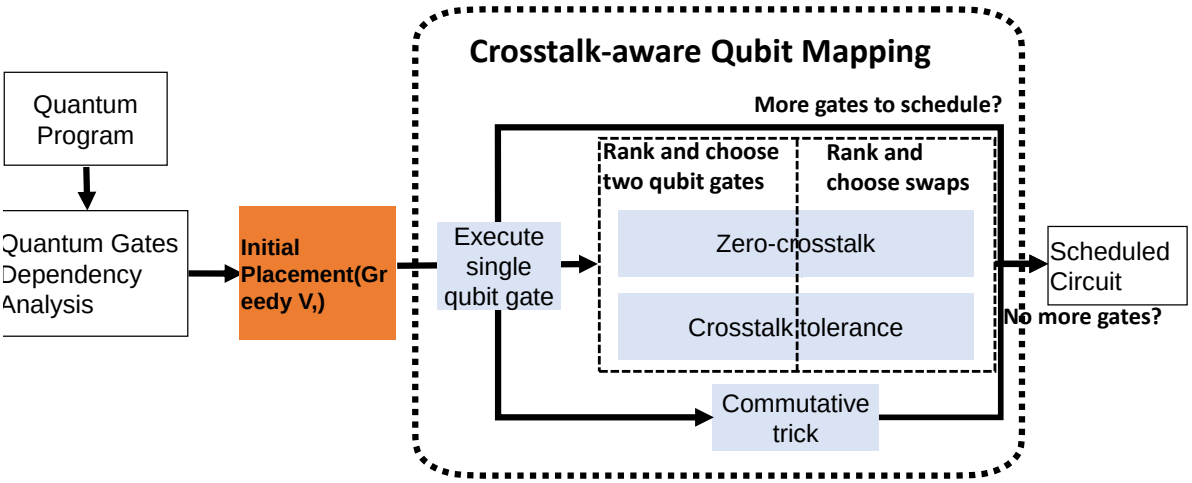

Figure 35

With the aim of inserting fewer swaps, we implement an initial mapping component by the subgraph isomorphic method. We take a subset of gates from the beginning of the circuit and try to construct a subgraph embedding with respect to the architecture coupling graph. Among all different subgraph embeddings, we pick the one that has minimal crosstalk.

Next we introduce our swap insertion and scheduling components. This is a repetitive process based on the ready gates in the frontier. We first schedule ready single-qubit gates in ready-gate-set. Then we will schedule the ready two-qubit gates and insert swaps if necessary. Moving to the next cycle, we update the qubit mapping and the ready-gate-set with respect to the DAG. We check whether there is any remaining gate in the DAG. If so, we repeat the process. Otherwise, we output the hardware-compliant circuit with inserted swaps and the timing information for each gate. To insert swaps and schedule gates efficiently, we have the following considerations:

We aim to schedule as many two-qubit gates simultaneously while maximally mitigate the crosstalk errors. Our main goal is to minimize the depth of the circuit while tolerating minimal crosstalk. In another word, we minimize the decoherence errors under a strict crosstalk-tolerance constraint.

While it is important to mitigate crosstalk error, it might be important to consider

mitigating the decoherence error, and in particular the trade-off between decoherence error and crosstalk error. We discovered that by relaxing the crosstalk constraint to some extent, we we can significantly reduce the depth of a circuit which contributes to significant reduction in the decoherence error. In order to achieve maximal fidelity of the overall circuit, we need to find a sweet spot between crosstalk error mitigation and decoherence error mitigation.

We exploit the gate-reordering feature in quantum programs for some particular. In an important class of applications called quantum approximate optimiztion algorithm (QAOA), gates can commute or even be reordered with respect to a certain rule. With the consideration of the gate-reordering feature in quantum programs, we can avoid crosstalk error without having to incur the cost of delayed gate execution.

We discuss in details each of the component in Section 6.3.

6.3 Crosstalk-aware compilation

In this section, we will talk about all details which have shown in our framework, and also we will introduce some considerations with tricks which can better improve our result. We use an example shown in Fig. 36 to clarify our different technique process.

6.3.1 Execute single qubit gates

We first execute the single qubit gates which are ready to be executed from the DAG information, we schedule the single qubit gate which is shown in Fig.36. We know that single qubit gate can always be executed as long as there is no dependency constraint in the dependency graph, from the Fig.36, we know that the first Hadamard gate H(0) can be execute and there is no cross-talk caused by single qubit gate. After schedule the single-qubit gates we want to schedule the two-qubit gates and inserting swaps.

6.3.2 Scheduling two-qubit gates and swap insertion

6.3.2.1 Strict Crosstalk Constraint

Based on our framework consideration, we first want to find the minimum depth given strict crosstalk constraint which we do not allow any crosstalk happens. Since there is a finite number of gates and physical qubits, there is only a finite number of combinations of gates and swaps that can be executed simultaneously at any given moment. The most common traditional solution for this problem is to use A* search from time-optimal mapping work [33]. We modified the original method by adding the crosstalk constraint. Then we find the optimal schedule choices which can minimize the depths. Previous work has been proved that this result is optimal and we can use this to find the minimum depths. However, due to the program size, using the optimal method to solve the mapping problem is not always practical in large quantum program, so we also introduce a reasonable heuristic solution suit for more complex program. We will show our solutions in details. We introduce a heuristic solution to deal with scheduling two-qubit gates and swap insertion. Since only adjacent qubits can be implemented the two-qubit gate operations. We will first schedule the two-qubit gates which do not need extra swaps, so our heuristic approach can be divided into two steps: 1) Execute two-qubit gates(Chromatic method). 2) Rank and choose swaps. Given a quantum machine architecture we will have its connectivity information from the coupling graph shown in Fig.36(a), it shows a Melbourne quantum machine architecture and its connectivity information. We first try to schedule the two-qubit gates by using the chromatic method shown in Fig.36(e), each vertex of graph represent a two-qubit gate operations, and they have edges if and only if they have crosstalk errors when executing them simultaneously. We color the vertex under the constraint that vertices can not have the same color if they share an edge, we pick the most vertices with the same color and execute the two-qubit gates. Which in Fig.36(e). CX(5,6) has the crosstalk with CX(8,9) so these two vertices has an edge. Also CX(10,11) has edge with CX(8,9). After coloring, we pick CX(5,6),CX(10,11).

After consider executing the single qubit gate and the cnots which does not need extra swaps, we need to consider all other cnots who need extra swaps before they can be executed.

Although there are huge search space for backtracking all the swap combinations, we have several constraints which can greatly prune the unwanted combinations and shrink the search space: 1) All swaps must contain the qubit relate to CX gate which need swaps. 2) All swaps can not use the qubit which has been used in executing single qubit gate and scheduling two-qubit gate process. 3) All swaps must meet the coupling constraint. So based on these three constraints, we can significantly shrink the search space and solve the search time overhead.

To properly evaluate which combination of swap insertion for the current and next cycles, we define the cost function shown in equation 2 . If the distance does not change or even increase after swaps, then it means the swap of qubits no longer get closer and we will ignore them. We want to see the distance will decrease after the swap operations, and it will decrease the distance for cnots in next layer. We set the weight w for different purpose, for here we set strict crosstalk constraint, so we set w to be the number of qubits. Then it means if any crosstalk happens, it will give us a big number which we will never use that. We use the cost function and rank all the combinations of swap insertion, choose the combination with lowest score.

We show an example in Fig. 37, we can see we have crosstalk between coupling quibts(1,2) and (11,12). And we score all the combinations after pruning by our three constraints talked before. There are mainly three types, 1)the first type is called depth overhead, although it does not arise extra crosstalk and really decrease the distance of the two qubits. However only one swap can happen at the same time. So the depth overhead will be huge and the score is -1 since it only decrease 1 distance at a time. 2)The second type is some swaps which are executed at the same time can cause crosstalk, we can see if we execute swap(1,2) and swap(11,12) at the same time. We will have crosstalk errors, so based on our cost function, we will give a large score. 3) Then for the third type is swap(11,10) and swap(0,1), this combination actually increases the distance of CX gates. So we score them as 2. Finally is our optimal choice which are swap(1,2) and swap(3,11), these swaps not only decrease 3 distance of CX gates but also will not cause any crosstalk errors, so we score them -3 and they have highest priority.

After choosing the swaps, we should updated the coupling map which correspond to

logical qubits map to physical qubits. And if there are still gates remaining, we will repeat this loop until we execute all the gates.

$$
\begin{aligned}
Cost\_swap(C_i) = & Num(crosstalk) * W - \\
& \sum_{i=1}^{n} (original\_distance(CNOT_i) - \\
& Update\_distance(CNOT_i)) - \\
& \alpha * \sum_{i=1}^{m} (original\_distance(CNOT_i) - \\
& Update\_distance(CNOT_i))
\end{aligned}
\tag{2}
$$

We show an algorithm below, the P is the quantum program and we want a crosstalk free program schedule:

6.3.2.2 trade-off between crosstalk errors and decoherence errors

To further seek the trade-off between depths and crosstalk errors. By allowing some crosstalk tolerances, we can further decrease the depths which is to decrease the decoherence error. In order to find the maximum fidelity of the overall circuit, we need to find a sweet point between crosstalk errors and decoherence errors.

We show an example below in Fig. 38, we show the trade-off between crosstalk error and depths. The input circuit is shown in (a) with two CX gates. If we use the strict constraint with no crosstalk, then our scheduled circuit will be like in Fig. 38(b). Since execute swap(0,1) and swap(2,3) will have crosstalk error, so we must execute them one by one which will cuase the increasing the depth to 11(or we can say increase the decoherence error).

However if we allow some crosstalk tolerance, then we can find that we can execute swap(0,1) and swap(2,3) at the same time shown in Fig. 38(b). Which can significantly decrease the total depth which is also decreasing which only takes 5 depths. This will also lead to the decreasing for deocherence error. We want to evaluate these two methods in fidelity for overall circuit based on the equation 3. [need to add the fidelity number for (b)(c) and help us choose the best choice]

Algorithm 3: Heuristic solution

```
Function get_cycles(P):
    cycles = 0;
    schedule = Map[cycles,set of CNOTs];
    global_min = Max_Int;
    min_set = ∅;
    unused_qubit_set Q =all qubits;
    Priority_Queue pq;
    for CNOTs i in P do
        if i.indegree == 0 then
            pq.insert(i);
    while !pq.isEmpty() do
        for each set s in all combination of CNOTs in pq without swaps) do
            if cost_cnot(s) ¡ global_min then
                global_min=cost_cnot(s); min_set=s;
        schedule[cycles].append(CNOTs in min_set);
        update_dependency(min_set);
        update_pq(min_set);
        Q.delete(qubits in min_set);
        global_min = max_integer;
        min_set = ∅;
        for each set t in all combination of swapping two-qubits in Q do
            if swap_cost(t) ¡ global_min then
                global_min=swap_cost(t); min_set=t;
        schedule[cycles:cycles+2].append(CNOTs in min_set);
        cycles++;
```

Based on this consideration, we want to introduce our method for maximizing the fidelity of the overall circuit. By allowing crosstalk tolerance, we start from 0 which is zero crosstalk, then we increase the number of crosstalk tolerance, since with more crosstalk tolerance, we will further decrease the decoherence error but will lead to more crosstalk errors at the same time. we try to find a sweet point which can maximize the fidelity of overall circuit.

6.4 Evaluation

6.4.1 Experiment setup

We evaluate our new crosstalk-aware mapping method and present the results here. The benchmarks are selected from RevLib [29], IBM Qiskit [1], and ScaffCC [9]. And to suit for larger benchmarks, we range the backend from ibmQ5 to large circuit Melbourne and Tokyo, which has different connectivity and our method can scale for different kind of architecture. For crosstalk prop, it is generated by machine experiment, for different time and date, the crosstalk error may vary significantly, we can manually input the crosstalk prop based on the machine back-end properties provided by qiskit.

Baseline We use the best known prior approach crosstalk-adaptive schedule as our baseline. Since their method mainly focus on the gate scheduling, for the swap and mapper part we choose Sabre's method [13]. It is worth mentioning that our approach can take any gate latency as input parameters and generate transformed circuits based on the input. To make evaluation results as close to real machines as possible. We use IBM's 14-qubit Melborune and 20-qubit Q20 Tokyo architecture [13] as the underlying quantum hardware for different benchmark sizes. We assume each gate takes unit time and swap gate is implemented by three CX gates.

6.4.2 Experiment result

We will show our experiments based on our methods from last section. We first show the result which try to minimize the depths under zero-crosstalk constraint. Then we will

show the result for maximizing the fidelity of the overall circuit by given some crosstalk tolerance. We also want to show the compilation time advantage of our work compared with the baseline.

6.4.2.1 Zero-crosstalk

We try to test our method on different benchmarks, it consists of different well known benchmarks QAOA, QFT and some small elementary building blocks. We want to compare the circuit execution depths to the baseline under strict zero crosstalk constraint, to quantify the result, we evaluate each single qubit gate as 1 depth, CNOT gate and other two qubit gate as 2 depths and swap consists of three CNOTs.

We compared our crosstalk-aware mapping method with the both optimal and heuristic method under zero crosstalk requirement, and we compare these two methods with the baseline to find minimum depths by using the crosstalk-aware mapping. Less depths not only execute the circuit in a more efficiently way, it also leads to less deocherence errors. The result is shown in Table. 5. We find that even our heuristic method can have better performance or at least the same as the prior work. Since sometimes there does not always exist pattern which needs to do the crosstalk-aware mapping. However for most of the benchmark our heuristic method and optimal one both has great improvement from 1.2 to 1.88.

6.4.2.2 Fidelity analysis

o show our result in fidelity improve, we show the result in table6, The fidelity is calculated by the formula3. It will calculated all the error rated caused by decoherence error, read out error and gate error, crosstalk error is already calculated in the gate error.

$$\begin{aligned} fidelity = \prod_{i=1}^{n} & (1 - Error_{Decoherence}) * \\ & (1 - Error_{Measurement}) * \\ & (1 - Error_{Gate}) * \\ & (1 - Error_{Crosstalk}) \end{aligned} \tag{3}$$

6.4.2.3 Compilation time analysis

To test the compilation time overhead, we want to carefully evaluate both our heuristic and optimal method compilation time with the baseline, since baseline use SMT solver and it has compilation time overhead, we want to compare the compilation time using their method. We show the result using a bar plot at Fig.41, we can find that the prior works takes exponential longer time than our heuristic and optimal solution. Our heuristic method only takes like 0.1% of the prior work's compilation time, and even our optimal solution takes 1-10% compilation time compared to baseline. The efficiency of the compilation time will make the quantum circuit's advantage stands out, since if we spends minutes to compile the circuit, then it might weak the advantage of quantum circuit.

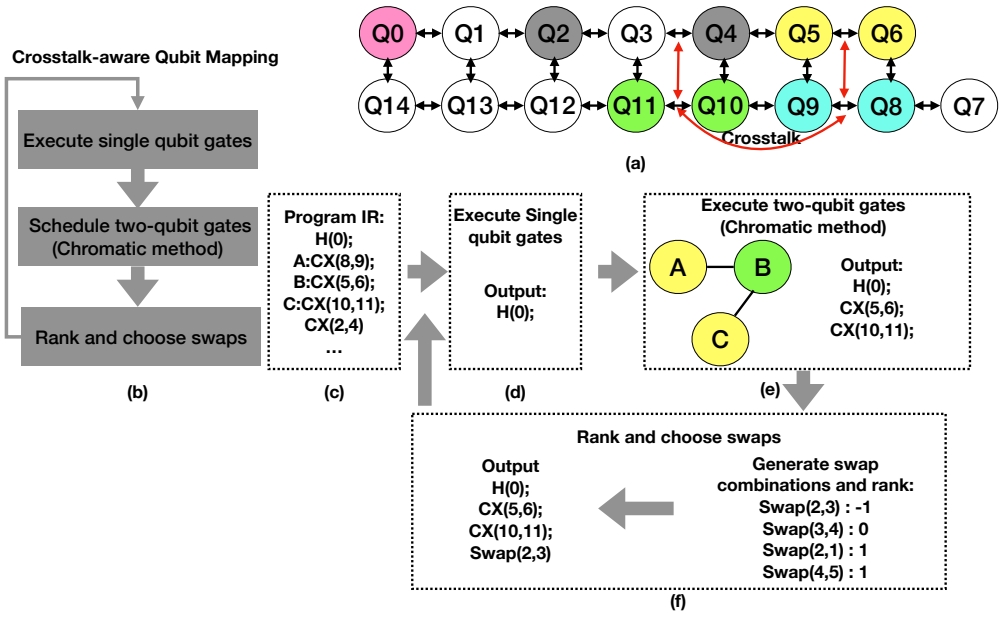

Figure 36: This is an example to show how our crosstalk-aware mapping approach:it will divided into three parts(b). (1) We first schedule the single qubit gate since single qubit gate does not need to worry about swaps and crosstalk error.(c) (2)Then we use the chromatic method to schedule the two-qubit gates which are adjacent to each other and do not need extra swaps(d)(3) After the first two steps, there are few unused qubits, and based on the connectivity, there are few choices of combination we can do the swap operations and only several swaps can help decrease the distance of those cnots which are not adjacent and need swap operations. we rank the swap combination by swap cost function, pick the combination which has the lowest score(f). We repeat this process until there are no more gates remaining.

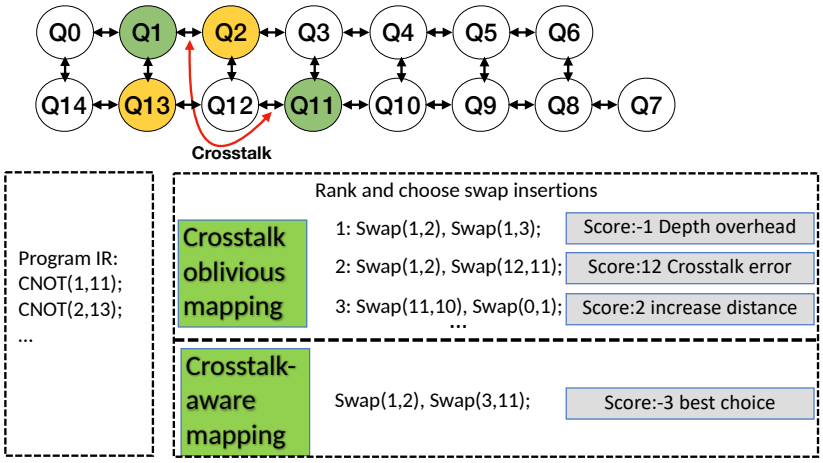

Figure 37: crosstalk-aware mapping: (a) is the input program IR with two CX gate (b) is the crosstalk-oblivious mapping which will choose swap(1,2) together with swap(3,4) since these two swap has crosstalk error, so adaptive schedule will delay these gates. (c) is our crosstalk-aware mapping will choose swap(0,1) together with swap(3,4)

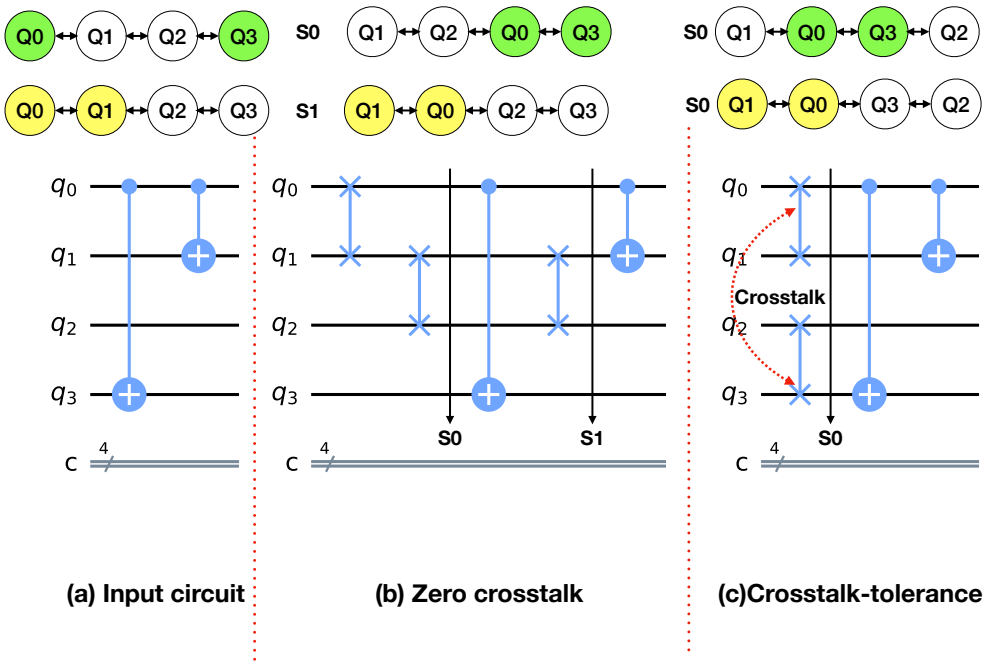

Figure 38: An example for showing trade-off between crosstalk error and depths. We show the input circuit in (a), and the scheduled gates in (b) if we set zero crosstalk error and the depth of the overall circuit is 11. However if we allow the crosstalk error in (c), we can greatly decrease the total depths to only 5.

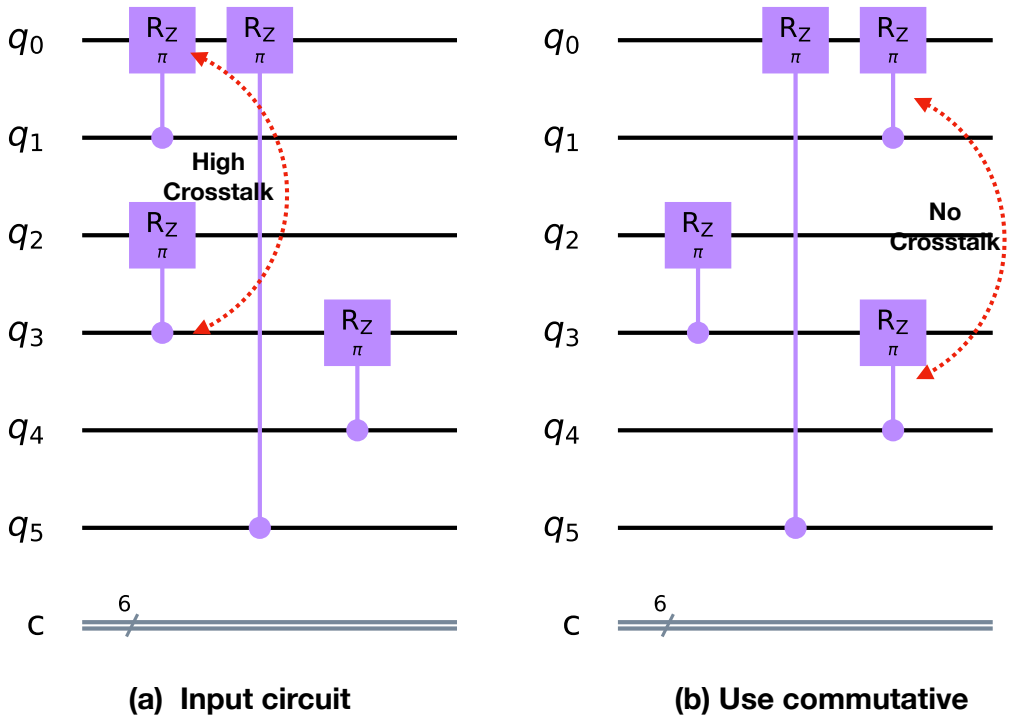

Figure 39: An example for showing adding commutative, (a) is the original input circuit which has high crosstalk error. If we use commutative to change the order of the gates, then in(b) we will find there is no more crosstalk error.

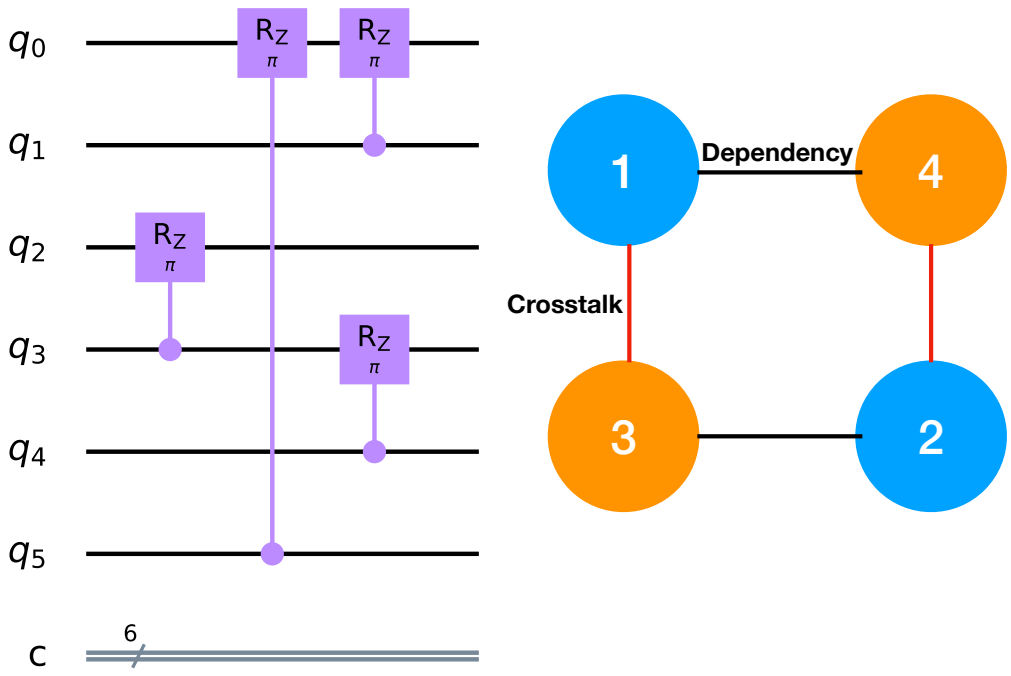

Figure 40: An example for showing we use graph coloring to find the schedule of the QAOA control rotation gates

Table 5: Summary of Experiment Results

Benchmark	Crosstalk-aware (optimal)	Crosstalk-aware (heuristic)	Baseline	Speed up
QFT4	10	12	12	1.2
QFT10	130	140	192	1.47
QFT13	141	150	221	1.56
QFT20	289	311	418	1.44
QAOA3	13	16	16	1.23
QAOA6	102	161	192	1.88
QAOA10	175	183	220	1.25
rd32-v0.66	38	46	52	1.52
rd32-v1_68	31	32	42	1.38
4gt11_84	19	20	22	1.15
urf2_268	14116	20112	22592	1.60

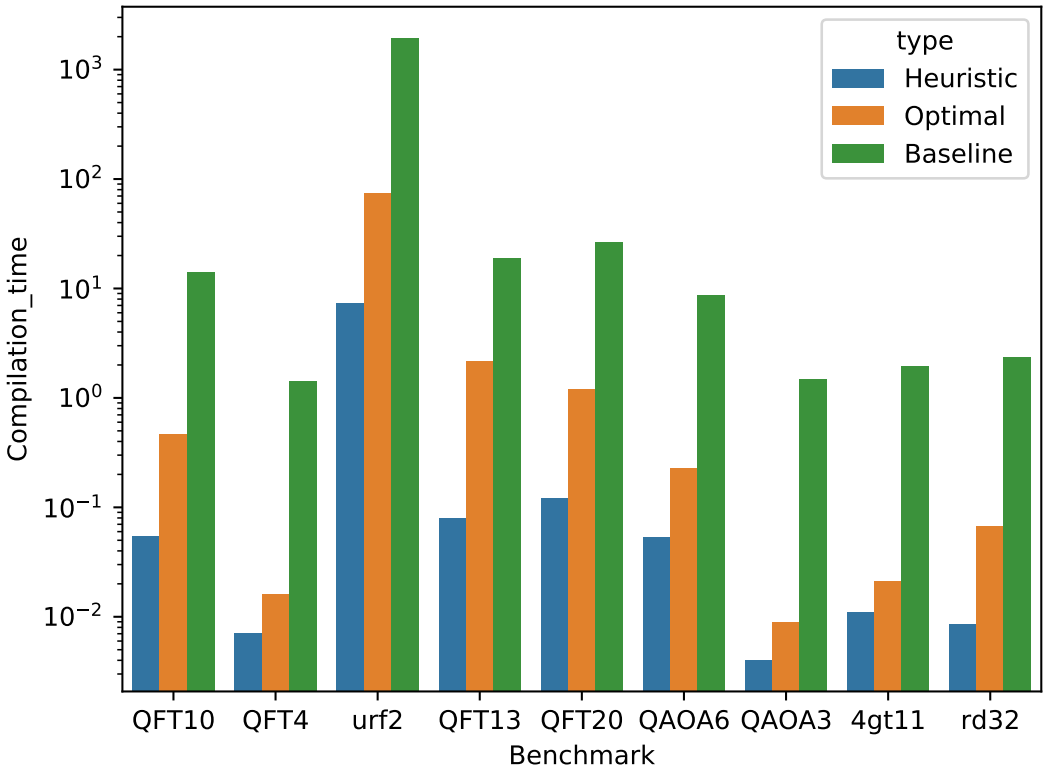

Figure 41: Compilation time sensitive analysis: the x axis is the different benchmarks and y axis is the compilation time in seconds, we find our heuristic and optimal methods mainly takes less than 1 second to finish and we have only used 0.1% of the baseline, which shows exponential speed up in compilation time

Table 6: Summary of fidelity

Benchmark	Optimal Depth	Optimal fidelity	Heuristic depth	Heuristic fidelity	baseline depth	baseline fidelity
QFT4	10	0.41	30	0.34	36	0.34
QFT10	130	1.00E-05	170	4.50E-06	192	4.70E-08
QFT13	141	4.16E-06	200	1.89E-06	221	3.68E-09
QFT20	289	9.35E-12	311	1.35E-12	418	1.12E-16
QAOA3	13	0.319067	16	0.245128	16	0.245128
QAOA6	102	0.00012805	161	7.17E-07	192	4.71E-08
QAOA10	175	2.10E-07	213	1.04E-07	233	4.02E-09
rd32-v0.66	38	0.0354656	46	0.0175592	52	0.010364
rd32-v1_68	31	0.0656067	32	0.0600877	42	0.0249549
4gt11_84	19	0.188323	20	0.172481	22	0.144682
urf2_268	19116	0	20112	0	22592	0

7.0 Conclusion

The physical layout of contemporary quantum devices imposes limitations for mapping a high level quantum program to the hardware. It is critical to develop an efficient qubit mapper in the NISQ era. Existing studies aim to reduce the gate count but are oblivious to the depth of the transformed circuit. The first work depth-aware qubit mapping presents the design of the first depth-aware swap insertion scheme. The second work slack-aware qubit mapping presents the design of the first time-efficient slack-aware swap insertion scheme. Experiment results show that our first and second works generate hardware-compliant circuits with reduced depth compared with state-of-the-art mapping schemes, with negligible overhead of increased gate count.

The third work presents a time-optimal qubit mapping model scheme. We present the first theoretical model for time-optimal qubit mapping without any implicit constraints. The theoretical model can be flexibly extended to practical algorithms. We discovered time-optimal solutions for quantum fourier transformation (QFT) on both 1D and 2D nearest neighbor architecture (using our search framework). Our solution for 1D nearest neighbor architecture is the same as a manual solution reported by Maslov [15]. But our optimal solution for 2D architecture is for the first time reported.

The fourth work, we focus on reducing the crosstalk errors in the qubit mapping process. We observe that most existing studies tackle the crosstalk problem at the very late stage of gate scheduling, missing the great opportunities in mitigating the crosstalk effect during program compilation. Therefore, we propose a crosstalk-aware qubit mapping framework that reduces the crosstalk during mapping. Evaluations show that our novel crosstalk-aware compilation can significantly reduce the error rate , with only incurring minimal circuit depth compare to state-of-the-art gate scheduling approaches.

Bibliography

[1] Héctor Abraham, AduOffei, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Gadi Alexandrowics, Eli Arbel, Abraham Asfaw, Carlos Azaustre, AzizNgoueya, Panagiotis Barkoutsos, George Barron, Luciano Bello, Yael Ben-Haim, Daniel Bevenius, Lev S. Bishop, Sorin Bolos, Samuel Bosch, Sergey Bravyi, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adrian Chen, Chun-Fu Chen, Richard Chen, Jerry M. Chow, Christian Claus, Christian Clauss, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Tareq El Dandachi, Matthieu Dartiailh, DavideFrr, Abdón Rodríguez Davila, Anton Dekusar, Delton Ding, Jun Doi, Eric Drechsler, Drew, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem EL-Safty, Eric Eastman, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Axel Hernández Ferrera, Albert Frisch, Andreas Fuhrer, MELVIN GEORGE, Julien Gacon, Gadi, Borja Godoy Gago, Claudio Gambella, Jay M. Gambetta, Adhisha Gammanpila, Luis Garcia, Shelly Garion, Austin Gilliam, Juan Gomez-Mosquera, Salvador de la Puente González, Jesse Gorzinski, Ian Gould, Donny Greenberg, Dmitry Grinko, Wen Guan, John A. Gunnels, Mikael Haglund, Isabel Haide, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Stefan Hillmich, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Haruki Imai, Takashi Imamichi, Kazuaki Ishizaki, Raban Iten, Toshinari Itoko, JamesSeaward, Ali Javadi, Ali Javadi-Abhari, Jessica, Kiran Johns, Tal Kachmann, Naoki Kanazawa, Kang-Bae, Anton Karazeev, Paul Kassebaum, Spencer King, Knabberjoe, Arseny Kovyrshin, Rajiv Krishnakumar, Vivek Krishnan, Kevin Krsulich, Gawel Kus, Ryan LaRose, Raphaël Lambert, Joe Latone, Scott Lawrence, Dennis Liu, Peng Liu, Yunho Maeng, Aleksei Malyshev, Jakub Marecek, Manoel Marques, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Dan McPherson, Srujan Meesala, Martin Mevissen, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Abby Mitchell, Nikolaj Moll, Michael Duane Mooring, Renier Morales, Niall Moran, MrF, Prakash Murali, Jan Müggenburg, David Nadlinger, Ken Nakanishi, Giacomo Nannicini, Paul Nation, Edwin Navarro, Yehuda Naveh, Scott Wyman Neagle, Patrick Neuweiler, Pradeep Niroula, Hassi Norlen, Lee James O'Riordan, Oluwatobi Ogunbayo, Pauline Ollitrault, Steven Oud, Dan Padilha, Hanhee Paik, Simone Perriello, Anna Phan, Francesco Piro, Marco Pistoia, Alejandro Pozas-iKerstjens, Viktor Prutyanov, Daniel Puzzuoli, Jesús Pérez, Quintiii, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Julia Rice, Diego M. Rodríguez, RohithKarur, Max Rossmannek, Mingi Ryu, Tharrmashastha SAPV, SamFerracin, Martin Sandberg, Hayk Sargsyan, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Zachary Schoenfeld, Travis L. Scholten, Eddie Schoute, Joachim Schwarm, Ismael Faro Sertage, Kanav Setia, Nathan Shammah, Yunong Shi, Adenilton Silva, Andrea Simonetto, Nick Singstock, Yukio Siraichi, Iskandar Sitdikov, Seyon Sivarajah, Magnus Berg Sletfjerding, John A. Smolin, Mathias Soeken, Igor Olegovich Sokolov, SooluThomas, Dominik Steenken, Matt Stypulkoski, Jack Suen, Shaojun Sun, Kevin J. Sung, Hitomi Takahashi, Ivano Tavernelli, Charles

Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Enrique de la Torre, Kenso Trabing, Matthew Treinish, TrishaPe, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Almudena Carrera Vazquez, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Rafal Wieczorek, Jonathan A. Wildstrom, Robert Wille, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, Steve Wood, James Wootton, Daniyar Yeralin, Richard Young, Jessie Yu, Christopher Zachow, Laura Zdanski, Christa Zoufal, Zoufalc, a matsuo, adekusar drl, azulehner, bcamorrison, brandhsn, chlorophyll zz, dan1pal, dime10, drholmie, elfrocampeador, faisaldebouni, fanizzamarco, gadial, gruu, jliu45, kanejess, klinvill, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sethmerkel, strickroman, sumitpuri, tigerjack, toural, vvilpas, welien, willhbang, yang.luh, yelojakit, and yotamvakninibm. Qiskit: An open-source framework for quantum computing, 2019.

[2] Mehdi Bozzo-Rey and Robert Loredo. Introduction to the ibm q experience and quantum computing. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, page 410–412, USA, 2018. IBM Corp.

[3] Andrew M Childs, Eddie Schoute, and Cem M Unsal. Circuit transformations for quantum architectures. *arXiv preprint arXiv:1902.09102*, 2019.

[4] 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), Sep 2019.

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

[6] Yongshan Ding, Pranav Gokhale, Sophia Fuhui Lin, Richard Rines, Thomas Propson, and Frederic T Chong. Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 201–214. IEEE, 2020.

[7] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.

[8] Jeremy Hsu. Intel's 49-Qubit Chip Shoots for Quantum Supremacy. https://spectrum.ieee.org/tech-talk/computing/hardware/intels-49qubit-chip-aims-for-quantum-supremacy, 2018.

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

[10] Julian Kelly. A Preview of Bristlecone, Google's New Quantum Processor. https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, 2018.

[11] Will Knight. IBM Raises the Bar with a 50-Qubit Quantum Computer. `https://www.technologyreview.com/s/609451/ibm-raises-the-bar-with-a-50-qubit-quantum-computer`, 2017.

[12] Lingling Lao, Hans van Someren, Imran Ashraf, and Carmen G. Almudever. Timing and resource-aware mapping of quantum circuits to superconducting processors, 2020.

[13] 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, 2019.

[14] Norbert M Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017.

[15] Dmitri Maslov. Linear depth stabilizer and quantum fourier transformation circuits with no auxiliary qubits in finite-neighbor quantum architectures. *Physical Review A*, 76(5), Nov 2007.

[16] 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*, ASPLOS '19, pages 1015–1029, New York, NY, USA, 2019. ACM.

[17] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[18] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue

solver on a photonic quantum processor. In *Nature Communications*, volume 5, page 4213, 2014.

[19] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[20] QISKit: Open Source Quantum Information Science Kit. `https://https://qiskit.org/`.

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

[22] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

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

[24] 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. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[25] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113–125. ACM, 2018.

[26] Bochen Tan and Jason Cong. Optimal layout synthesis for quantum computing. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[27] 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*, ASPLOS '19, pages 987–999, New York, NY, USA, 2019. ACM.

[28] 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 Annual Design Automation Conference 2019, page 142. ACM, 2019.

[29] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W Dueck, and Rolf Drechsler. Revlib: An online resource for reversible functions and reversible circuits. In *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, pages 220–225. IEEE, 2008.

[30] Chi Zhang, Yanhao Chen, Yuwei Jin, Wonsun Ahn, Youtao Zhang, and Eddy Z Zhang. A depth-aware swap insertion scheme for the qubit mapping problem. *arXiv preprint arXiv:2002.07289*, 2020.

[31] Chi Zhang, Yanhao Chen, Yuwei Jin, Wonsun Ahn, Youtao Zhang, and Eddy Z Zhang. Slackq: Approaching the qubit mapping problem with a slack-aware swap insertion scheme. *arXiv preprint arXiv:2009.02346*, 2020.

[32] Chi Zhang, Ari Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. Time-optimal qubit mapping. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, Virtual, 2021. ACM.

[33] Chi Zhang, Ari B Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z Zhang. Time-optimal qubit mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 360–374, 2021.

[34] Alwin Zulehner, Stefan Gasser, and Robert Wille. Exact global reordering for nearest neighbor quantum circuits using A∗. In *International Conference on Reversible Computation*, pages 185–201. Springer, 2017.

[35] Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the ibm qx architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1135–1138. IEEE, 2018.