University of California
Santa Barbara

Towards High-Performance, Efficient, and Reliable Quantum Computing System

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Gushu Li

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Yufei Ding, Co-Chair
Professor Timothy Sherwood
Professor Zheng Zhang

December 2022

The Dissertation of Gushu Li is approved.

Professor Timothy Sherwood

Professor Zheng Zhang

Professor Yufei Ding, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

October 2022

Towards High-Performance, Efficient, and Reliable Quantum Computing System

by

Gushu Li

To my friends and family

Acknowledgements

The dissertation is not possible without the help of many people. I would first thank my advisors Dr. Yuan Xie and Dr. Yufei Ding. Yuan's background is more on the hardware side while Yufei's background is more on the software side. Working with both of them simultaneously equips me with a holistic view of the entire system and enables many more research opportunities. They have always been supportive and willing to encourage me to explore new research directions. They helped me build courage and confidence, which I believe are the most important things for an independent researcher. I will be trying to convey such mental strength to the next generation.

I am also grateful to my other committee members, Dr. Timothy Sherwood and Dr. Zheng Zhang, for their valuable feedback about my research and helpful suggestions about my job search. And I learned a lot of writing skills by reading Dr. Sherwood's award papers.

Then I want to thank Ms. Val de Veyra who helped me with a lot of paperwork and administration matters. On my first day at UCSB ECE department, I was told that 'whatever help you need, you can always ask Val'. In the last five and a half years, I emailed her or walked into her office numerous times and she can always provide the help I need.

I also want to send my greetings to Dr. Shuangchen Li and Dr. Xiang Fu who helped me a lot at the very beginning of my Ph.D. study. I first worked with Shuangchen on processing in memory after I joined UCSB. I learned many practical research methodologies and research experiences from him. These experiences have been helping me even after I changed to the quantum computing area. After I changed the research topic, I had very little idea about quantum computing. Xiang was already known for his work on quantum control architecture at that time. He shared with me a lot of valuable guidance

during my hard time and help me get onto the right track quickly.

My two joyful summer internships are not possible without my mentor at IBM, Dr. Ali Javadi-Abhari. Ali is a great collaborator who always supports my idea, find the resource I need, and provide me with a lot of industry insights. I am also fortunate to collaborate with Dr. Yunong Shi. Yunong has a strong background in quantum computing. We have been frequently discussing quantum computing research over phone calls in the last few years. Many of my research ideas come from these discussions.

I want to thank my theory collaborators, Dr. Li Zhou, Dr. Nengkun Yu, and Dr. Mingsheng Ying. Our collaboration help me get into the programming language research area and resulted in the first distinguished award paper about quantum programming language at major programming language conferences.

A lot of people helped me during my job search. They include Dr. Zhaoran Wang, Dr. Xiaodi Wu, Dr. Yunong Shi, Dr. Qian Zhang, Dr. Hongye Hu, Dr. Xing Hu, Dr. Shuangchen Li, Dr. Tevfik Bultan, Mr. Xueyang Wang, Mr. Haoxiang Wang, etc. I would also thank my labmates, including Maohua Zhu, Liu Liu, Abanti Basak, Fengbin Tu, Jiayi Huang, Xueqi Li, Yu Ji, Pengfei Zuo, Peng Gu, Wenqin Huangfu, Xinfeng Xie, Ling Liang, Yuke Wang, Boyuan Feng, Zheng Qu, Jilan Lin, Nan Wu, Bangyan Wang, Zhaodong Chen, Guyue Huang, Zheng Wang, Hao Li, Siqi Li, Keyi Yin, Zhaohui Yang. I am fortunate to meet you during my Ph.D. journey and wish you all the best.

Last but not the least, I would like to thank my parents, Mr. Zhenping Li and Mrs. Fengsen Wu, for their continuous unconditional support. And I would like to thank my wife Tianqi Tang for her love and support over the last eight years, from Beijing to Santa Barbara.

Curriculum Vitæ

Gushu Li

Education

2022	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2021	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2015	B.E. in Electronic Engineering, Tsinghua University.

Publications

[C1]. Anbang Wu, Hezi Zhang, **Gushu Li**, Alireza Shabani, Yufei Ding, Yuan Xie, "A Synthesis Framework for Stitching Surface Code with Superconducting Quantum Devices", *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[C2]. Anbang Wu, **Gushu Li**, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, Yuan Xie, "A Synthesis Framework for Stitching Surface Code with Superconducting Quantum Devices", *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2022.

[C3]. **Gushu Li**, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, Yuan Xie, "Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[C4]. **Gushu Li**, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, Yuan Xie, "On the Co-Design of Quantum Software and Hardware", *ACM International Conference on Nanoscale Computing and Communication (NanoCom)*, *invited paper*, 2021.

[C5]. Boyuan Feng, Yuke Wang, **Gushu Li**, Yuan Xie, Yufei Ding, "Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew", *USENIX Annual Technical Conference (ATC)*, 2021.

[C6]. Yuke Wang, Boyuan Feng, **Gushu Li**, Shuangchen Li, Lei Deng, Yuan Xie, Yufei Ding. "GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[C7]. **Gushu Li**, Yunong Shi, Ali Javadi-Abhari, "Software-Hardware Co-optimization for Computational Chemistry on Superconducting Quantum Processors", *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2021.

[C8]. Yuke Wang, Boyuan Feng, **Gushu Li**, Georgios Tzimpragos, Lei Deng, Yuan Xie, Yufei Ding, "TiAcc: Triangle-inequality based Hardware Accelerator for K-means on FPGAs", *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021.

[C9]. **Gushu Li***, Li Zhou*, Nengkun Yu, Yufei Ding, Mingsheng Ying, Yuan Xie, "Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs", *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

[C10]. **Gushu Li**, Yufei Ding, Yuan Xie, "Eliminating Redundant Computation in Noisy Quantum Computing Simulation", *Design Automation Conference (DAC)*, 2020.

[C11]. **Gushu Li**, Yufei Ding, Yuan Xie, "Towards Efficient Superconducting Quantum Processor Architecture Design", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[C12]. **Gushu Li**, Yufei Ding, Yuan Xie, "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[C13]. **Gushu Li**, Guohao Dai, Shuangchen Li, Yu Wang, Yuan Xie, "GraphIA: An In-situ Accelerator for Large-scale Graph Processing", *International Symposium on Memory Systems (MEMSYS)*, 2018.

[C14]. **Gushu Li**, Xiaoming Chen, Guangyu Sun, Henry Hoffmann, Yongpan Liu, Yu Wang, Huazhong Yang, "A STT-RAM-based low-power hybrid register file for GPGPUs", *Design Automation Conference (DAC)*, 2015.

[J1]. Yuke Wang, Boyuan Feng, **Gushu Li**, Lei Deng, Yuan Xie, Yufei Ding. "STPAcc: Structural TI-based Pruning for Accelerating Distance-related Algorithms on CPU-FPGA Platform." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

Please Note: Text, tables, and figures from these papers are used and appear in this dissertation.

Abstract

Towards High-Performance, Efficient, and Reliable Quantum Computing System

by

Gushu Li

As the new "race to the moon", quantum computing can possibly trigger a computation revolution due to its strong potential in several important domains, e.g., cryptography, chemistry simulation, optimization, and machine learning. However, as an emerging research area, grand challenges remain ahead since state-of-the-art quantum computing, from software to hardware, is still highly immature. This dissertation explores high-performance, efficient, and reliable quantum computing systems, and strikes a synergy among different technology stacks, including application, programming language, compiler optimization, hardware architecture design, and simulation. In particular, this dissertation focuses on two directions: 1) cross-layer co-design for quantum computing system; and 2) enabling deep quantum software/compiler optimizations at the high level. In the first direction, this dissertation studies how to efficiently map quantum software to hardware via carefully designed compiler optimization, and then investigates the application-specific architecture design with substantial hardware efficiency improvement. Following the application-specific principle and putting the algorithm optimization and hardware design together, this dissertation proposed a software-hardware co-optimization for chemistry simulation and achieved a wide range of benefits across multiple system stacks. In the second direction, this dissertation explores leveraging the algorithmic information, which is usually carried by new high-level programming languages, to design quantum software optimizations that are hard to implement in conventional quantum software infrastructures. These optimizations include a Pauli-string-based intermediate

representation for large-scope compiler optimization on quantum simulation programs, a projection-operator-based runtime assertion language for efficient quantum program testing and debugging, and a trial scheduling technique to identify and eliminate redundant computation in noisy quantum computing simulation.

Contents

Chapter 1

Introduction

The idea of building a quantum computer started from the 1980s when physicists found a theoretical quantum Turing machine model [1] and later it is pointed out that such a quantum machine may simulate a quantum system that cannot be simulated efficiently on a classical computer [2]. A milestone in the development of quantum computing is Shor's algorithm [3] which, proposed by Peter Shor in 1994, can factor a large integer in polynomial time and thus attack the widely-used RSA-based cryptography system. These early efforts kicked off the second quantum revolution [4].

After the proposal of building a quantum computer, people started to attack two research directions. First, on the theory side, we need to figure out the applications that can potentially be accelerated using a quantum computer. Till now, there have been hundreds of quantum algorithms developed for different purposes (e.g., chemistry simulation [5], combinatorial optimization [6], search [7]) with different speedups [8]. Second, on the practical side, which technology we can use to physically implement a quantum computer? There are several candidate technologies (e.g., superconducting quantum circuit [9], ion trap [10], photonics [11]) but it is not yet known which one will finally win this race.

Recent progress towards practical quantum computing has been inspiring and the advantage of quantum computing has been experimentally demonstrated. In 2019, Google announced that 'quantum supremacy', which is a computational task that can be efficiently done using a quantum computer but is hard to solve on a classical computer, has been achieved in the random circuit sampling [12]. Later, the 'quantum supremacy' was also demonstrated on the Gaussian boson sampling task [13, 14, 15]. However, all the quantum supremacy tasks are basically sampling from a random distribution and it is hard to find practical applications from the random distribution sampling.

Naturally, the next step in the development of quantum computing is to demonstrate practical quantum advantage. This requires efforts from multiple technology stacks in a quantum computer system. For now, quantum computing is in its vacuum-tube era or even pre-vacuum-tube era. A classical analogy can be the time when the first general-purpose digital classical computer ENIAC was developed [16].

1.1 Overview

This dissertation features the emerging **Quantum Computer System** and studies how to deeply optimize the effectiveness, efficiency, and reliability of quantum computation by attacking critical problems at multiple technology stacks. The works presented in this dissertation can be categorized in two research directions: 1) quantum computing system cross-layer co-design [17, 18, 19, 20], and 2) enabling deep quantum software/-compiler optimizations at high-level [21, 22].

1.1.1 Cross-Layer Co-Design in Quantum Computing System

The current quantum computing system follows a vertically layered design similar to its classical counterpart: algorithm - programming language - compiler - architec-

ture - device. Such a layered structure can simplify the system design by hiding the details inside each individual layer, but it also misses optimization opportunities since the information flow across different layers can be blocked by the layer abstractions. Moreover, the design objectives of different layers may contradict each other, which will limit the overall improvement when optimizing different system stacks individually. This dissertation [18, 19, 20, 17] strikes a synergy among different technology stacks and optimizes the quantum computing system by mutually exposing key information between the quantum software and hardware. In particular, we vision that an array of quantum computing accelerators, each of which is tailored to a specific application, is much more likely to be adopted with relatively modest resource requirements. By introducing the *application-specific design principle into quantum computing*, our work is able to coordinate optimizations at different stacks and then outperform the direct combination of individually optimizing the different layers.

Qubit Mapping on Superconducting Devices

My research on qubit mapping (ASPLOS'19 [18]) studies how to map logical qubits to physical qubits on a superconducting architecture. This work has been adopted by several quantum compilers including IBM's Qiskit and the qcor compiler by Oak Ridge National Lab. On a superconducting quantum processor, two-qubit gates are only supported on physically nearby qubits while the logical two-qubit gates can be applied on arbitrary two logical qubits. A quantum compiler has to decide the logical-to-physical qubit mapping and meanwhile dynamically insert additional SWAP operations to remap some qubits so that all two-qubit gate dependencies in the input program are satisfied. It is desirable to reduce the number of inserted SWAPs as more SWAPs will increase the error. Yet, this optimization problem is indeed NP-hard and an efficient and effective heuristic is missing before our work. We propose a fast yet effective algorithm for this

qubit mapping problem. We first change the traditional mapping-transition-based search to a new SWAP-sequence-based search. This reduces the search space greatly since the number of possible SWAPs is much smaller than that of possible mappings. We also propose an inverse-search method to optimize the initial mapping based on a key observation that the qubit mapping problem is reversible. The final mapping of a reserved quantum circuit can improve the initial mapping of the original circuit. To accommodate quantum devices with different characteristics, we design a decay effect to let the compiler tend to select non-overlapping SWAPs to reduce the circuit depth and control the trade-off between gate count and circuit depth. Compared with state-of-the-art solutions, our algorithm can generate comparable or better mapping results ($\sim$ 10% gate count reduction) in a much shorter time (100x-1000x speedup) for better scalability.

Efficient Superconducting Quantum Processor Architecture Design

For superconducting quantum processors, people would like to integrate more qubits and more qubit connection in one substrate to support the execution of larger quantum programs and reduce the qubit mapping overhead. However, more computation resources will also increase the fabrication difficulty and lower the yield rate. Seeking both the high yield rate and the high performance simultaneously for a superconducting quantum processor design is hard due to this intrinsic trade-off. My research explores the efficient superconducting quantum processor architecture design via the application-specific design principle. In particular, we propose an end-to-end design flow (ASPLOS'20 [19]) to automatically extract program information and then generate a superconducting quantum processor architecture that can support the target program with both high performance and high yield rate. This first stage of the design is to extract two-qubit gate patterns in the target application because executing two-qubit gates is the performance bottleneck and its underlying hardware support introduces the fabrication complexity. The second

stage is a hardware design flow with three key subroutines, i.e., layout design for qubit location, bus selection for qubit connection, and frequency allocation for physical qubit design. Each subroutine focuses on different hardware resources and cooperates with corresponding profiling results and physical constraints. More hardware resources are deployed on specific locations only when they are expected to benefit the performance most. Experiments show that our design flow could outperform IBM's general-purpose designs with better Pareto-optimal results, e.g., magnitudes of yield improvement with negligible performance loss.

Software Hardware Co-Optimization for Quantum Computational Chemistry

Variational Quantum Eigensolver (VQE) for quantum chemistry is one leading candidate application for near-term quantum computers to demonstrate a quantum advantage with practical usage. However, conventional setup for VQE on near-term superconducting quantum processors cannot accommodate simulating large-size chemical systems because it suffers from the large circuit size, the limited hardware resource, and the deficient compiler optimization. Optimizations have been made at different system stacks, but they did not come collaboratively, leading to insufficient overall improvement. My research identifies a Pauli-string-centric software-hardware co-optimization (ISCA'21 [20]) that can coordinate optimizations at the algorithm level, hardware level, and compiler level. Each of the proposed optimizations not only focuses on the design objectives of one individual technology but also considers the optimizations in other parts of the stack. This is possible because Pauli strings are the central building blocks of quantum simulation circuits whose two-qubit gate pattern and synthesis flexibility can guide the hardware and compiler design. Based on this observation, we propose 1) a hardware-friendly VQE circuit pruning method with minimal accuracy loss 2) a X-Tree superconducting architecture that can execute Pauli string simulation circuits using a minimal number of

connections, and 3) a compilation algorithm to deploy the pruned circuits onto X-Tree architecture with negligible overhead. Experimental results show that our co-optimization outperforms conventional VQE setups with significant program size reduction, faster convergence speed, mild simulation accuracy loss, more efficient hardware design, and negligible compilation mapping overhead.

1.1.2 Deeper Quantum Program Optimization at High Level

Quantum software, which will deploy and optimize the quantum programs onto the underlying physical quantum hardware platforms, is essential and critical in a quantum computing system. Yet, today's quantum compiler/software infrastructures are still far from optimal. One reason is that most optimizations in today's quantum compilers are local program transformations over very few qubits and gates. In general, it is conceptually hard for a compiler that runs on a classical computer to automatically derive large-scale quantum program optimizations at the assembly gate level. My research [22, 21] tackles this challenge and systematically enhances the quantum software frameworks by introducing *high-level program optimizations in the quantum domain*. Instead of optimizing the quantum programs at the gate level, we design new quantum programming language primitives and intermediate representations that can encode the high-level semantics of the programs. Such high-level information can then be leveraged to derive new large-scale quantum program optimizations beyond the capabilities of conventional gate-level optimizations.

Large-Scale Quantum Compiler Optimization with High-Level Intermediate Representation.

We develop *Paulihedral* (ASPLOS'22 [21]), an algorithmic quantum compiler framework for the quantum simulation kernel which is a subroutine widely used in many algorithms. This framework has been adopted by IBM's Qiskit and Amazon's Braket. Paulihedral enables large-scale compiler optimizations via a new formal high-level intermediate representation (IR), namely Pauli IR. This IR is built upon the Pauli strings, the central building blocks of quantum simulation kernels. Pauli IR can efficiently encode high-level algorithmic information in a quite compact form where the operator size grows linearly with respect to the number of qubits. The follow-up program analysis and optimization can be performed on this IR without processing the quantum gate matrices whose sizes grow exponentially. The syntax and semantics of Pauli IR are defined to uniformly represent simulation kernels of different forms and constraints from all quantum algorithms, as far as we know, and accommodate various backends (i.e., the fault-tolerant quantum computer and the near-term superconducting quantum processor). Several new optimizations are proposed to leverage the Pauli algebra and reconcile multiple optimization factors, including instruction scheduling, circuit synthesis, gate cancellation, and qubit layout/routing. These optimizations are hard to implement in existing quantum compilers because reconstructing high-level semantics in today's compiler infrastructures, in which the programs are represented by assembly-style low-level gate sequences, is extremely hard. Experimental results show that our work can significantly outperform state-of-the-art quantum compilers with more effective, scalable optimizations, and better reconfigurability.

Projection-Based Quantum Program Assertion

We develop the projection-based quantum program assertion (OOPSLA'20 [22]) to advance quantum program testing, debugging, and error mitigation. This work won the Distinguished Paper Award and has been adopted by the Quantinuum's t|ket⟩ framework. As in its early stage, the basic testing and debugging approaches are not yet available or well-developed for quantum programs before our work. The predicates of quantum programs in prior quantum program testing works are usually expressed in classical logic languages which limits the testing capability. The measurement operations in the assertion checking may destroy the tested quantum state, leading to a highly inefficient checking procedure. We observe that the quantum states can be quantified by linear subspaces of the entire system state space and such linear subspaces can be naturally expressed by projection operators. We then formally define our new quantum program assertion primitive upon the projection operator. Compared with previous quantum assertions that express quantum state predicates in classical logical languages, projections have much greater logical expressive power and unique hardware-friendly property as it naturally aligns with the projective measurement operations in most quantum hardware platforms. We rigorously prove the statistical checking efficiency and propose several compilation techniques to resolve machine constraints. To the best of our knowledge, this is the first runtime assertion scheme with such expressive predicates and practical implementation details. We remark that our projection-based assertion can also be applied as an error mitigation technique to filter out hardware noise and improve the overall fidelity of the results of a quantum program beyond its usage of debugging. For example, Quantinuum has utilized our projection-based assertion technique in their molecular simulation experiment on an ion-trap-based quantum computer. Their results confirmed that adding a projection-based assertion can reduce about 50% error of the final molecular

energy estimation.

1.1.3 Architectural Modeling and Simulation

Due to the limited access to the real quantum computing systems, simulation is an important approach when proposing and evaluating quantum computing system innovations without accessing realistic hardware. Since a complete NISQ system consists of two major components, the quantum processor and its classical control system, a simulator for NISQ systems needs to meet the following requirements: 1) simulating a noisy quantum processor, and 2) simulating a classical control system. In this dissertation, we propose a simulation framework, namely SANQ, for NISQ computing system design and evaluation [23]. SANQ consists of one noisy quantum computing simulator for the quantum processor, and one architectural simulation infrastructure to construct behavior models for the classical control system, leading to a comprehensive evaluation of NISQ systems and preparing for future design innovations.

Noisy quantum computing simulation that could consider various noise effects is widely used in algorithm development and device performance evaluation. In Monte Carlo (MC) noisy simulation, noise effects can be treated as errors that are randomly injected during the computation. To model such random effects, the same input quantum program is simulated for a large number of times, and in each simulation trial, errors are randomly injected based on an error model of the target quantum device. Previous optimizations focused on single-trial simulation optimization while little consideration has been given to inter-trial optimization. My research optimized the MC noisy quantum simulation by eliminating the great computation redundancy among those MC simulation trials [24]. It is possible that some share MC simulation trials share the same intermediate states which can be temporarily stored and reused to save computation.

However, saving a state takes significant memory space, which may limit the size of the simulated program. Therefore, it would be desirable to remove redundant computation with the stored intermediate state as few as possible. We propose a trial reorder scheme to 1) efficiently identify and remove the computation redundancy in the MC simulation, and 2) minimize the number of stored intermediate states. Instead of direct running the MC simulation, all the simulation trials are first generated without actually running the simulation. These trials are analyzed and reordered based on the locations of the injected errors. The overlapped computation between two consecutive trials is maximized so that saving one intermediate state can save more computation. Those intermediate states that cannot be reused in the follow-up computation are dropped to reduce the memory requirement. Experiment results show that we can save around 80% computation on average with only a small number of state vectors stored at most on a realistic device model. The test on larger-size device models demonstrates that our noisy quantum computing simulation optimization has great scalability as it could save even more computation when simulating future NISQ devices with lower error rates and more simulation trials.

The architectural simulation infrastructure is specifically designed for control system design. Users can construct a behavior model for a classical control system with provided common hardware modules or with customized newly designed hardware components. SANQ currently focuses on the execution fidelity and timing simulation, which are both critical in NISQ system evaluation, while it is extensible to accommodate more simulation, e.g., power and reliability.

1.2 Outline

The rest of this dissertation will detail all these works mentioned above and is organized as follows. We start from the necessary background about quantum computing in

Chapter 2. Then we first introduce the SABRE qubit mapping algorithm and the efficient superconducting quantum procesor architecture design in Chapter 3 and Chapter 4, respectively. Putting software and hardware together, we explore the software-hardware co-design for quantum chemistry simulation on superconducting quantum processors in Chapter 5. The high-level quantum program optimization framework for quantum simulation, Paulihedral, is introduced in Chapter 6, followed by the projection-based quantum program assertion in Chapter 7. Our architectural modeling and simulation are in Chapter 8. Finally, we conclude this dissertation and discuss the future research of this dissertation in Chapter 9. The description of the artifact of Paulihedral is available in Appendix A. The proof of the theorems in Chpater 7 is postponed to Appendix B.

Chapter 2

Background

In this chapter, we will provide a brief introduction to quantum computing basic concepts. We try to limit our discussion and only keep the necessary content to help formulate and understand this dissertation. We refer the readers to excellent resources [25] for more details. Quantum computing research spans all technology stacks from high-level theory and algorithm, to mid-level architecture and low-level physics [26, 27, 28]. We will start with the software and theory basics, followed by the hardware background.

There are several theoretically equivalent computing diagrams of quantum computing. This dissertation will focus on the most widely adopted gate-based quantum circuit model. Other computing models like adiabatic quantum computing [29], the measurement-based quantum computing [30], topological quantum computing [31] are not covered.

2.1 Quantum Computing Software

Quantum computing is based on quantum systems evolving under the law of quantum mechanics. The state space of a quantum system is a Hilbert space (denoted by $\mathcal{H}$), a

complete complex vector space with the inner product defined. Naturally, the state of a quantum system is described by the elements in this Hilbert space $\mathcal{H}$. And the operations to manipulate the quantum states are the operators applied on the elements in $\mathcal{H}$.

2.1.1 State Vector and Density Operator

When the state of a quantum system is in an exact state, such an exact state is known as a *pure state*. A pure state is described by a unit vector $|\psi\rangle$ in the state space $\mathcal{H}$ of the target quantum system. The size of the state vector is determined by the number of dimensions of the state space. For example, a qubit (the quantum counterpart of a bit in classical computing) is the basics information processing unit in quantum computing and it has a two-dimensional state space $\mathcal{H}_2 = \{a\,|0\rangle + b\,|1\rangle\}$, where $a, b \in \mathbb{C}$, $|a|^2 + |b|^2 = 1$ and $|0\rangle$, $|1\rangle$ are two computational basis states. Another commonly used basis is the Pauli-X basis, $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Different from classical bit, one qubit can be the linear combination of the two basis states, which can be represented by $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.

Suppose we are using the computational basis and the state of one qubit is $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. The state vector of this state is $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ and we usually denote $|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$. Different from a classical bit which can only in the 0 or 1 state, a qubit can be in the linear combination two basis states $|0\rangle$ and $|1\rangle$, which is called the superposition.

When two or more quantum systems are combined, the overall state space is the tensor product of the state spaces of all individual systems. For a quantum system with n qubits, the state space of the composite system is the tensor product of the state spaces of all its qubits: $\bigotimes_{i=1}^{n} \mathcal{H}_i = \mathcal{H}_{2^n}$. For example, the state of a two-qubit system can be represented by $|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$ in a 4-dimensional Hilbert

space. The state vector is $\begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$. This dissertation only considers finite-dimensional quantum systems because realistic quantum computers only have a finite number of qubits.

When the exact state is unknown, but we know it could be in one of some pure states $|\psi_i\rangle$, with respective probabilities p_i, where $\sum_i p_i = 1$, a density operator ρ can be defined to represent such a mixed state with $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$ ($\langle\psi|$ is the complex conjugate of $|\psi\rangle$). A pure state is a special mixed state. For example, suppose a qubit is in state $|0\rangle$ with a probability of 0.5 and in state $|1\rangle$ with a probability of 0.5. Then its density operator is $\rho = 0.5|0\rangle\langle 0| + 0.5|1\rangle\langle 1|$. Using the computational basis, this density operator can be represented by a 2×2 matrix $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$. Note that this mixed state is different from the pure state $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ whose density operator is $\rho = (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle)(\frac{1}{\sqrt{2}}\langle 0| + \frac{1}{\sqrt{2}}\langle 1|)$ with matrix form of $\begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$. In general, the matrix of a density operator for an n-qubit system is a complex matrix of size $2^n \times 2^n$.

2.1.2 Gate/Unitary Transformation

To perform computation over the quantum states, we need to manipulate the quantum states. Such dynamics are called quantum operations in the most general case. In this dissertation, we focus on two major types of operations performed on a quantum system, unitary transformation (also known as quantum gates) and quantum measurement.

Definition 2.1.1 (Unitary transformation) *A unitary transformation U on a quan-*

tum system in the finite-dimensional Hilbert space $\mathcal{H}$ is a linear operator satisfying

$$UU^\dagger = I_\mathcal{H}$$

where $I_\mathcal{H}$ is the identity operator on $\mathcal{H}$.

We list the definitions of the unitary transformations used in the rest of this dissertation as follows:

Single-qubit gates: H (Hadamard) $= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Two-qubit gates: CNOT(Controlled-NOT, Controlled-X), Swap:

$$\text{CNOT} = |0\rangle\langle 0| \otimes I_2 + |1\rangle\langle 1| \otimes X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \text{Swap} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Three-qubit gates: Toffoli, Fredkin (Controlled-Swap, CSwap):

$$\text{Toffoli} = |0\rangle\langle 0| \otimes I_4 + |1\rangle\langle 1| \otimes \text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\text{Fredkin} = |0\rangle\langle 0| \otimes I_4 + |1\rangle\langle 1| \otimes \text{Swap} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

After a unitary transformation, a state vector $|\psi\rangle$ or a density operator ρ is changed to $U|\psi\rangle$ or $U\rho U^\dagger$, respectively. Here are some examples of the statevector and gate.

Example 2.1.1 (Single-qubit unitary/gate operation) *Suppose we have one qubit whose initial state is* $|\psi\rangle_0 = |0\rangle$. *Its state vector is* $|\psi\rangle_0 = |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. *We apply a Hadamard gate on it. The new state is* $|\psi\rangle_1 = \mathrm{H}|\psi\rangle_0 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

Example 2.1.2 (Multi-qubit system) *Suppose the first qubit is in state* $|\psi\rangle_1 = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ *and we have a second qubit whose state is* $|0\rangle$. *The overall system state is* $|\psi\rangle = |\psi_1\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$. *We usually denote it by* $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$.

Example 2.1.3 (Multi-qubit unitary/gate operation) *Suppose we have two qubits*

whose state is $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$. *We apply a CNOT gate on it. The new state is*

$$|\psi\rangle_2 = \mathrm{CNOT}\,|\psi\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Applying a unitary/gate operation U to a density operator is similar by changing $U\,|\psi\rangle$ to $U\rho U^\dagger$.

2.1.3 Measurement

The gate/unitary transformation introduced above can manipulate the quantum data but cannot 'read' the quantum data. That is, the gate/unitary transformations can modify the data but cannot obtain information from the quantum data. To obtain information from the quantum data, we need to perform the quantum measurement operation.

Definition 2.1.2 (Quantum measurement) *A quantum measurement on a quantum system in the Hilbert space* $\mathcal{H}$ *is a collection of linear operators* $\{M_m\}$ *satisfying*

$$\sum_m M_m^\dagger M_m = I_{\mathcal{H}}$$

.

Different from a classical data read operation which usually does not change the observed data, a quantum measurement operation may change the quantum data measured and destroy the quantum state measured. And a quantum measurement is intrinsically probabilistic and different outcomes are obtained with different probabilities. After a

quantum measurement on a pure state $|\psi\rangle$, an outcome m is returned with probability $p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle$ and then the state is changed to $|\psi_m\rangle = \frac{M_m|\psi\rangle}{\sqrt{p(m)}}$. Note that $\sum_m p(m) = 1$. For a mixed state ρ, the probability that the outcome m occurs is $p(m) = tr(M_m^\dagger M_m \rho)$, and then the state will be changed to $\rho_m = \frac{M_m \rho M_m^\dagger}{p(m)}$.

Example 2.1.4 (Single-qubit measurement) *Suppose we have one qubit in state* $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. *We perform a quantum measurement* $M = \{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$ *on this qubit. Then the probability of observing the outcome* 0 *in this measurement is* $p(0) = \langle\psi| M_0^\dagger M_0 |\psi\rangle = \frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)|0\rangle\langle 0|0\rangle\langle 0|\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}$. *After the measurement, the state is changed to* $|\psi_0\rangle = \frac{M_0|\psi\rangle}{\sqrt{p(0)}} = \frac{|0\rangle\langle 0|\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)}{\sqrt{\frac{1}{2}}} = |0\rangle$. *Note that* $\langle 1|0\rangle = \langle 0|1\rangle = 0$ *since* $|0\rangle$ *and* $|1\rangle$ *formulate an orthonormal basis set. And we have* $\langle 0|0\rangle = \langle 1|1\rangle$ *because the states should be normalized. Similarly, the probability of observing the outcome* 0 *in this measurement is* $p(1) = \langle\psi| M_1^\dagger M_1 |\psi\rangle = \frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)|1\rangle\langle 1|1\rangle\langle 1|\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}$. *After the measurement, the state is changed to* $|\psi_1\rangle = \frac{M_1|\psi\rangle}{\sqrt{p(1)}} = \frac{|1\rangle\langle 1|\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)}{\sqrt{\frac{1}{2}}} = |1\rangle$.

Multi-qubit measurement cases can be generalized [25]. We can also observe that after the measurement, the overall system state is in a mixed state whose density operator is $\rho = \frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$.

2.1.4 Quantum Circuit

Quantum circuit is one of the most widely used diagrams to represent a quantum program. Figure 2.1 shows an example quantum circuit. It is quantum teleportation, the 'hello world' program in quantum computing. In the quantum circuit, each horizontal line represents one logical qubit. There are totally three logical qubits in this quantum teleportation program. The gates and measurement operations are represented by different blocks applied on the lines. The square with the letter 'H' represents a Hadamard

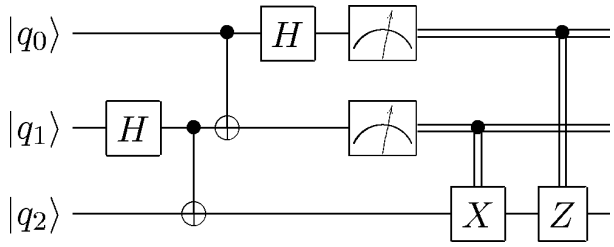


Figure 2.1: Quantum circuit of the quantum teleportation program

gate (defined above). This square symbol is placed on one line, meaning that this gate is applied on the qubit associated with the line. The vertical symbols connecting two lines are the CNOT gates applied on the two connected qubits. On the top right of Figure 2.1, there are two squares with instrument symbols inside them. These squares are the single qubit measurements $M = \{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$ applied on the qubits associated with the lines. The last two X and Z gates are classically conditioned by the measurement results of the first two qubits. When the measurement result is 1, the classically controlled gate is executed. Otherwise, the gate is not executed. To execute a quantum circuit, we need to execute the gates and measurements from the left to the right.

Note that it has been proved that arbitrary quantum circuit can be expressed by compositions of a set of single-qubit gates, CNOT gate, and measurements [32]. On the other hand, many quantum hardware platforms only support single-qubit gates and two-qubit gates. Therefore, in this dissertation, we mostly only consider quantum programs without three-qubit or large-size gates. For example, Figure 2.2 shows a quantum circuit that decomposes the Toffoli gate [25] using only single- and two-qubit gates. The three-qubit gate on the left is Toffoli gate. On the right side is a sequence of single-qubit gates and CNOT gates which implement exactly the same function as a Toffoli gate.

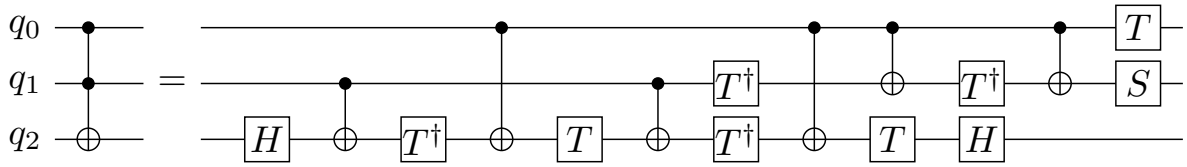


Figure 2.2: Quantum circuit of the Toffoli gate

2.2 Quantum Computing Hardware

After we have the quantum program, we need hardware platforms that can physically implement the qubits and manipulate the qubits specified by the gates/measurements in the program. Several potential technologies are explored and developed to implement quantum machines. It is still not clear which one of them will finally beat the others since they all have some advantages and disadvantages. In this section, we will focus on the superconducting quantum circuit technology as it is the hardware model of most works covered in this dissertation. For other technologies, there are many excellent learning materials, including but not limited to [10, 33, 11].

2.2.1 Superconducting circuit

The superconducting circuit is currently the most popular technology to implement a quantum chip. It is pursued by many physics research groups and many industry leaders like IBM, Google, Intel, Microsoft, etc. The qubits in superconducting circuits are realized by the two lowest energy levels of resonant microwave circuits embedding a Josephson tunnel junction [34, 9]. The gates and measurements are implemented by applying microwave frequency radiation. For more details about the superconducting quantum circuit technology, we recommend [35, 36]

Figure 2.3 shows a picture of IBM's 5-qubit device. There are five dark square structures and they are the five physical superconducting qubits. There are also several wires

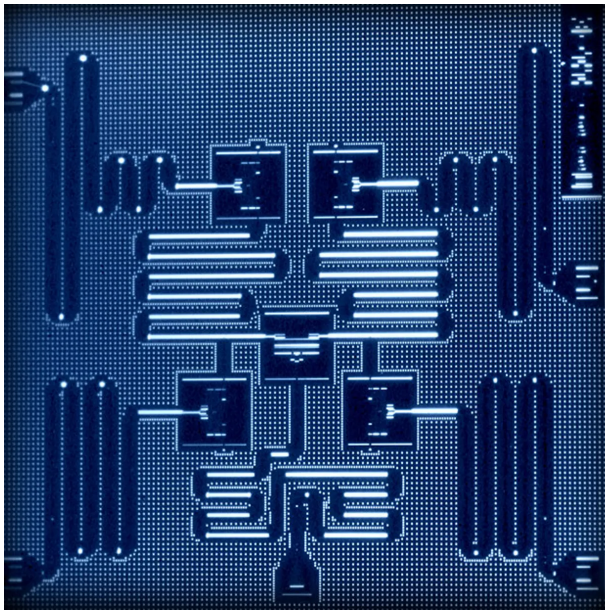


Figure 2.3: IBM 5-qubit chip[37]

that implement connections among the qubits and the I/O ports. These wires are the resonators and are required to implement two-qubit gates and measurements. State-of-the-art quantum processors are at a much larger scale. For example, IBM has announced its 127-qubit chip [38]. Google also announced its 72-qubit chip [39]. Another startup Rigetti also recently released their 40-qubit and 80-qubit devices [40].

Recent Progress

A lot of important quantum computing experiments have been performed using superconducting quantum processors. In 2019, Google announced that 'quantum supremacy' has been achieved on their 53-qubit device [12]. Several quantum error correction code experiments on the superconducting quantum circuit processors have successfully demon-

strated that quantum error correction can suppress the logical error rate [41, 42, 43, 44]. Various algorithms with practical usage are also implemented using this technology [45, 46, 47]. The major challenges of superconducting circuits are the low temperature requirement, circuit cross coupling when scaling up, and the sensitivity to imperfections.

Chapter 3

Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices

3.1 Introduction

Quantum computing has been rapidly growing in the last few decades because of its potential in various important applications, including integer factorization [3], database search [7], quantum simulation [48], etc. Recently, IBM, Intel, and Google released their quantum computing devices with 50, 49, and 72 qubits respectively [49, 50, 51]. IBM and Rigetti also provide cloud quantum computing services [37, 52], allowing more people to study real quantum hardware. We are expected to enter the Noisy Intermediate-Scale Quantum (NISQ) era in the next few years [53], when quantum computing devices with dozens to hundreds of qubits will be available. Though the number of qubits is insufficient for Quantum Error Correction (QEC), .it is expected that these devices will be used to solve real-world problems beyond the capability of available classical computers [54, 55].

However, there exists a gap between quantum software and hardware due to technology constraints in the NISQ era. When designing a quantum program based on the most popular circuit model, it is always assumed that qubits and quantum operations are perfect and any quantum-physics-allowed operations can be applied. But on NISQ hardware, the qubits have limited coherence time, and quantum operations are not perfect. Furthermore, only a subset of theoretically possible quantum operations can be directly implemented, which calls for a modification in the quantum program to fit the target platform.

In this chapter, we will focus on the *qubit mapping problem* caused by limited two-qubit coupling on NISQ devices. Two-qubit gates are one important type of quantum operations applied on two qubits. They can create quantum entanglement, an advantage that does not exist in classical computing. Two-qubit gates can be applied to arbitrary two logical qubits in a quantum algorithm but this assumption does not hold with NISQ devices. When running a quantum program, the logical qubits need to be mapped to the physical qubits (an analogy in classical computation is register allocation). But for the physical qubits on NISQ devices, one qubit can only couple with its neighbor qubits directly. So that for a specific mapping, two-qubit gates can only be applied to limited logical qubit pairs, whose corresponding physical qubit pairs support direct coupling. This makes a quantum circuit not directly executable on NISQ devices.

As a result, circuit transformation is required to make the circuit compatible with NISQ device during compilation. Based on a given quantum circuit and the coupling information of the device, we need 1) an initial logical-to-physical qubit mapping and 2) the intermediate mapping transition which is able to remap the two logical qubits in a two-qubit gate to two coupled physical qubits. The qubit mapping problem has been proved to be NP-Complete [56].

Previous solutions to this problem can be classified into two types. One type is to

formulate this issue into an equivalent mathematical problem and then apply a solver [57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67]. These attempts suffer from very long runtime and can only be applied to small size cases. Moreover, general software solvers can not exploit the intrinsic feature of the quantum mapping problem. Another type of approach is heuristic search [68, 69, 70, 71, 72, 73, 74, 75], while most of them were developed on ideal 1D/2D lattice model and not applicable to NISQ devices with more irregular and restricted coupling connections. Some recent works [76, 56, 77] targeting IBM chip architecture are able to handle arbitrary coupling but they suffer from very long runtime due to exhaustive mapping search, and their solutions for initial mapping lack the ability of global optimization. Moreover, none of them have the ability to control the generated circuit quality among multiple optimization objectives to fit in NISQ devices with different characteristics.

In this chapter, a **S**W**A**P-based **B**idi**RE**ctional heuristic sear-ch algorithm, named SABRE, is proposed to solve this qubit mapping problem and overcome the drawbacks mentioned above. With the observation that many attempts in exhaustive search can be redundant and effective mapping transition needs to start from the qubits in the two-qubit gates that need to be executed, we design an optimized SWAP-based heuristic search scheme in SABRE with significantly reduced search space. Initial mapping has been proved to be very important in this problem since it can significantly affect the final circuit quality [56, 77]. We present a novel reserve traversal search technique in SABRE to naturally generate a high-quality initial mapping through traversing a reverse circuit, in which more consideration is given to those gates at the beginning of the circuit without completely ignoring the rest of the circuit. Moreover, we introduce a *decay* effect, which will slightly increase our heuristic cost function values when evaluating overlapped SWAPs, to let SABRE tend to select non-overlapped SWAPs. This optimization enables the control of parallelism in the additional SWAPs and can further generate different

hardware-compliant circuits with a trade-off between circuit depth and the number of gates.

SABRE is evaluated with various benchmarks on a latest IBM 20-qubit chip model [37] compared with the best known solution [77]. Experimental results show that SABRE is able to find the optimal mapping for small benchmarks and the number of additional gates is reduced by 91% or even fully eliminated. For larger benchmarks, SABRE can demonstrate exponential speedup against the previous solution and still outperform it with around 10% reduction in the number of additional gates on average with the assistance of the high-quality initial mapping generated by our proposed method. In some cases, the best known previous solution cannot even finish execution due to exponential execution time and memory requirement, while SABRE can still work with short execution time and low memory usage. By tuning the decay parameters in our algorithm, SABRE shows the ability to control the generated circuit quality with about 8% variation in generated circuit depth by varying the number of gates.

The major contributions of this chapter can be summarized as follows:

- We perform a comprehensive analysis on the shortcomings of previous solutions, and then summarize the objectives and metrics that should be considered when designing a heuristic solution for the qubit mapping problem.

- We propose a SWAP-based search scheme which can produce comparable results with exponential speedup in the search complexity compared with previous exhaustive mappingsearch algorithms. This fast search scheme ensures the scalability of SABRE to accommodate larger-size quantum devices in the NISQ era.

- We present a reverse traversal technique to enable global optimization in the initial mapping solution by leveraging the intrinsic reversibility in qubit mapping problem. Our high-quality initial mapping can significantly reduce the overhead in the

generated circuit.

- By introducing a decay effect in the heuristic cost function, we are able to generate different hardware-compliant circuits by trading the number of gates in the circuit against the circuit depth. This makes SABRE applicable for NISQ devices with different characteristics and optimization objectives.

3.2 Background

In this section, we will give a brief introduction to quantum hardware in the NISQ era. We will focus on IBM's superconducting quantum processor and the qubit connectivity constraints.

3.2.1 Quantum Computing Hardware in the NISQ Era

There are several different candidate technologies to implement quantum computing on hardware, including superconducting quantum circuit [78], ion trap [79], quantum dot [80], neutral atom [81], etc. We will use superconducting quantum circuit, which is currently the most promising technology, as an example to introduce quantum computing hardware model.

Figure 3.1 shows the information about IBM Q20 chip [37]. The lifetime of the qubits are about 50μs on average. The average error rates are 4.43×10^{-3}, 8.47×10^{-2}, 3.00×10^{-2} for single-qubit gate, measurement, and CNOT gate respectively. As mentioned in Chapter 2, the physical superconducting qubits are connected by physical resonators, also known as couplers. Since the qubits are placed on a planar geometry, the couplers can only connect one qubit to its neighboring qubits due to on-chip placement-and-routing constraints. This qubit connectivity constraint can be abstracted and represented in a

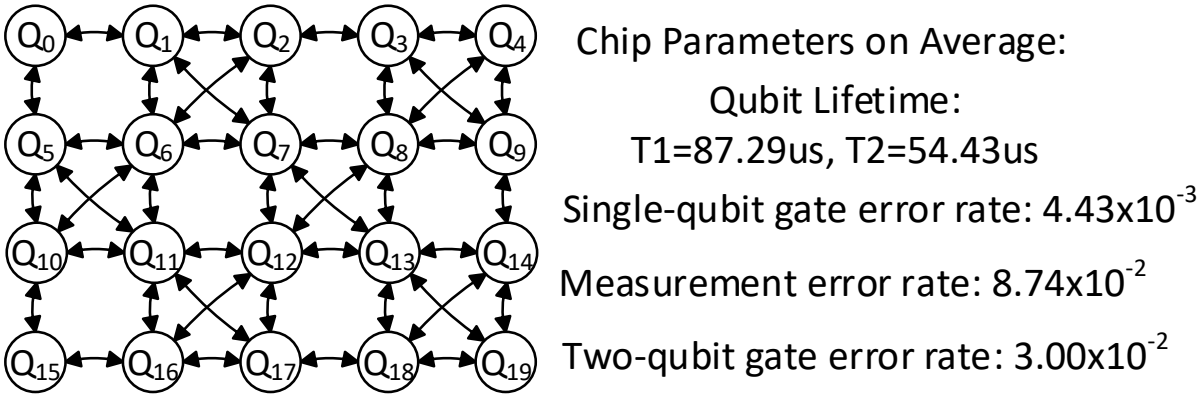


Figure 3.1: IBM Q20 Tokyo Information [37] (Vary over Time)

coupling graph shown on the left. Each node in this graph represents a qubit and two nodes are connected by a bidirectional arrow if their corresponding qubits are physically connected. For example, Q_0 is connected to Q_1 and Q_5 through couplers, which means a CNOT gate can be applied on qubit pair $\{Q_0, Q_1\}$ and $\{Q_0, Q_5\}$ in either direction. However, Q_0 is not directly connected with Q_6 and you cannot apply a CNOT gate on these two qubits directly.

John Preskill proposed this NISQ concept, referring to quantum computers with the number of qubits ranging from dozens to hundreds [53]. Quantum computers of such size are expected to appear in the next few years. Due to limited number of qubits in the NISQ era, all logical qubits in the quantum circuit are directly implemented by physical qubits without QEC. NISQ hardware is not as perfect as the model used when we design a quantum program. In this chapter, the following three major limitations are considered:

1. **Qubit Lifetime.** A qubit can only retain its state for a very short time. It may decay to another state or interact with the environment and lose the original quantum state. The coherence time of state-of-the-art superconducting qubits can reach ~ 100 μs [37]. All the computation must be accomplished within a fraction

of qubit coherence time, which sets an upper bound on the number of sequential gates that can be applied on qubits.

2. **Operation Fidelity.** Quantum operations applied to the qubits can also introduce errors. For example, the error rate for operations is reported to be around 10^{-3} for single-qubit gates, and 10^{-2} for two-qubit gates and measurements [37, 82, 83]. Therefore, it is important to minimize the number of gates in a quantum algorithm to reduce the amount of error accumulated.

3. **Qubits Coupling.** A physical connection is required when applying two-qubit gates, which means that two-qubit gates can only be applied on two physically nearby qubits. One popular coupling structure is the 2D Nearest Neighbor structure which fits in the planar layout of qubits on state-of-the-art superconducting quantum chips.

3.3 Problem Analysis

In this section, we will illustrate the challenge of qubit mapping caused by the three limitations discussed above. We first introduce qubit mapping problem with a small-size example. Then we will discuss the design objectives and the metrics used to evaluate our solution.

3.3.1 Problem in Qubit Mapping

We will use a small-size example to explain this qubit mapping problem. A 4-qubit device model is used as the hardware platform (shown in Figure 3.2 (b)). Two-qubit gates are allowed on the following physical qubit pairs:$\{Q_1, Q_2\}$, $\{Q_2, Q_4\}$, $\{Q_4, Q_3\}$, $\{Q_3, Q_1\}$ and not allowed on $\{Q_1, Q_4\}$, $\{Q_2, Q_3\}$.

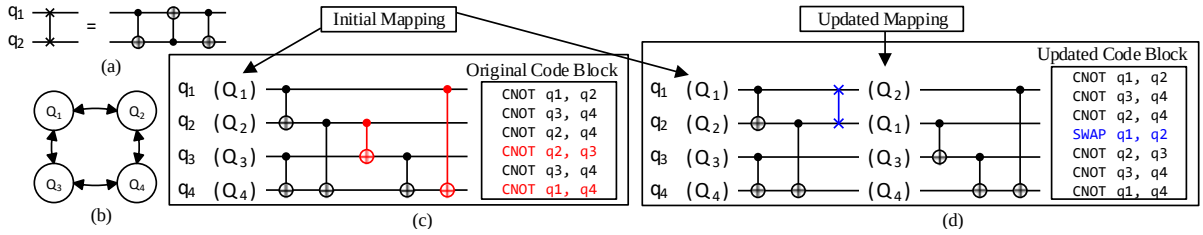


Figure 3.2: (a) SWAP Gate Decomposition, (b) Physical Qubit Coupling Graph Example, (c) Original Quantum Circuit, (d) Updated Hardware-Compliant Quantum Circuit

Now suppose we have a small quantum circuit to be executed on this 4-qubit device. This quantum circuit consists of six CNOT gates (shown in Figure 3.2 (c)). We assume the initial logical-to-physical qubits mapping is $\{q_1 \mapsto Q_1, q_2 \mapsto Q_2, q_3 \mapsto Q_3, q_4 \mapsto Q_4\}$. We can find that four of the six CNOT gates can be directly executed, but the fourth and the sixth CNOT gates (marked red in Figure 3.2 (c)) cannot be executed because the corresponding qubit pairs are not connected on the device. A perfect initial mapping to satisfy all two-qubit gate dependencies does not exist in this example and we need to change the qubit mapping during execution and make all CNOT gates executable.

SWAP Qubit Mapping. Same as previous solutions, we employ SWAP operations to change the qubit mapping by exchanging the states between two qubits. It consists of three CNOT gates (shown in Figure 3.2 (a)). We can employ multiple SWAPs to move one logical qubit to arbitrary physical qubit location. Even two qubits are not nearby on the quantum device, we can still move them together and then apply the two-qubit gate in the circuit. Figure 3.2 (d) shows that the updated quantum circuit is now executable after we insert one SWAP operation between q_1 and q_2 after the third CNOT gates. The first three CNOT gates can be executed under initial mapping. After the inserted SWAP, mapping is updated to $\{q_1 \mapsto Q_2, q_2 \mapsto Q_1, q_3 \mapsto Q_3, q_4 \mapsto Q_4\}$. All three remaining CNOT gates now can be executed under this updated mapping.

Other Methods. Prior work also tried to employ other circuit transformation meth-

ods [56] like 'Reverse' or 'Bridge' because of the asymmetric connection hardware model from IBM's 5-qubit and 16-qubit chips [37]. On those chips, CNOT gate is only allowed in one direction even if two physical qubits are connected on the chip. Fortunately, physical experiments have shown that the connection between superconducting qubits can be symmetric [84] and on IBM's latest 20-qubit chip [37, 85], CNOT gate can be applied on either direction between any connected qubit pair. Since the difficulty from the asymmetric connection is overcome by technology advance, we will focus on the latest symmetric coupling model and only consider inserting SWAPs for mapping change.

By introducing additional SWAPs in the quantum circuit, we can solve all the two-qubit gate dependencies and generate a hardware-compliant circuit without changing the original functionality. However, due to limitations of NISQ devices, inserting SWAPs in the quantum circuit will also cause the following problems:

1. The number of operations in the circuit is increased. Since the operations are imperfect and will introduce noise, the overall error rate will increase.

2. The circuit depth may also be increased, which means the total execution time will be increased and too much error can be accumulated due to qubit decoherence.

If we compare the original circuit and the updated circuit in Figure 3.2 (c) and (d), the number of gates increases from 6 to 9 and the circuit depth increases from 5 to 8. Additional SWAPs will bring significant overhead in terms of fidelity and execution time. As a result, we hope to minimize the number of additional SWAPs in order to reduce the overall error rate and total execution time. We formally define the qubit mapping problem as follows:

Definition: Given an input quantum circuit and the coupling graph of a quantum device, find an **initial mapping** and the intermediate qubit **mapping transition** (by inserting SWAPs) to satisfy all two-qubit constraints and try to minimize the number of

additional gates and circuit depth in the final hardware-compliant circuit.

3.3.2 Objectives and Metrics

Since qubit mapping problem is NP-Complete [56], it is hard to directly find the optimal solution. We will design a heuristic algorithm trying to find a solution to this problem with the following objectives:

1. **Flexibility.** NISQ devices may have an irregular coupling design which can evolve over time. Our algorithm should be able to deal with arbitrary symmetric coupling cases for various benchmarks.

2. **Fidelity.** This objective comes from the imperfect quantum operations. The error rate of a CNOT gate is high and one SWAP even requires 3 CNOT gates. We target to improve the overall fidelity by reducing the number of quantum gates, especially two-qubit gates, of the final hardware compliant circuit.

3. **Parallelism.** This objective comes from the limited qubit lifetime. Inserting SWAPs may increase the depth of the circuit. If our algorithm can insert SWAPs that can be executed in parallel and control the final circuit depth, a deeper circuit will be allowed to execute on hardware.

4. **Scalability.** Our algorithm targets to be scalable with an acceptable execution time for NISQ devices which contain dozens to hundreds of qubits. As the number of qubits continues to increase beyond the scope of NISQ in the future, QEC might be used, and the problem addressed in the chapter turns into another one, as discussed in other papers [86, 87, 88, 89].

Metrics. Our algorithm is evaluated by a set of benchmarks of various sizes on IBM's latest public superconducting chip model [37] to test the flexibility and scalability. The

Table 3.1: Definition of Notations used in this Chapter

Notation	Definition
n	number of logical qubits
$q_{\{1,2,\cdots,n\}}$	logical qubits in quantum circuit
g	number of gates in the circuit
d	depth of the circuit
N	number of physical qubits
$Q_{\{1,2,\cdots,N\}}$	physical qubits on quantum device
$G(V,E)$	the coupling graph of the chip
$D[\,][\,]$	the distance matrix of the physical qubits $D[i][j]$ is the distance between Q_i,Q_j
$\pi()$	a mapping from $q_{\{1,2,\cdots,n\}}$ to $Q_{\{1,2,\cdots,N\}}$
$\pi^{-1}()$	a mapping from $Q_{\{1,2,\cdots,N\}}$ to $q_{\{1,2,\cdots,n\}}$
F	Front Layer, defined in Section 3.4.1
E	Extended Set, defined in Section 3.4.4

metrics are the total number of gates and the circuit depth in the generated hardware-compliant circuit.

3.4 Finding Initial Mapping and SWAPs

In this section, we will introduce our heuristic approach SABRE step by step to illustrate how our design search could overcome the shortcomings of previous work. We start with preprocessing steps in Section 3.4.1 and the overview of SABRE's SWAP-based heuristic search algorithm in Section 3.4.2. Then we use several examples to explain key design decisions in SABRE in Section 3.4.3, followed by the heuristic function design in Section 3.4.4. We summarize the notations used in this chapter in Table 3.1.

3.4.1 Preprocessing

Before our heuristic search, some preprocessing steps are performed to prepare and initialize the required data.

Distance matrix computing. Given the coupling graph $G(V, E)$ of a quantum device, we will first compute the All-Pairs Shortest Path (APSP) by Floyd-Warshall algorithm [90] to obtain the distance matrix $D[\,][\,]$. Each edge in the coupling graph has distance of 1 because one SWAP is required to exchange the two qubits of an edge. So that $D[i][j]$ represents the minimum number of SWAPs required to move a logical qubit from physical qubit Q_i to Q_j. The complexity of this step is $O(N^3)$, which is acceptable for NISQ devices with hundreds of qubits.

Circuit DAG generation. We use a Directed Acyclic Graph (DAG) to represent

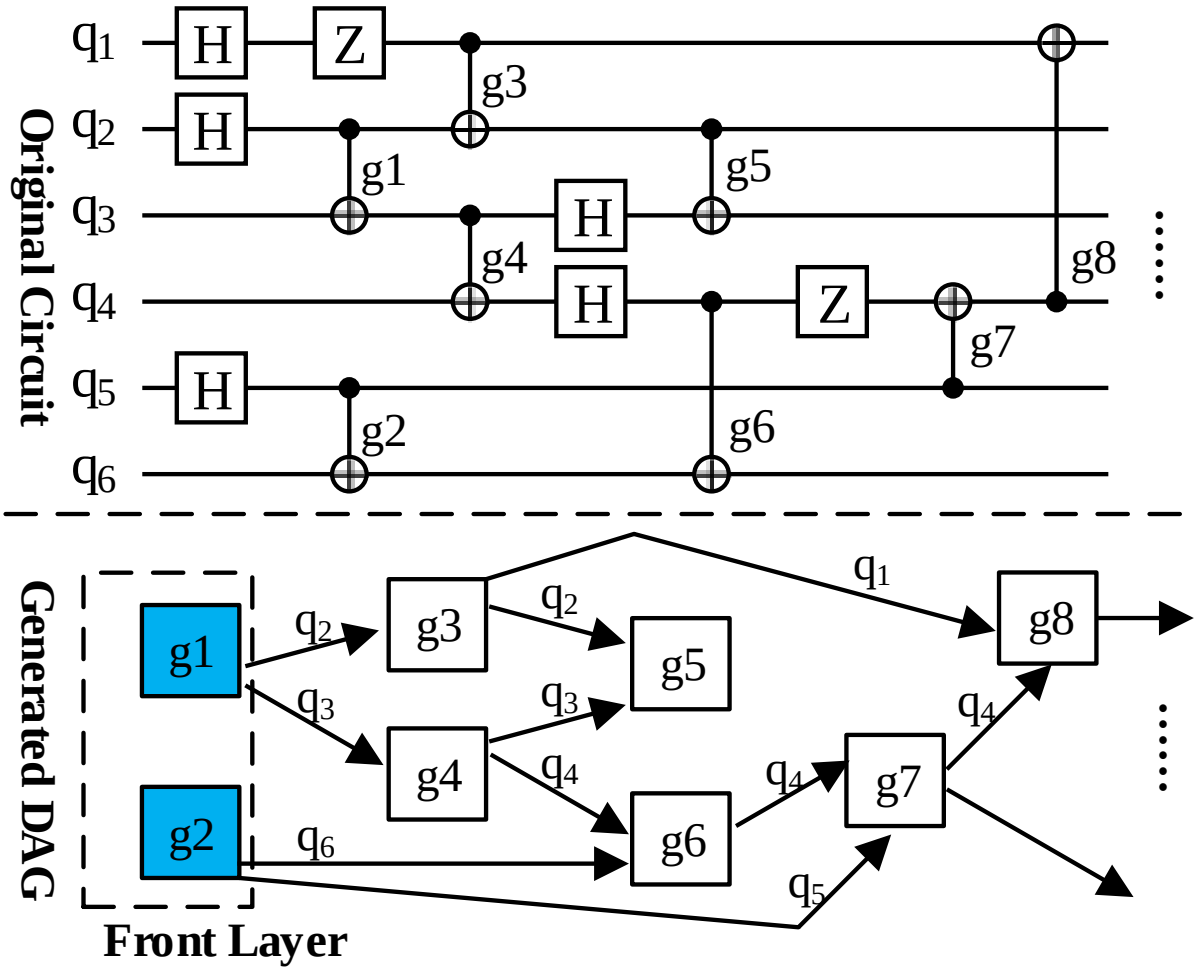


Figure 3.3: Example of DAG Generation and Front Layer Initialization.

the execution constraints between the two-qubit gates in a quantum circuit. The single qubit gates are not considered here because they can always be executed locally on one qubit without bringing dependencies on other qubits. A two-qubit gate $CNOT(q_i, q_j)$ can be executed only when all the previous two-qubit gates on q_i or q_j have been executed. We traverse the entire quantum circuit and construct a DAG to represent execution dependencies with complexity $O(g)$. An example is shown in Figure 3.3. The DAG in the lower half is generated from the quantum circuit above. For example, the gate $g3$ depends on gate $g1$ because qubit q_2 is in both $g1$ and $g3$ can not be executed before $g1$.

Front layer initialization. A front layer (denoted as F) in this chapter is defined as the set of all the two-qubit gates which have no unexecuted predecessors in the DAG. These gates can be executed instantly from a software perspective. For a two-qubit gate $CNOT(q_i, q_j)$, it can be placed in the set F when all previous gates on q_i or q_j have been executed. By checking the generated DAG, we can select all vertices in the graph with 0 indegree, which means the corresponding two-qubit gates have no dependencies, to initialize F. In Figure 3.3, the initial front layer contains $g1$ and $g2$ because they have no predecessors.

Temporary initial mapping generation. SABRE does not give the initial mapping at the preprocessing stage, but a temporary initial mapping is still required to start our heuristic search. We randomly generate an initial mapping as a start point. Later in Section 3.4.3, we will finally update this initial mapping at the end of SABRE.

3.4.2 SWAP-Based Heuristic Search

The preprocessing stage leads to the distance matrix $D[\,][\,]$, circuit DAG, initial F, and an initial mapping. In this section, we introduce the complete SWAP-Based heuristic search procedure.

Algorithm 1: SABRE's SWAP-based Heuristic Search

Input: Front Layer F, Mapping π, Distance Matrix D, Circuit DAG, Chip Coupling Graph $G(V, E)$
Output: Inserted SWAPs, Final Mapping π_f

```
1  while F is not empty do
2      Execute_gate_list = ∅ ;
3      for gate in F do
4          if gate can be executed on device then
5              Execute_gate_list.append(gate);
6          end
7      end
8      if Execute_gate_list ≠ ∅ then
9          for gate in Execute_gate_list do
10             F.remove(gate);
11             obtain successor gates from DAG;
12             if successor gates' dependencies are resolved then
13                 F.append(gate);
14             end
15         end
16         Continue;
17     else
18         score = [];
19         SWAP_candidate_list = Obtain_SWAPs(F, G);
20         for SWAP in SWAP_candidate_list do
21             π_temp = π.update(SWAP);
22             score[SWAP] = H(F, DAG, π_temp, D, SWAP);
23         end
24         Find the SWAP with minimal score;
25         π = π.update(SWAP);
26     end
27 end
```

Algorithm 1 shows the pseudo code of our search algorithm for one traversal, which scans through the entire DAG and inserts SWAPs to make all CNOT gates executable. Later in Section 3.4.3, this procedure will be used multiple times to update the initial mapping and improve the results. Generally, SABRE's heuristic search will iterate until the front layer F is empty, which means all the gates in the circuit have been executed

and the algorithm should stop. In each iteration, it will first check if there are any gates in F that can be directly executed on the chip. If so, it will execute these gates, remove them from F, and then add new gates to F if possible. Otherwise, it will try to search for SWAPs, insert the SWAPs in the circuit, and update the mapping. A detailed explanation of each step is listed as follows:

- Our heuristic search algorithm will first check if F is empty. If so, all the two-qubit gates in the circuits have been executed and we should finish our search algorithm. Otherwise, it will initialize an *Execute_gate_list* and try to add some gates from F to *Execute_gate_list*.

- To determine whether a gate should be added into *Execute_gate_list*, SABRE's search algorithm will extract the logical qubits, q_i and q_j, in the gate and use the current mapping to find the corresponding physical qubits $Q_m, Q_n = \pi(q_i), \pi(q_j)$ on the chip. If Q_m and Q_n are connected by an edge in the coupling graph G, then this two-qubit gate on q_i and q_j can be executed directly and will be added to *Execute_gate_list*.

- If *Execute_gate_list* is not empty, all gates in the list are removed from F. After that, we will check the successor gates of these executed gates. For a successor CNOT gate on q_i and q_j, if there is no gate in F that is applied on any of them, then logically this successor gate is ready to be executed and we will add it to F. After executing some gates and adding the successor gates, we will go back to the beginning and the check for the executable gates again.

- If *Execute_gate_list* is empty, all the gates in F can be executed in software but not on hardware. SWAPs need to be inserted to move the logical qubits in a two-qubit gate close to each other.

- Instead of searching for a mapping, which will require exponential time and space, we only search for SWAPs associated with the qubits in F (in Section 3.4.3). Suppose q_1 is a target of a two-qubit gate in F now, we find the corresponding physical qubit $Q_i = \pi(q_1)$ in G and then locate all its 5 neighbors $Q_{i1}, \ldots, Q_{i5}$. After that we use reverse mapping to find the corresponding logical qubits $q_{i1}, \ldots, q_{i5} = \pi^{-1}(Q_{i1}), \ldots, \pi^{-1}(Q_{i5})$. For logical qubit pairs $(q_1, q_{i1}), \ldots, (q_1, q_{i5})$, it is possible to insert a SWAP between the two qubits in a qubit pair since their corresponding physical qubits are connected by an edge in the coupling graph, and two-qubit gates between these two qubits are supported by the hardware. The SWAPs on these qubit pairs will be added to $SWAP\_candidate\_list$. We repeat the procedure above for all the qubits involved in F.

- A heuristic cost function H is then used to rate each SWAP in the $SWAP\_candidate\_list$. The SWAP with the lowest score is selected to update the mapping π. After that, the algorithm continues to check for executable gates if F is not empty; otherwise, it terminates.

3.4.3 Key Design Decisions

Compared with previous solutions, SABRE features three points to ensure the design objectives can be achieved. Three corresponding examples are given to demonstrate the benefits of our design decisions.

SWAP-Based Search Scheme.

Previous works usually employ mapping-based exhaustive search to find the valid mapping transition with low overhead [71, 77]. For example, Zulehner *et al.* search all possible combination of SWAPs that can be applied concurrently to minimize the output

circuit depth and the number of additional SWAPs at the same time [77]. However, such exhaustive search requires $O(exp(N))$ time and space, which makes the algorithms not applicable to larger-size NISQ devices (experimental results discussed in Section 3.5.2).

We observe that many SWAPs in the mapping-based exhaustive search can be redundant. Figure 3.4 shows an example of how we reduce the search space and find the SWAP. Suppose we have a 9-qubit device. The coupling graph and initial mapping are shown on the right side. The program we need to execute is on the left side. The first two CNOT gates are in the front layer and ready to be executed. The third CNOT needs to be executed after the first one due to the dependency on q_7. The first two gates cannot be executed directly because their corresponding physical qubit pairs are not connected. All qubits not involved in the front layer (q_2, q_4, q_5, q_6, q_9) are considered as low priority ones and any SWAPs inside this low priority qubit set cannot help with resolving dependencies in the front layer. Thus, only the SWAPs that associate with at least one qubit in the front layer (the edges marked red in Figure 3.4) are the candidate SWAPs .

For all the candidate SWAPs, we design a heuristic cost function to help find the SWAP that can reduce the sum of distances between each qubit pairs in the front layer. Moreover, we also enable look-ahead ability in the heuristic cost function by considering the gates right after the front layer. The detailed design of our heuristic cost function is in Section 3.4.4. Here in this example in Figure 3.4, we can find that the SWAP marked by a purple arrow is the best one. It can make all the CNOT gates in the front layer executable and also reduce the distance between q_2 and q_7, which are in a CNOT gate right after the front layer. For the long-term gates far away from the front layer, we temporarily do not consider them because the mapping may vary significantly during execution and it is hard to estimate the cost accurately over a long gate sequence without an exhaustive search.

Complexity Analysis. An upper bound of the computation complexity can be

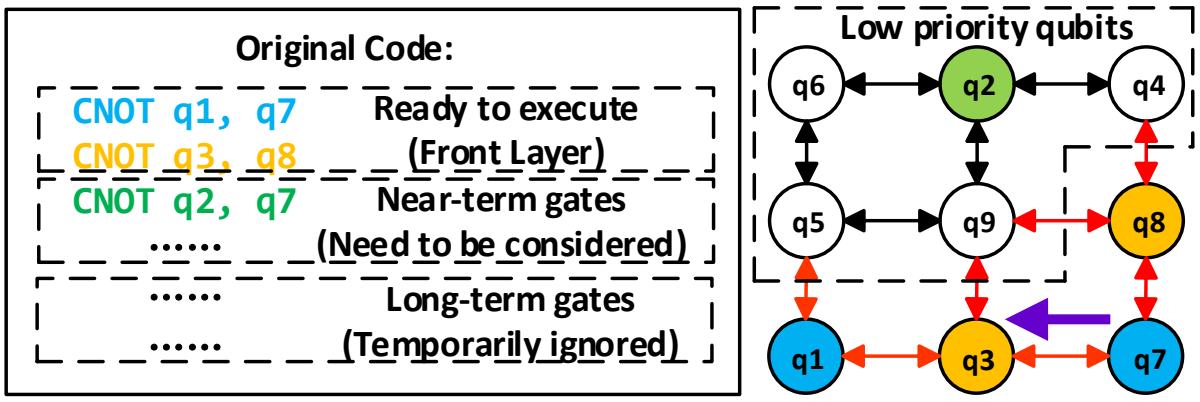


Figure 3.4: Example of SWAP-Based Heuristic Search

estimated by the worst case, in which each two-qubit gate is satisfied individually. The problem is resolved when all two-qubit gates have been satisfied. The time complexity to satisfy one two-qubit gate is the multiplication of the time to evaluate a potential option in the search space, the size of the largest possible search space, and the maximum number of search steps per two-qubit gate. The complexity of the heuristic cost function computation is $O(N)$ (in Section 3.4.4). This SWAP-based search could bring exponential speedup by reducing the search space from $O(exp(N))$ to $O(N)$ (in the worst case all the qubits are involved in the front layer), which makes SABRE scalable to larger size cases. Although it increases the number of search steps because since multiple SWAPs may be needed for one two-qubit gate, the benefit is still significant because we need, at most, the diameter of the chip coupling graph ($O(\sqrt{N})$ for 2D layout) number of SWAPs to move two qubits together for each two-qubit gate. In summary, our SWAP-based search scheme can reduce the complexity from $O(exp(N))$ to at most $O(N^{2.5})$for each two-qubit gate, which makes SABRE exponentially faster as N increases.

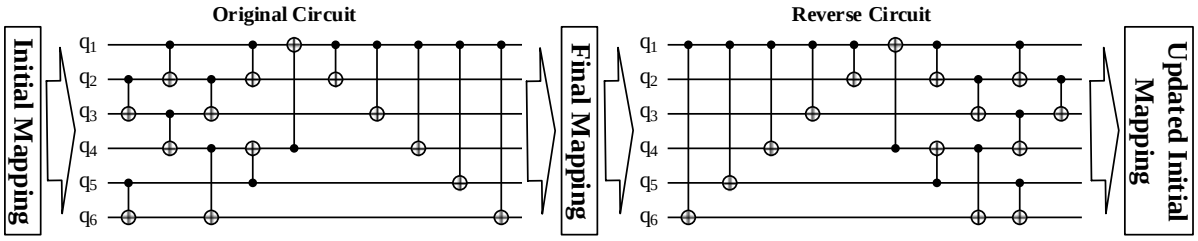


Figure 3.5: Initial Mapping Update Using Reverse Traversal Technique

Reverse Traversal for Initial Mapping.

It has been proved that initial mapping could have a huge impact on the final result [56, 77]. However, no previous solution could give an initial mapping with global consideration. Siraichi *et al.* counted the number of coupled logical qubits in the circuit for each logical qubit and tried to find a match with the outdegree of the physical qubit in the coupling graph with no temporal information considered [56]. Zulehner *et al.* determined the initial mapping by those two-qubit gates at the beginning of the circuit without global consideration [77].

Different from classical circuit or programs, quantum circuits are reversible. You can easily generate a reverse circuit of the original circuit. The two-qubit gates in the reverse circuit will be exactly the same with only the order reversed. Figure 3.5 shows an example of the reverse circuit. The last (first) CNOT gate in the original circuit will be the first (last) CNOT gate in the reverse circuit on the same qubits. This symmetry between the original circuit and the reverse circuit creates a new opportunity for initial mapping optimization. If we know the final mapping of a quantum circuit, we can use this final mapping as the initial mapping to solve qubit mapping problem for the reverse circuit on the same hardware model. The final mapping of the reverse circuit can be an initial mapping for the original circuit. This updated initial mapping comes with better quality because all the gates' information is considered. The gates that are closer to the beginning of the circuit will have more impact on the initial mapping optimization. The

gates far away from the beginning have less impact but can still be considered through these forward and backward traversals.

Based on this observation, we propose a novel reverse traversal technique to generate high-quality initial mapping with global information considered. Figure 3.5 illustrates the procedure of this technique. We first randomly generate an initial mapping and then apply our SWAP-based heuristic search to traverse through the original circuit. The final mapping obtained from this forward traversal will be used as the initial mapping in the following reverse traversal. We use the same SWAP-based search with only the circuit reversed, and the original initial mapping will be updated to the final mapping in the reverse traversal.

Trade-off between the Circuit Depth and the Number of Gates.

When we insert SWAPs in the original quantum circuit, there is a trade-off between these two metrics: the number of gates and the circuit depth (an analogy in classical computation can be the trade-off between area and latency in digital circuit design). Figure 3.6 shows an example. Suppose there is a 9-qubit device and we have 2 CNOT gates on $\{q1, q2\}$, $\{q3, q4\}$ (marked by blue and green) to execute. The initial mapping is shown on the left side. We have two different solutions with different optimization objectives: **1) Depth First**. By inserting 4 non-overlap SWAPs on $\{q1, q5\}$, $\{q2, q9\}$ $\{q3, q7\}$, and $\{q4, q8\}$ (marked by 4 red arrows) which can be executed simultaneously, we can satisfy these 2 two-qubit gate dependencies with 4 additional SWAPs, and the circuit depth increases by 1 SWAP. **2) Number of Gates First**. $\{q2, q9\}$ is first swapped and then two qubit pairs $\{q2, q3\}$ and $\{q4, q8\}$ are swapped simultaneously. The SWAP on $\{q2, q3\}$ must be applied after the first SWAP on $\{q2, q9\}$ so that the circuit depth increases by 2 SWAPs, but only 3 additional SWAPs are required to resolve all the dependencies.

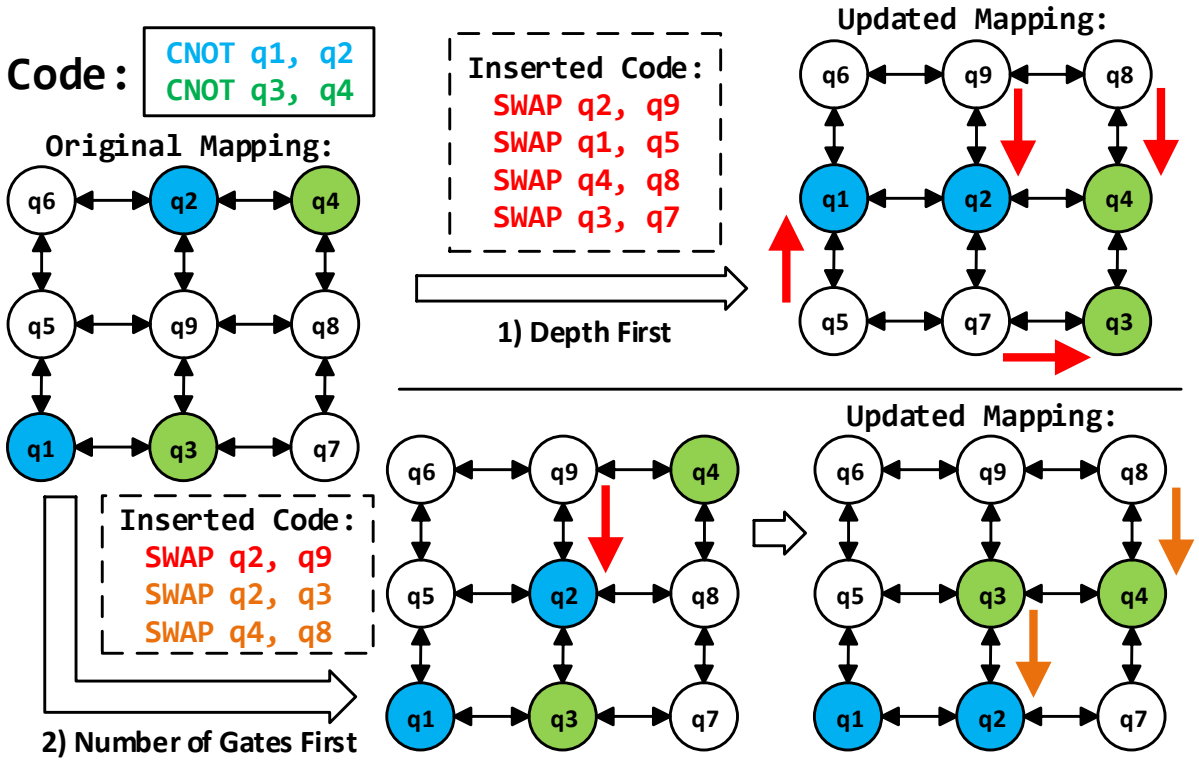


Figure 3.6: Example of Generated Circuits for Different Optimization Objectives (a Trade-off between d and g)

The two solutions above showed an example of a trade-off between d and g. To enable the control of such trade-off, a *decay* effect is introduced in SABRE which makes our heuristic search algorithm prone to selecting non-overlap SWAPs. For example, after the SWAP on q_2 and q_9, the heuristic cost function result for any SWAPs containing q_2 or q_9 will increase slightly to let our search algorithm favor choosing SWAPs containing other qubits.

In summary, these three design decisions bring exponential speedup for scalability, an high-quality initial mapping solution, and the controllability between different optimization objectives. These advantages ensure SABRE achieves all the design objectives discussed in Section 3.3.2.

3.4.4 Design the Heuristic Cost Function

As mentioned above, the objectives for heuristic cost function are summarized as follows:

1. H should be able to indicate the SWAP that can move the qubits in F closer to finally allow the physical execution of the two-qubit gates in F.

2. Besides the two-qubit gates in F, the heuristic cost function should be able to consider follow-up two-qubit gates for more effective qubit movement.

3. It should be able to control the parallelism of inserted SWAPs to enable the trade-off between gate count and circuit depth mentioned in Section 3.4.3.

Nearest Neighbor Cost (NNC) function is used to construct the basic heuristic function. Further optimization is introduced later to achieve all the design objectives.

Nearest Neighbor Cost Function.

NNC-based heuristic function has been widely used in previous research [71, 77, 69]. NNC is the minimal number of SWAPs required to move two logical qubits adjacent to each other on the quantum device. On ideal 1D/2D lattice hardware models, NNC can be easily obtained from the coordinates of the physical qubits while on NISQ devices with irregular coupling, NNC is the length of the shortest path between two physical qubits on the coupling graph, which has already been obtained in $D[\ ][\ ]$ during the preprocessing stage (an offset -1 is ignored without affecting the result). In our design, the summation of the distances between all qubit pairs in F is the basic heuristic cost function (shown in Equation 3.1). To evaluate the candidate SWAPs, the mapping π is temporarily changed by a SWAP and then H_{basic} is calculated. If H_{basic} is small, it generally means the distances between the two qubits in the qubit pairs from F are short,

and this SWAP is more likely to make the gates in F executable. The SWAP with the minimal H_{basic} will be selected.

$$H_{basic} = \sum_{gate \in F} D[\pi(gate.q_1)][\pi(gate.q_2)] \tag{3.1}$$

Look-Ahead Ability and Parallelism.

Although H_{basic} is able to guide the heuristic search and solve the qubit movement, it only considers the two-qubit gates in F. However, a local qubit movement can affect not only the gates in F but also the following gates. For the example in Figure 3.2, the SWAP between q_3 and q_7 is a good selection because it not only resolves the dependencies for the gates in the front layer but also makes the q_2 and q_7 closer in the following gate. Thus, we introduce the Extended Set E, which contains some closet successors of the gates from F in the DAG. The size of E is flexible, depending on how much look-ahead ability we hope to have. A large E is not necessary since the summation over E is only an inaccurate estimation of the effect of a SWAP and the amount of computation will also increase.

In the updated heuristic cost function, we sum over the gates in both E and F to enable the look-ahead ability. Since E and F are different sizes, we normalize the two summations by the sizes of F and E respectively. Also, the gates in F should have some priority since they need to be executed before those in E. So that a weight parameter $W, 0 \leq W < 1$, is added to reduce the effect of the second term.

In order to select SWAPs that can be executed in parallel, a *decay* effect is introduced in the heuristic cost function. If a qubit q_i is involved in a SWAP recently, then its *decay* parameter will increase by δ ($decay(q_i) = 1 + \delta$). This decay parameter will let our heuristic search tend to select non-overlap SWAPs and increase the parallelism in the generated circuit. Moreover, by tuning the value of δ, we are able to control the

'willingness' of our heuristic search to generate different circuits with a trade-off between the number of gates and circuit depth. The final version of our optimized heuristic function is shown in Equation 3.2. The complexity of this heuristic function is $O(N)$ since all qubits appear in F in the worst case. The size of E is not considered because it will not be very large and is set to N in our evaluation.

$$\begin{aligned} H = max(decay(SWAP.q_1), decay(SWAP.q_2)) \\ *\{\frac{1}{|F|}\sum_{gate \in F} D[\pi(gate.q_1)][\pi(gate.q_2)] \\ +W * \frac{1}{|E|}\sum_{gate \in E} D[\pi(gate.q_1)][\pi(gate.q_2)]\} \end{aligned} \tag{3.2}$$

3.5 Evaluation

In this section, we evaluate SABRE with a set of benchmarks on the latest, reported hardware model based on the superconducting circuit technology.

Benchmarks. The benchmarks are selected from previous work [56, 77], including quantum programs from IBM's QISKit [76], some functions from RevLib [91], and some algorithms compiled from Quipper [92] and ScaffCC [93].

Hardware Model. We use the coupling graph from IBM's Tokyo chip [37] (Figure 3.1) with 20 qubits. All the couplings are symmetric and the CNOT gate is allowed in both directions between each pair of connected physical qubits.

Experiment Platform. All experiments in this chapter are executed on a server with 2 Intel Xeon E5-2680 CPUs (48 logical cores) and 378GB memory. The Operating System is CentOS 7.5 with Linux kernel version of 3.10.

Algorithm Configuration. The size of the Extended Set $|E|$ is fixed to be 20 and the the weight W to be 0.5. The *decay* parameter δ increases from 0.001 and this *decay* function is reset every 5 search steps or after a CNOT gate is executed. The algorithm

is executed for 5 times, each with a different initial mapping for each benchmark. Each time we run 3 traversals (forward-backward-forward) and report the best result out of 5 attempts.

Comparison. There are several existing algorithms with the flexibility to be applied to an arbitrary coupling graph proposed by IBM [76], Siraichi *et al.* [56], and Zulehner *et al.* [77]. Among them, Zulehner *et al.*'s algorithm has beaten the other two solutions and is used as the Best Known Algorithm (BKA) in this chapter. For a fair comparison, their source code [94] is downloaded and only the embedded hardware model is modified to be the same IBM 20-qubit chip model. It is then recompiled with full optimization, and executed on the same server with SABRE.

3.5.1 Number of Gates Reduction

Table 3.2 shows the gate counts reduction of SABRE compared with BKA [77]. SABRE could beat BKA on various benchmarks of different sizes.

Small Size Cases and Ising Model.

SABRE could perform much better than BKA on small-size benchmarks. It is able to find a good initial qubit mapping with no or very few additional SWAPs required. The number of additional gates could be significantly reduced by 91% or even fully eliminated. For ising model benchmarks, the optimal solution is trivial since the ising model in quantum mechanics only considers nearby coupling energy. Although the number of qubits and the number of gates are much larger compared with small cases, SABRE can still find the optimal solution. BKA only considers the two-qubit gates at the beginning of the circuit without such a scheme to improve the initial mapping.

Table 3.2: Number of Additional Gates and Runtime Compared with BKA [77]

Original Circuit				BKA [77] (C++)			SABRE (Python)				Comparison		
type	name	n	g_{ori}	g_{add}	g_{tot}	t_{tot}	g_{la}	g_{op}	t_1	t_{op}	t_{tot}/t_{op}	Δg	$\Delta g/g_{add}$
small	4mod5-v1_22	5	21	15	36	0	6	0	0	0	N/A	15	100%
small	mod5mils_65	5	35	18	53	0	12	0	0	0	N/A	18	100%
small	alu-v0_27	5	36	33	69	0	30	3	0	0	N/A	30	91%
small	decod24-v2_43	4	52	27	79	0	9	0	0	0	N/A	27	100%
small	4gt13_92	5	66	42	108	0	18	0	0	0	N/A	42	100%
sim	ising_model_10	10	480	18	498	1.37	39	0	0.003	0.004	342.5	18	100%
sim	ising_model_13	13	633	60	693	42.46	66	0	0.005	0.007	6066	60	100%
sim	ising_model_16	16	786	Out of Memory			84	0	0.008	0.01	N/A	N/A	N/A
qft	qft_10	10	200	66	266	0.22	93	54	0.004	0.103	2.136	12	18%
qft	qft_13	13	403	177	580	266.27	204	93	0.015	0.036	7396	84	47%
qft	qft_16	16	512	267	779	474.81	276	186	0.028	0.084	5652	81	30%
qft	qft_20	20	970	Out of Memory			429	372	0.034	0.102	N/A	N/A	N/A
large	rd84_142	15	343	138	481	1.97	243	105	0.012	0.035	56.29	33	24%
large	adr4_197	13	3439	1722	5161	4.53	2112	1614	0.19	0.49	9.245	108	6%
large	radd_250	13	3213	1434	4647	2.23	1488	1275	0.16	0.48	4.646	159	11%
large	z4_268	11	3073	1383	4456	1.15	1695	1365	0.15	0.44	2.614	18	1%
large	sym6_145	14	3888	1806	5694	0.56	1650	1272	0.19	0.56	1.000	534	30%
large	misex1_241	15	4813	2097	6910	0.3	2904	1521	0.29	0.89	0.337	576	27%
large	rd73_252	10	5321	2160	7481	1.19	2391	2133	0.31	0.94	1.266	27	1%
large	cycle10_2_110	12	6050	2802	8852	1.31	2622	2622	0.44	1.35	0.970	180	6%
large	square_root_7	15	7630	3132	10762	2.81	5049	2598	0.63	1.5	1.873	534	17%
large	sqn_258	10	10223	4737	14960	16.92	5934	4344	1.23	3.52	4.807	393	8%
large	rd84_253	12	13658	6483	20141	15.25	7668	6147	1.82	5.39	2.829	336	5%
large	co14_215	15	17936	9183	27119	18.37	10128	8982	3.18	9.51	1.932	201	2%
large	sym9_193	10	34881	17496	52377	72.61	26355	16653	11.11	30.17	2.407	843	5%
large	9symml_195	11	34881	17496	52377	81.73	25368	17268	11.1	31.42	2.601	228	1%

small: small quantum arithmetic. **sim:** quantum simulation. **qft:** quantum fourier transform. **large:** large quantum arithmetic. n**:** number of logical qubits in the original circuit. g_{ori}**:** original number of gates. g_{add}**:** number of additional gates. g_{tot}**:** total number of gates. t_{tot}**:** total runtime in seconds, '0' means shorter than 0.001 second. g_{la}**:** number of additional gates with only look-ahead heuristic. g_{op}**:** number of additional gates after reversal traverse. t_1**:** runtime of first traverse in seconds. t_{op}**:** runtime of all 3 traversals. Δg**:** $= g_{add} - g_{op}$. **Out of Memory:** the program required more than 378 GB memory (entire memory space on the test server)

Large Size Cases.

For larger circuits in type 'large' and 'qft', SABRE can still be better than BKA. Since the BKA searches a much larger space in each step, SABRE may not achieve the same or better result in the first traversal. The g_{la} column in Table 3.2 shows the number of additional gates after the first traversal with look-ahead heuristic function and g_{la} is larger than g_{add} in most cases. However, with the help of our reverse traversal technique, SABRE (shown in g_{re}) is able to outperform BKA with the updated initial mapping and reduce the number of additional gates by 10% on average.

Note that the gate count reduction for large size cases is less significant than that for small size cases. This difference comes from whether a perfect initial mapping, which could satisfy all the CNOT gate constraints in the program after the inital mapping and does not require further SWAPs, can be found. For small benchmarks, there often exists a physical qubit coupling subgraph that can perfectly or almost match logical qubit coupling in the benchmarks. Our algorithm can find such matching (at least for all small benchmarks we have tested), while BKA cannot. This leads to substantial benefit since very few or no SWAPs are inserted. For the benchmarks with larger number of gates, a physical qubit subgraph that can match the logical qubits coupling usually does not exist. Therefore, both our approach and the baseline need to insert more SWAPs, leading to less benefit.

3.5.2 Runtime Speedup and Scalability

As discussed in Section 3.4.3, the size of search space is $O(exp(N))$ in BKA, which limits its scalability in terms of the number of qubits. But the search space size in SABRE is only $O(N)$. Although more search steps are required since only one SWAP is selected in each step, the overall complexity in the worst case is still $O(N^{2.5}g)$. Such a difference in complexity makes BKA not applicable to larger size cases.

Runtime Comparison.

BKA is written in C++ and compiled with GCC O3 optimization, while SABRE is implemented in pure Python without any parallelization or C/C++ accelerated library. The 't_{tot}/t_{op}' column in Table 3.2 shows the ratio between the execution time of BKA and SABRE. For most benchmarks, SABRE requires significantly less execution time. Even in the worst case 'misex1_241', SABRE only needs about 3 times runtime compared with

the BKA. Since the intrinsic speed difference between C++ and Python can be over 100 times, the speedup can still be estimated to be dozens of times if the same programming language is used.

Limit of BKA and Scalability.

Our experiments have reached the limit of BKA (shown in Table 3.2 with 'Out of Memory'). For the 'sim' and 'qft' type, the benchmarks share the same function with different input sizes. The runtime of BKA grows rapidly as the number of qubits increases. For 'qft_16' benchmark, we observe that BKA requires more than **40GB** memory and **474.81** seconds runtime while SABRE only required about **200MB** memory and **0.08** seconds runtime. For 'ising_model_16' and 'qft_20' benchmarks, the BKA requires more than **378GB** memory and can not be executed on our server. But SABRE can still solve it in **0.1** seconds with about **300MB** memory. These results show that SABRE is much more scalable than BKA.

3.5.3 Trade-off between Number of Gates and Depth

The *decay* effect is introduced in the heuristic cost function in order to reduce the depth of the generated circuit. Figure 3.7 shows the generated circuit variation with different δ values for 9 benchmarks. The X-axis is the number of gates normalized to g_{ori} (in Table 3.2). The Y-axis represents the generated circuit depth normalized to the original circuit depth. These results showed that SABRE could provide about 8% variation in generated circuit depth by varying the number of gates and control the generated circuit quality. For a specific implementation technology, we can change the δ according to the qubit coherence time and gate fidelity data. However, if we continue to increase δ, both the circuit depth and the number of gates may increase (not shown

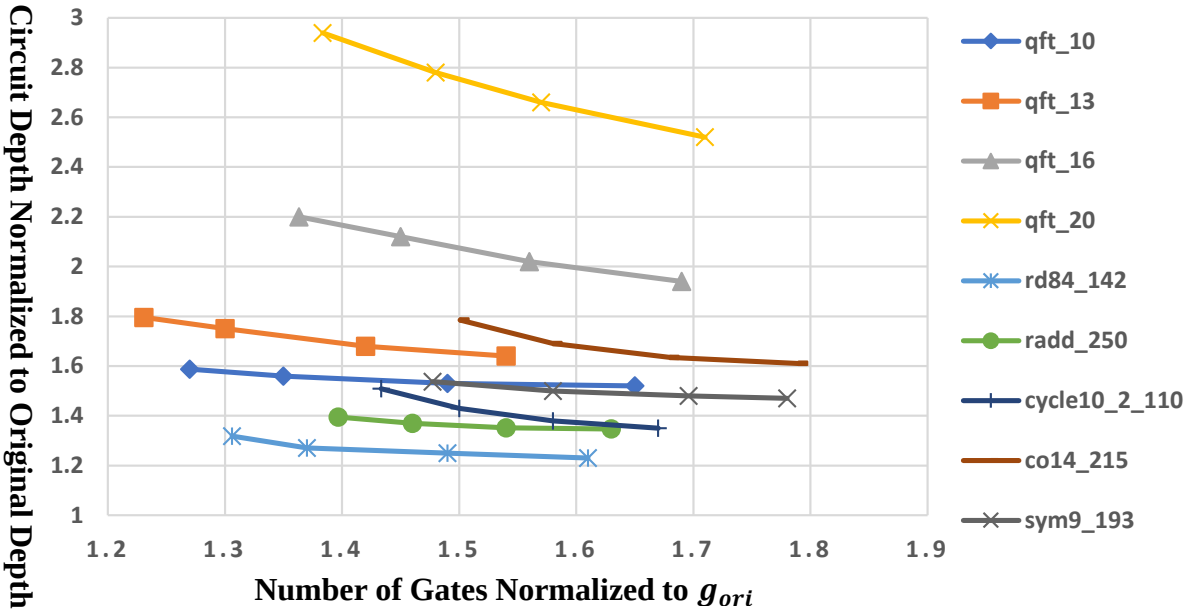


Figure 3.7: Trade-off between g and d in the Output Circuits

in the figure) because our search algorithm will consider more about unmoved qubits instead of trying to satisfy a CNOT dependency, which will bring redundant SWAPs.

3.6 Limitation and Future Work

This chapter provides an effective, flexible, and scalable solution for the qubit mapping problem. However, some of our assumptions may not hold due to the rapid development in this area. Some limitations and potential future research directions are listed as follows:

Benchmarks. We select 26 benchmarks of different sizes and functions from several benchmark suites. However, these quantum circuits may not be able to fully represent the characteristics of emerging practical NISQ applications which are still under development.

Various Chip Architecture. We use the hardware model from the latest IBM's 20-qubit chip, on which each connected qubit pair support CNOT gates. However, the chip model varies among different vendors. For example, Rigetti's QPU supports CPhase

and iSWAP two-qubit gates [52, 95]. How to design more general circuit transformations is beyond the scope of this chapter, but can be a future research direction.

More Precise Hardware Modeling. Besides the qubit coherence time, gate fidelity, and available on-chip coupling, the difference in the error rate of various quantum gates and of the same quantum gate applied on different qubits or qubit pairs may also influence the fidelity of executing a quantum algorithm [85]. In addition, realistic hardware suffers from more imperfections which are not covered in this chapter, such as the cross talk between qubits. Both facts call for a more precise hardware model to enable better platform-specific quantum circuit optimization.

3.7 Related Work

Although the qubit mapping problem shares some similarities with the register allocation [96, 97] and instruction scheduling problem [98, 99, 100] in classical computing, the constraints are different. In register allocation, the main constraint is the limited number of registers while for quantum computing, the number of physical qubits cannot be smaller than that of logical qubits. In instruction scheduling, the main constraints are the data dependency and limited number of computing units. But in the qubit mapping problem, the major constraint is the limited coupling between physical qubits. Therefore, existing methods for such problems cannot be directly applied in this qubit mapping problem.

It is well known that nearest neighbor coupling is the most feasible and promising when there were only devices with a very limited number of qubits. Attempts to solve qubit mapping problem at that time were made on hypothetical quantum hardware models like ideal 1D/2D lattice models and can be classified into two types. One popular type of approach is to formulate the qubit mapping problem into a mathematically equiva-

lent optimization problem and then apply a software solver [57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67]. The major drawback of this type of approach is that a general solver cannot utilize the intrinsic feature in qubit mapping and the execution time is usually very long compared with the following heuristic approaches. Another type of approach is search algorithms guided by heuristic cost functions. Several attempts have been made on ideal 1D/2D lattice qubit coupling models [68, 69, 70, 71, 72, 73, 74, 75], but they are not applicable in the NISQ era since qubit coupling can be much more complex and restricted on NISQ devices. Some other works target hypothetical large-scale quantum computers [86, 88], which is beyond the scope of NISQ and the qubit mapping problem turns out to be another one [86, 87, 88, 89].

After IBM launched its quantum cloud service, more people were able to work on hardware models from realistic devices. IBM provides a mapper targeting IBM's chips in its quantum computing toolkit QISKit [76]. This mapper divides the quantum circuit into independent layers. Each layer only contains non-overlapped operations. Then it randomly searches satisfying mappings for each layer guided by certain heuristics [56, 77]. Besides IBM's solution, two more recent works [56, 77] are proposed for IBM's chips and can handle devices with arbitrary coupling, which are discussed as follows.

Siraichi *et al.* studied the qubit allocation problem on IBM QX2 and QX3 chips [56]. They proposed a search algorithm to find the optimal solution based on dynamic programming. However, this optimal algorithm requires exponential time and space to execute and can only work for circuits with 8 or fewer qubits. For larger size cases, they proposed a heuristic method for both initial mapping and intermediate qubit movement. Their initial mapping solution counted the number of two-qubit gates between each pair of logical qubits and tried to find a matched edge on the physical chip with no temporal information considered in this stage. For the qubit movement, they only resolved one two-qubit gate each time and determined whether to move qubits depending on the

number of two-qubit gates between them greedily without considering the effects of these local decisions. Their heuristic method is fast but oversimplified with results worse than IBM's solution.

Zulehner *et al.* tried to use A* search plus heuristic cost function [77] (the BKA in this chapter). They divided the two-qubit gates into independent layers similar to IBM's solution. Then they searched all possible combination of SWAP gates to minimize the sum of distance between the coupled qubits in the layer and reduce the depth of the final output circuit at the same time. Although their method is more efficient than IBM's approach and only requires up to several minutes on 16-qubit circuits, searching all possible combinations of concurrent SWAP gates still requires exponential time. Their initial mapping was determined by only those two-qubit gates at the beginning of the circuit without global consideration.

3.8 Conclusion

The NISQ era is coming in the next few years while a significant gap remains between quantum software and imperfect NISQ hardware. In this chapter, we try to solve the qubit mapping problem caused by limited physical qubits coupling on NISQ devices. Two-qubit gate is allowed between arbitrary two logical qubits but can only be implemented between two nearby physical qubits on NISQ hardware. The initial mapping between logical qubits and physical qubits and its evolution need to be carefully designed to minimize the circuit transformation overhead. We propose SABRE, a novel SWAP-based bidirectional heuristic search method to overcome the drawbacks of previous works and ensure flexibility, scalability, controllability, and high-quality initial mapping. Experiment results show that SABRE can generate hardware-compliant circuit among different objectives with less or comparable overhead consuming much shorter execution

time. Although SABRE works for IBM chips with arbitrary symmetric CNOT coupling, the hardware model, which differs among vendors and may change over time, is also simplified and single-qubit gates are not yet considered. We only add additional gates instead of modifying the original circuit, while the latter one is much more complicated. In conclusion, this work explores one step in mitigating the quantum software-hardware gap. Future work is required to take more precise hardware models into consideration.

Chapter 4

Towards Efficient Superconducting Quantum Processor Architecture Design

4.1 Introduction

The superconducting quantum circuit [9] has become one of the most promising technique candidates for building quantum computing systems [101, 102, 84] due to the ever-increasing qubit coherence time, individual qubit addressability, fabrication technology scalability, etc. Towards efficient superconducting quantum circuit based quantum computing systems, significant research has recently been conducted, ranging from compiler optimization [103, 104] to periphery control hardware support [28, 105] and device innovation [78, 106].

Despite these system optimizations, the performance of a superconducting quantum processor is still highly limited by the amount of computation resource on it. Researchers have been trying to integrate more qubits and qubit connections on one superconducting

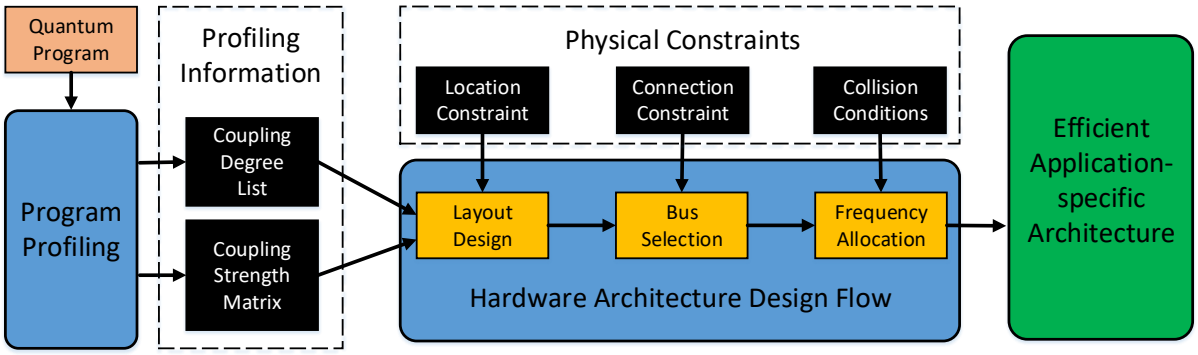


Figure 4.1: Overview of the Proposed Architecture Design Flow

quantum processor substrate. For example, IBM's first superconducting quantum chip on the cloud has 5 qubits with 6 qubit connections, while its latest published chip has 20 qubits with 37 qubit connections [107]. Increasing the number of physical qubits on a superconducting quantum processor allows programs with more logical qubits to be executed. Denser qubit connections can increase the overall chip performance by reducing the overhead of qubit mapping and routing [56, 77, 18, 108].

Nevertheless, more qubits and qubit connections will, unfortunately, increase the probability of defect occurrence on a chip, leading to lower yield rate and blocking future development of larger-scale superconducting quantum processor. For example, the yield rate of a 17-qubit chip can be lower than 1% under IBM's state-of-the-art technology [109]. Such a low yield rate comes from *frequency collision*, a unique defect on superconducting quantum processors [110, 111]. The frequencies of physically connected qubits may 'collide' with each other when their values satisfy some specific conditions. More qubit connections naturally increase the probability of frequency collision and lower the yield rate.

To optimize both the yield rate and performance would be desirable, but it is difficult in general due to the inherent trade-off between these two objectives. Most previous efforts on them are direct device-level improvement [78, 106, 82, 112], while little atten-

tion has been given to the architectural design of a superconducting quantum processor. This chapter fills the gap by exploring the possibility of efficient *application-specific architecture design* to reach an optimized balance between yield rate and performance. We vision that an array of quantum computing accelerators, each of which is tailored to a specific application, is much more likely to be adopted in the near term where computation resources are still limited before we can reach a universal quantum computer (i.e., one quantum computer that runs all kinds of quantum programs). Our design shares the same high-level spirit with the hardware architecture designs in classical computing (e.g., machine learning [113, 114], graph processing [115, 116]), but faces different scenarios because both the program patterns and the hardware design space are different in quantum computing.

In particular, we highlight two key challenges to be addressed before the application-specific principle can be applied in superconducting quantum processor design. **First**, we need to identify and abstract the computation pattern of quantum programs that can guide the hardware architecture design. Prior quantum program analysis studies [117, 118, 119, 120, 121, 122] mainly focused on software or compiler optimization and cannot extract appropriate information for hardware architecture optimization. **Second**, the abstracted computation pattern must give guidance to efficient architectural designs, which employ fewer computation resources with physical constraints satisfied to achieve both high yield rate and performance. Existing superconducting quantum processor design schemes cannot handle such irregular/complicated application-specific architecture design tasks [123, 124, 109, 125].

To overcome these two challenges, we design a systematic design flow to automatically generate efficient superconducting quantum processor architecture designs for different quantum programs (shown in Figure 4.1). We first identify two key computation patterns in quantum programs, ***coupling degree list*** and ***coupling strength matrix***. A

profiler is built to automatically extract them from an input quantum program. Both of them are critical to the program performance and hardware yield rate, and thus optimizing their underlying architecture support can potentially achieve a better balance between the performance and yield rate. We then propose an architecture design flow, which comes with three key subroutines, ***layout design***, ***bus selection***, and ***frequency allocation***. Each subroutine focuses on different hardware resources and must cooperate with corresponding profiling results and physical constraints. We further propose an array of heuristics to ensure the scalability and effectiveness of the architecture search process. Empirical studies show that these heuristics can find 'near-optimal' solution in the reduced search space.

In summary, this chapter makes the following contributions:

- We are the first to identify the optimization opportunity from the architecture level to push forward the balance between performance and hardware yield rate for superconducting quantum processors.
- We formalize an end-to-end design flow, equipped with a set of novel algorithmic primitives, to automatically generate a series of application-specific architectural designs under different hardware resource limits.
- Comprehensive experiments show that our design flow could outperform IBM's general-purpose designs with better Pareto-optimal results, e.g., magnitudes of yield improvement with negligible performance loss.

4.2 Background

In this section, we will introduce more technical details about the superconducting quantum circuit devices to help understand the follow-up superconducting quantum

processor architecture design flow. Note that in this chapter we assume that the input quantum circuit has been decomposed and gates with three or more qubits are not considered.

4.2.1 Superconducting Quantum Circuit Basics

All the qubits and quantum operations in a quantum circuit must be implemented in a real physical quantum computing system to execute the program. In this chapter, we focus on superconducting quantum processors with fixed-frequency Josephson-junction-based transmon qubits [78] and all-microwave cross-resonance two-qubit gates [126] that are adopted by IBM [109].

Physical Qubit and Frequency Figure 4.2 shows the physical circuit and energy levels of a transmon qubit [78]. Due to the nonlinearity of the Josephson junction, the gaps between the energy levels in this quantum anharmonic oscillator are different, which allows us to use the ground state $|0\rangle$ and the first-excited state $|1\rangle$ as the computation basis without populating other states. Suppose the energy gap between $|0\rangle$ and $|1\rangle$ for a qubit is E_{01}. The *frequency* of this qubit f_{01} is defined as $f_{01} = E_{01}/h$, where h is the Planck constant. Similarly, we use f_{12} to represent the energy gap between $|1\rangle$ and $|2\rangle$. For a typical qubit design with effective operations [110], f_{01} and f_{12} are about $5GHz$ and $4.66GHz$, respectively. The anharmonicity of this qubit is defined to be $\delta = f_{12} - f_{01}$, which is $-340MHz$ under this typical design [125, 127].

Qubit Layout The superconducting physical qubits are confined on a 2-dimensional planar substrate. Although the qubit placement can be flexible, major vendors fabricate the qubits in a regularized structure to ensure scalability and reduce the fabrication complexity. For example, IBM's 16-qubit and 20-qubit chips [37] placed their qubits on the nodes of 2×8 and 4×5 lattices, respectively. Google's 72-qubit chip placed its qubits

on some nodes of an 11×12 lattice [51].

Qubit Connection To enable two-qubit gates between two physical qubits, resonators, also known as qubit buses, are employed to connect nearby qubits [126]. For examples, Figure 4.2 shows two types of commonly used buses. The first one is a 2-qubit bus connecting two physical qubits. The second one is a 4-qubit bus, which connects four physical qubits in a square together. The coupling graphs of these two types of buses are shown on the right. Compared with a 2-qubit bus, 4-qubit bus support two-qubit gates on not only the four qubit pairs on the edges but also two qubit pairs on the diagonals.

Qubit Mapping It is usually assumed that a two-qubit gate can be applied on arbitrary two logical qubits in a quantum program but some two-qubit gates may not be executable due to the limited qubit connection on a superconducting quantum processor. On the hardware side, this problem can be relieved by employing more physical qubit connections so that two-qubit gates can be directly supported on more qubit pairs. On the software side, a qubit-remapping compiler [57] can resolve the dependency of the remaining unexecutable two-qubit gates while additional operations must be introduced with longer execution time and higher error rate. Therefore, more physical qubit connections can help with the overall performance by allowing native two-qubit gates on more physical qubit pairs.

Fabrication Variation Variation is inevitable when fabricating a superconducting quantum processor. If a qubit is designed to have frequency f, the actual frequency after fabrication will be $f' = f + n_f$, where n_f satisfies Gaussian distribution $N(0, \sigma)$. σ is the fabrication precision parameter, which is around $130MHz \sim 150MHz$ under IBM's state-of-the-art technology [109]. Such noise makes it hard to predict the post-fabrication frequency precisely, which brings the probability of frequency collision.

Frequency Collision When two or three qubits are connected, *frequency collision* may happen and cause defects on the device. Figure 4.3 summaries seven qubit frequency

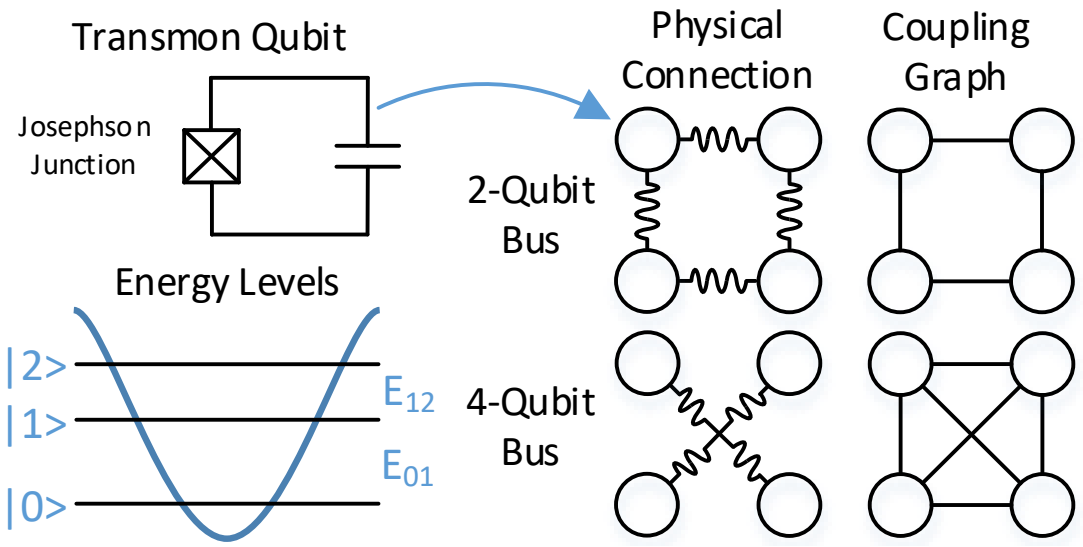


Figure 4.2: Superconducting Qubit and Connection

collision conditions in IBM's devices [109, 111]. On the left is a table showing the conditions and thresholds of different collision situations. Condition 1, 2, 3, and 4 involve two connected qubits (j and k). Condition 5, 6, and 7 involve three qubits of which two qubits (k and i) both connect to the other qubit j. The approximate equations and the corresponding thresholds determine whether one frequency collision happens. For example, if qubit j and k are connected and $|f_j - f_k| < 17MHz$, then the first condition is satisfied and frequency collision occur. Note that the fourth condition has no threshold because it is an inequality rather than an approximate equation. On the right is a graphical illustration, showing the geometric locations of the qubits that may have frequency collisions of different conditions in two subfigures. Each circle represents one qubit and the gray square represent a 4-qubit bus connecting the four surrounding qubits.

	Conditions	Thresholds
1	$f_j \cong f_k$	$\pm 17MHz$
2	$f_j \cong f_k - \delta/2$	$\pm 4MHz$
3	$f_j \cong f_k - \delta$	$\pm 25MHz$
4	$f_j > f_k - \delta$	
5	$f_i \cong f_k$	$\pm 17MHz$
6	$f_i \cong f_k - \delta$	$\pm 25MHz$
7	$2f_j + \delta \cong f_k + f_i$	$\pm 17MHz$

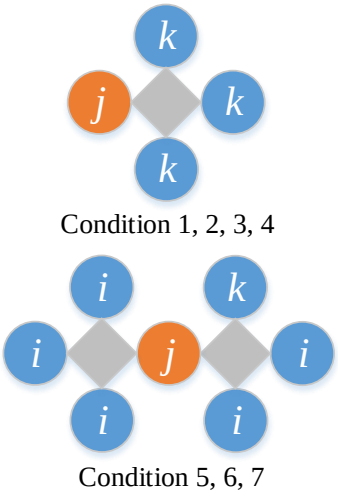


Figure 4.3: Frequency Collision Conditions [109, 111]

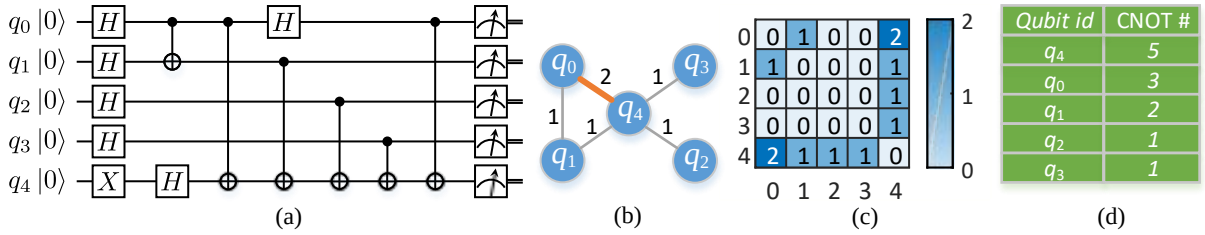


Figure 4.4: Example of the Profiling Method

4.3 Quantum Program Profiling

The first step towards the development of an architecture-specific quantum processor for both high performance and yield rate is to determine what program information we should focus on. There are several different types of components in a quantum circuit but not all of them will significantly affect the hardware design. Our target program component(s) should satisfy two conditions: 1) the component's execution is a performance bottleneck which can be dramatically improved with optimized hardware support, and 2) the component's required hardware should significantly affect the yield rate.

We found that two-qubit gates can be a key factor to bridge performance and yield.

To execute two-qubit gates on a quantum processor with limited qubit-to-qubit coupling, a large number of additional operations are introduced to satisfy their dependencies. But implementing two-qubit gates on two physical qubits require on-chip qubit connections which can lower the yield rate through increasing the probability of frequency collision. Therefore, we give logical qubits and qubit pairs priorities based on the number of involving two-qubit gates to help with the following architecture design. Critical qubits and qubit pairs will have more hardware support to improve the efficiency of the generated architectures.

These remaining components, single-qubit gates, initialization, and measurement operations, do not involve qubit-to-qubit interactions and all happen locally on individual qubits when they are implemented on hardware. As a result, hardware support for these components will not affect the chip yield through frequency collision.

4.3.1 Profiling Method

As discussed above, our profiling will focus on the logical qubits and the two-qubit gates. Figure 4.4 shows an example to illustrate the profiling procedure. Suppose we have a quantum circuit as shown in Figure 4.4 (a). It has 5 logical qubits denoted by $q_{0,1,2,3,4}$. All of them are initialized to be $|0\rangle$. Then some single-qubit gates and two-qubit gates are applied. Measurement operations are at the end.

We first ignore all single-qubit gates, initialization, and measurement operations. Then we create a logical coupling graph, in which each vertex represents one logical qubit in the circuit. Two vertices are connected by an undirected edge if there exists two-qubit gates applied on the two corresponding logical qubits. The weight of an edge is the number of two-qubit gate instances on the two connected vertices. In this example, Figure 4.4 (b) shows the generated graph for the example circuit. The weight of the edge

between vertex q_0 and vertex q_4 is 2 since there are two two-qubit gates on q_0 and q_4. For all other edges, the weight is 1 because there is only one two-qubit gate on each of those qubit pairs. The first profiling result is the weighted adjacency matrix of the logical coupling graph, namely the *coupling strength matrix*. The element with indices (i, j) represents the number of two-qubit gates between q_i and q_j. Figure 4.4 (c) shows the *coupling strength matrix* for the example circuit. Note that *coupling strength matrix* is always a symmetric matrix.

The second result is *coupling degree list*. For each qubit, we sum the weights of edges that connect to its corresponding vertex and define the number of two-qubit gates applied on it as the *coupling degree* of one qubit. If one qubit is associated with more two-qubit gates in a quantum circuit than other qubits, this qubit will use the physical qubit connections more frequently when executing on the chip. Naturally, we should pay more attention to those qubits with larger coupling degree. Therefore, all qubits are placed in a sorted list, namely the *coupling degree list*. Figure 4.4 (d) is the *coupling degree list* in this example. The first one in this list is q_4 because it has the largest coupling degree. All qubits are in a descending order.

4.3.2 Gate Pattern Examples

In this section, we show the existence of distinct two-qubit gate patterns and discuss the opportunity for application-specific architecture design with two examples. Figure 4.5 shows their *coupling strength matrices*. On the left is an 8-qubit UCCSD ansatz for VQE, a quantum simulation algorithm [48]. The high coupling strength qubit pairs form a chain structure marked by a red rectangle. Q_0 and Q_1 have a large number of two-qubit gates between them, as well as $\{Q_1Q_2, Q_2Q_3, \cdots, Q_6Q_7\}$. For other qubit pairs, the coupling strength is much lower (only about 10%). On the right is a 15-qubit quantum arithmetic

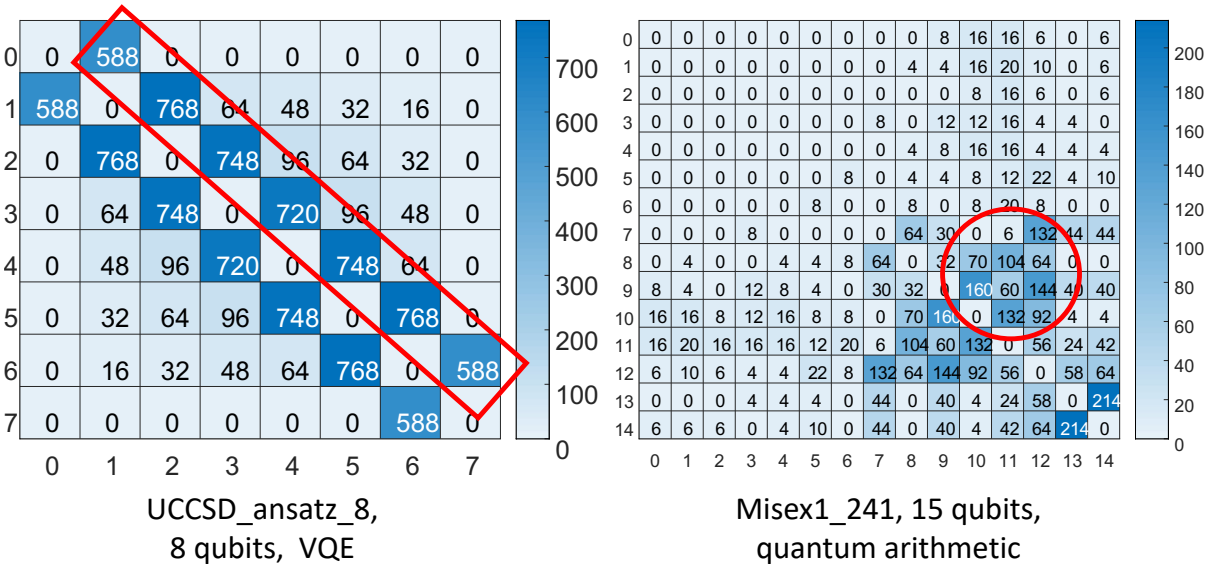


Figure 4.5: Qubit Coupling Strength Pattern Examples

function [91]. The coupling strength among $Q_0Q_1 \cdots Q_5$ are 0 since there are no two-qubit gates on any two of them. However, there is a large number of two-qubit gates where one qubit is in the set $Q_{7,8,9,10}$ and the other qubit is in the set $Q_{10,11,12}$ (marked by a red circle). The analysis of these two motivating examples provides us two observations:

1. The numbers of two-qubit gates on different logical qubit pairs can vary dramatically in a real quantum program.

2. Different types of quantum programs can have different two-qubit gate patterns.

These observations suggest that quantum processors can be customized for different programs with different patterns. An efficient architecture can focus on supporting the high-density coupling in a quantum program to reduce the number of connections on-chip. For example, a quantum processor with an 8-qubit chain structure (8 qubits and 7 qubit connections) can immediately support most of the two-qubit gates in the 8-qubit UCCSD ansatz program. The rest two-qubit gates can be supported through remapping without introducing too many additional operations because the total number

of the remaining two-qubit gates is relatively small. Such application-specific quantum computing accelerators with simplified architectures can be a more realistic goal in the near term than a general-purpose quantum processor with a large number of hardware resources.

4.4 Architecture Design

After a quantum circuit is profiled, a straightforward quantum processor architecture for such a circuit is to organize the on-chip qubits and qubit connections directly based on the logical coupling graph. However, we must consider the physical constraints for a practical architecture. For example, a logical coupling graph may not be perfectly fabricated on hardware since the allowed connections among superconducting qubits are very limited. Moreover, we hope to improve the yield rate by delivering architecture designs with fewer hardware resources. Therefore, the proposed hardware design flow must not only invest more hardware resource on frequent operations based on the profiling results, but must also obey the physical constraints on the hardware components arrangement.

To accomplish such a complicated task in a scalable way, we decouple the hardware design procedure into three subroutines and each subroutine focuses on different architecture components, i.e., qubit layout, connection, and frequency. For each subroutine, we first review the difficulty and the physical constraints considered. Then we discuss the design objectives, and how they are achieved in the proposed design algorithms.

4.4.1 Layout Design

The first step is to determine where to place the qubits. To ensure scalability and modularity, we follow the convention from major vendors introduced in Section 4.2 and will only place qubits on the nodes of a 2D lattice. We start from a large 2D lattice, in

which each node is initialized to be empty (Figure 4.6 (a)). Then physical qubits can be placed in the empty nodes and one node can contain at most one qubit.

There are many ways to place a given number of qubits on a 2D lattice. For example, 16 qubits can constitute a 4×4 lattice, a 2×8 lattice, or other more irregular structures. But we need to select one qubit layout that is most suitable for executing the program, i.e., most operations can be directly supported or indirectly supported with low overhead. The objectives of this qubit layout design subroutine are summarized as follows.

- Since we need to consider the profiling information, we create a pseudo mapping between logical qubits in the profiled program and the physical qubits in hardware architecture to be delivered. For two logical qubits with a large number of two-qubit gates between them, we hope to place their corresponding physical qubits in adjacent nodes so that later those two-qubit gates can be directly supported by the connection between the two physical qubits.

- One physical qubit can only have a limited number of directly connected qubits. For those two-qubit gates that cannot be directly supported, we hope to reduce the amount of additional operations introduce for remapping the qubits.

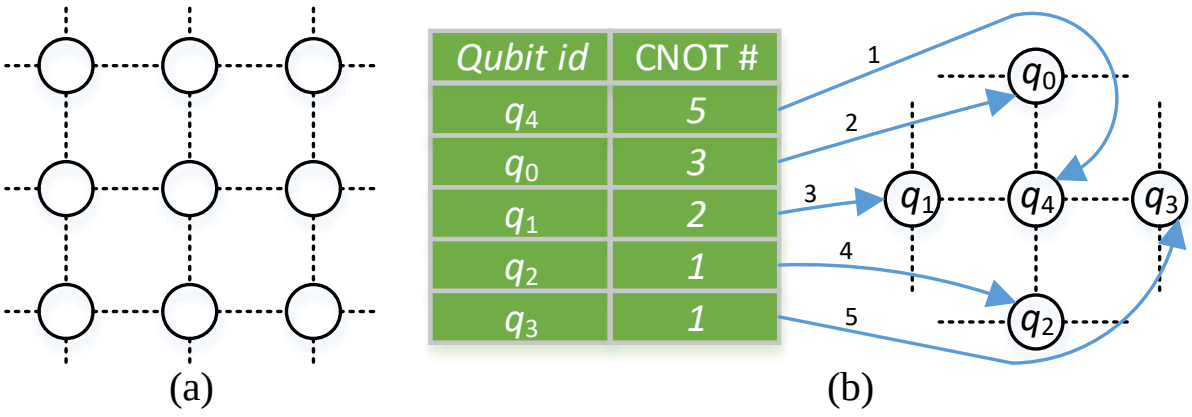


Figure 4.6: (a) Empty Lattice, (b) Qubit Placement Example

Algorithm 2: Qubit Placement on 2D Lattice

Input: *coupling degree list* L, *coupling strength matrix* M
Output: Geometric coordinates of placed qubits

Place the qubit with the largest coupling degree in L at one node with coordinate $(0, 0)$;
R = all the qubits remaining; `// R is the set of qubits that has not been placed yet.`
while R *is not empty* **do**
  `/* Find the next qubit to place */`
  $qubit\_candidate\_list = \varnothing$;
  for $\boldsymbol{q}$ *in* R **do**
    if $\boldsymbol{q}$ *is connected to any placed qubits* **then**
      $qubit\_candidate\_list.append(\boldsymbol{q})$;
    end
  end
  Find the qubit $\boldsymbol{q}$ with the largest coupling degree in $qubit\_candidate\_list$;
  $node\_cost = [\ ]$;
  `/* Determine the placement location */`
  for ***location*** *of the nodes that are empty and connected to at least one occupied node* **do**
    `/* Heuristic Cost function */`
    $node\_cost[\boldsymbol{location}] = \sum_{q' \in \boldsymbol{q}.neighbors} M[\boldsymbol{q}, q'] * distance[\boldsymbol{location}, q'.node]$
  end
  `/*` q' `must be placed neighbor qubits */`
  Place $\boldsymbol{q}$ in the ***location*** with the minimal score;
  $R.remove(\boldsymbol{q})$;
end

We propose a *coupling-based* qubit placement algorithm to determine the geometric locations of the qubits on a 2D lattice (pseudocode shown in Algorithm 2). We illustrate the algorithm with an example in Figure 4.6. First, we put the first qubit in the *coupling degree list*, q_4, on one node of the 2D lattice. Since the initial 2D lattice is empty, the location of q_4 does not matter. We set the geometric coordinate of the first qubit to be $(0, 0)$ and then place the rest qubits around q_4. q_4 has four neighbors, $q_{\{0,1,2,3\}}$, in the logical coupling graph. We need to select the next one to place. By checking the *coupling degree list*, we can see that q_0 is the one with the largest coupling degree. The node

occupied by q_4 has four equivalent adjacent nodes and we can place q_0 on any of them. In this example, we select the node on the north of q_4 with coordinate $(0, 1)$. Such an algorithm design ensures that the strongly coupled qubit pairs are given higher priority and placed on adjacent nodes, accomplishing the first objective mentioned above.

Then we need to place q_1 since its coupling degree is larger than that of q_2 and q_3. q_1 is connected to both q_4 and q_0 so that we need a more sophisticated way to evaluate all potential nodes for q_1. We use the function in line 13 of Algorithm 2 to find the node that can make q_1 close to its strong coupled neighbors in the logical coupling graph. This function is the summation over all q_1's placed neighbors. Each term in the summation is the product of the coupling strength between q_1 and one logical coupling neighbor q' and the Manhattan distance between the evaluated node location and the location of q'. After evaluating all the empty nodes that are adjacent to placed nodes q_4 and q_0, we will find that the nodes on the east and west of q_4 are the best ones because they are closest to q_4 but not far away from q_0. Here we select the one on the west of q_4 with coordinate $(-1, 0)$. This summation function can help reduce the number of operations for later remapping and achieve the second design objective.

The remaining qubits can be placed in a similar procedure until all the qubits have been placed on the 2D lattice. In this example, q_2 and q_3 are placed on the nodes with coordinates $(0, -1)$ and $(1, 0)$, respectively. All the qubits have their locations (coordinates) on a 2D lattice where we can fabricate one physical qubit on each occupied node. Finally, the nodes with no qubits are removed.

4.4.2 Bus Selection

In the second step, we need to connect the placed physical qubits to enable two-qubit gates. The difficulty comes from the large size of the design space. For N qubits, there

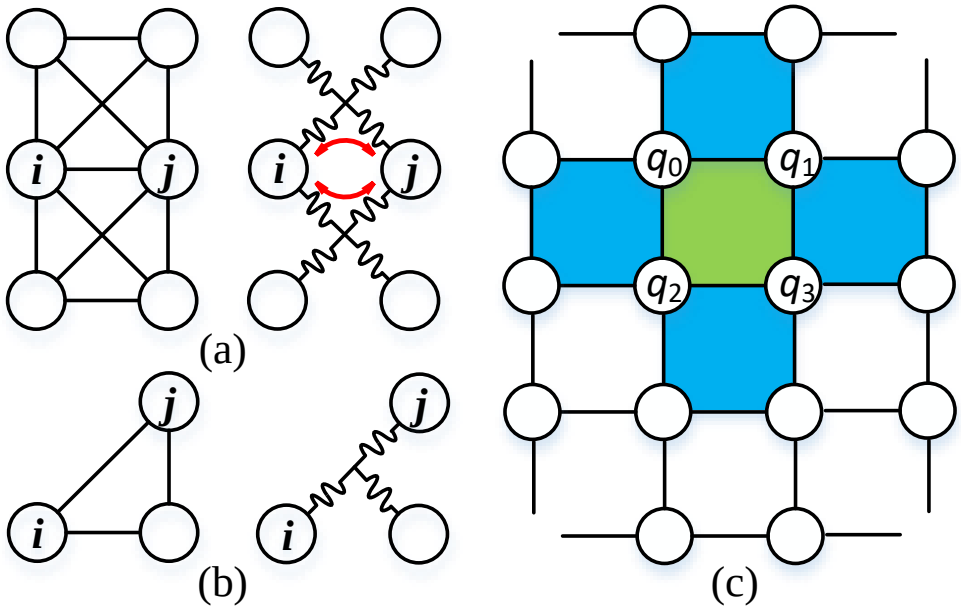


Figure 4.7: (a) Prohibited Condition, (b) Corner Case,(c) Filtered Weight

are $\binom{N}{2}$ distinct qubit pairs. Any of them can be either connected or disconnected so that there are $2^{\binom{N}{2}}$ different cases. Even after considering the nearest-neighbor coupling constraint in which one qubit can only connect with few qubits around it on the lattice, the size of the design space is still $O\left(exp\left(N\right)\right)$. More importantly, more qubit connections will improve the performance but lower the yield rate in general so that we need to identify those connections with the most potential performance benefit in a very large design space.

This chapter simplifies the connection design problem by considering two types of common buses, 2-qubit bus and 4-qubit bus (shown in Figure 4.2). These two types of buses naturally fit in the 2D lattice qubit layout and can be easily fabricated because at most 4 nearby qubits are connected by one bus. After placing the qubits on a 2D lattice in the first step, 2-qubit buses can be directly generated on the edges that connect two occupied nodes but the qubits on a diagonal of a 4-qubit square can never be connected with only 2-qubit buses. Replacing some 2-qubit buses with 4-qubit buses could provide

Algorithm 3: 4-qubit Bus Selection

Input: Geometric coordinates of placed qubits, *coupling strength matrix*, Maximum number of 4-qubit buses K

Output: Locations of 4-qubit Buses

```
Calculate the cross coupling weight for each square;
while K > 0 do
    // Select one square in each iteration
    for square(i, j) in all squares do
        filtered_weight(i, j) = weight(i, j) - weight(i+1, j) - weight(i, j+1) -
          weight(i-1, j) - weight(i, j-1);
    end
    if no square available for 4-qubit bus then
        Break;
    end
    Select the square with the highest filtered_weight;
    Set the weights of squares (i+1, j), (i, j+1), (i-1, j), and (i, j-1) to be 0 and
      mark them to be blocked;
    K = K − 1;
end
```

more qubit connection by trading in yield rate while it is not yet clear where to apply the 4-qubit buses can achieve the Pareto-optimal results. The bus selection subroutine was proposed to identify the locations for 4-qubit buses. Other potential bus designs are left as future research directions and will be discussed in Section 4.6.

Instead of considering the nodes in a 2D lattice, we consider the squares that are naturally formed by the edges in the 2D lattice. Each square can be configured to 2-qubit bus or 4-qubit bus. Now the problem is on which squares we should use 4-qubit buses. The size of search space, even for this 4-qubit bus square selection problem, is still $O\left(exp\left(N\right)\right)$. But the simplification allows us to design high-quality heuristics to guide the selection. Before introducing our solution, one additional prohibited condition must be considered.

Prohibited Condition One physical constraint that we must consider when applying 4-qubit buses is that we cannot have 4-qubit buses in two adjacent squares. The

reason is explained with the example in Figure 4.7 (a). Suppose we have two adjacent squares and both of them are using 4-qubit buses. Then there will be two physical connections between qubit i and j. When we use one of the connections, the other one will bring unexpected effects so that employing 4-qubit bus in one square will immediately block using 4-qubit buses in any of its adjacent squares.

Considering the physical constraints mentioned above, the objectives of this step are summarized as follows:

- Since adding more qubit connections will increase the probability of frequency collision and lower the yield, we hope to apply 4-qubit buses on those squares that can benefit the performance most. In other words, the additional connections are expected to directly support as many two-qubits gates as possible.

- Applying 4-qubit bus in one square will block adjacent squares, making it impossible to directly support some two-qubit gates in those blocked squares. This effect should also be considered when selecting the 4-qubit squares.

We propose a 4-qubit bus selection algorithm to select some squares for 4-qubit buses (pseudocode shown in Algorithm 3). In each iteration, one square that could benefit most from a 4-qubit bus will be selected. Users can specify the maximum number of 4-qubit buses they hope to have. By varying the number of selected squares, a series of architectures can be generated with a trade-off between yield and performance.

To find the most fitting square, we first need to calculate how much one square could benefit from a 4-qubit bus. Since the difference between a 2-qubit bus square and a 4-qubit bus square is whether the qubit pairs on the diagonals are connected, we define the cross-coupling weight for each square as the sum of the coupling strength of the qubit pairs on the diagonals. For the example in Figure 4.7 (c), the cross-coupling weight of the green square is the coupling strength of (q_0, q_3) plus that of (q_1, q_2). A corner case in

the coupling weight computation is the square with only 3 qubits (shown in Figure 4.7 (b)). In such squares, 4-qubit buses can naturally reduce to 3-qubit buses which support coupling between any two of the three connected qubits. The weight of a 3-qubit square is only the weight of logical coupling between the two qubits on one diagonal since the other diagonal only has one qubit. For example, the weight of the 3-qubit square in Figure 4.7 (b) is the (i, j) element in the *coupling strength matrix*. Except for this small modification, 3-qubit squares are treated equally as other 4-qubit squares in our bus selection step. This cross coupling weight can estimate the potential benefit of applying 4-qubit bus in one square and realize the first objective.

However, the cross-coupling weight is not accurate enough to evaluate the benefit of 4-qubit for a square because the prohibited condition is not yet considered. We design a filter to apply this constraint. For each square, the filtered weight is its original cross-coupling weight minus all its neighbors' weights. For example in Figure 4.7 (c), the filtered weight of the green square is its original weight minus the weights of the four blue squares. This filter can take the prohibited condition into consideration and achieve the second objective.

After applying the filter, we will select one square with the highest filtered weight. Then we will label the selected square and its adjacent neighbors so that it will no longer be available for future 4-qubit buses. We also change their weights to zero because they should not affect the 4-qubit selection among the remaining squares. The algorithm will iterate again to select the next square until there are not more squares available or we have already applied enough number of 4-qubit buses.

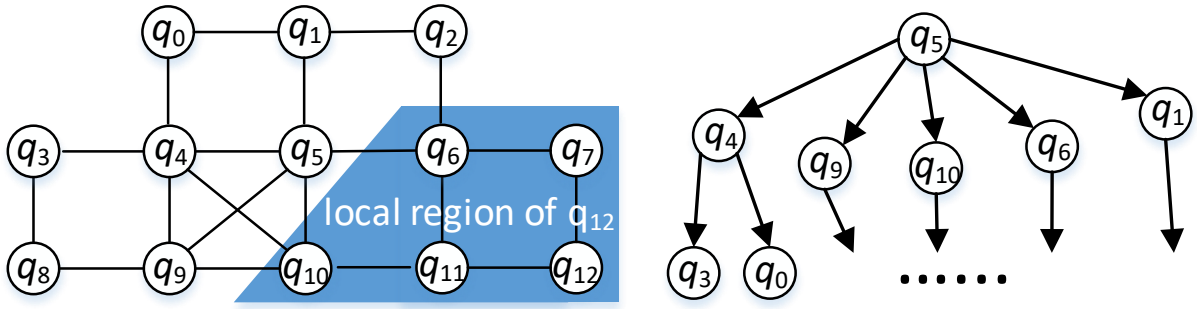


Figure 4.8: Breath First Frequency Allocation

4.4.3 Frequency Allocation

After the two steps above, we now have a complete coupling topology design of a superconducting quantum processor. In the third step, we need to designate the pre-fabrication frequency of each qubit. IBM's 5-frequency scheme is a regular frequency designation [109]. However, the generated qubit layout and connection in our design flow can be irregular since more hardware sources are invested in locations that can benefit the performance most. Thus, we need a more flexible frequency allocation scheme to leverage this unbalanced qubit layout and connection. The objective of this step is to minimize the probability of post-fabrication frequency collision and improve the yield rate. The physical constraints are the frequency collision conditions in Figure 4.3.

Finding the qubit frequency allocation plan to maximize the yield rate is a hard problem. The complex collision conditions make it difficult to find an analytic expression for the yield rate and a brute-force search over all possible frequency configurations will be very time-consuming. For example, if there are M candidate frequencies for each qubit and we have N qubits in total, the total number of possible frequency configurations is M^N. For each of these potential configurations, we need to run a yield simulation (introduced in Section 9) and then select the one with maximal yield rate. This method is not acceptable due to its high complexity. We propose to optimize the qubit frequency allocation algorithm based on the facts that 1) the physical qubits in the geometric center

Algorithm 4: Frequency Allocation

Input: Qubit Location and Connection
Output: Frequency Configuration of Each Qubit

```
Select the qubit in the geometric center of the placed qubits and set its
  frequency to be the middle of the allowed frequency range;
repeat
    Find the next qubit q_i in breadth-first traversal order;
    for temp_freq in all frequency samples do
        Set the frequency of q_i to be temp_freq;
        Simulate the yield rate within q_i's local region;
    end
    Assign the frequency with maximal yield rate to q_i;
until the frequencies of all qubits are determined;
```

of the qubit lattice are more likely to involve in a frequency collision since they usually have more qubit connections, and 2) frequency collision only happens among nearby qubits.

Our algorithm determines the qubit frequencies from the center to the periphery (pseudocode shown in Algorithm 4). Since this step is purely about hardware, the input of our algorithm is only the qubit location and connection generated from the previous two subroutines. To reduce the manufacturing difficulty and help prevent the collision condition 4, we follow the convention from IBM and set an allowed frequency interval $5.00GHz$ to $5.34GHz$. All pre-fabrication frequencies are limited within this interval. First, we locate the qubit that is closest to the center of the qubit lattice and assign its frequency to be the center of the allowed frequency interval. Then we apply breadth-first traversal on the coupling graph from the first qubit in the center. For example, q_5 is the center qubit in the example shown in Figure 4.8. In the breadth-first traversal, we will first access $q_{4,9,10,6,1}$ as shown on the right. Each time we access one new qubit, we will immediately determine its frequency. A list of candidate frequencies is prepared. In this chapter, the candidate frequencies are $5.00, 5.01, 5.02, \ldots, 5.33, 5.34GHz$ to achieve

an accuracy of $0.01GHz$. We can also have more candidate frequencies but it will take more time to evaluate all of them.

To evaluate a candidate frequency on a new qubit, we temporarily assign the candidate frequency to the new qubit and then simulate the yield rate within its local region. The local region of a qubit is defined as a sub-graph of the original chip coupling graph in which a qubit may collide with the new qubit. For example in Figure 4.8, when we are searching for the best frequency of q_{12}, the local region is marked in blue. Note that it is necessary to consider two hops when allocating frequency for one qubit because the frequency collision conditions in row 5, 6, and 7 of Figure 4.3 involve 3 connected physical qubits. Qubits not in this region like q_5 cannot collide with q_{12}. We will select the frequency with the maximal yield rate and assign it to the new qubit. Now the time complexity of the frequency allocation algorithm is $O(MN)$ where M is the number of candidate frequencies and N is the number of qubits.

Yield Simulation

We developed a yield simulator based on IBM's yield model [109, 111]. The fabrication process can be modeled by adding a Gaussian noise $N(0, \sigma)$ to the pre-fabrication frequency of a qubit to generate its post-fabrication frequency where σ is the fabrication precision parameter. For a given superconducting quantum processor design, we estimate its yield rate through Monte Carlo simulation. Each time we will simulate if one fabrication is successful. We first generate the post-fabrication frequencies by adding a random noise sampled from Gaussian distribution mentioned above. Then we check if any frequency collision condition listed in Figure 4.3 occurs in the post-fabrication frequencies. If so, this fabrication fails. Otherwise, it is successful. All possible cases are taken into account. For example, we will examine the two frequencies of all connected physical qubit pairs for condition 1, 2, 3, and 4. If they meet any one of the inequali-

ties of the conditions, frequency collision is considered to occur in this simulation. This simulation process is repeated many times. The yield rate can be estimated by the ratio between the number of successful simulations and the total number of simulations.

4.5 Evaluation

To demonstrate that the proposed application-specific architecture design flow can deliver hardware designs with better Pareto-optimal results in terms of performance and yield rate, we conduct experiments over various benchmarks to show not only the overall improvement but also the breakdown of benefits from each of our hardware design subroutines.

4.5.1 Experiment Setup

Benchmarks Twelve quantum programs are collected from IBM's Qiskit [76] and RevLib [91], or compiled from ScaffCC [117]. These benchmarks cover several important domains (e.g., simulation, arithmetic) and have various sizes (from 7- to 16-qubit) for a versatility test of the proposed design flow.

Metrics To evaluate the efficiency of an architecture, we need both the yield rate and performance. An architecture with a higher yield rate can be successfully fabricated with fewer attempts, indicating a lower hardware cost. In our experiments, the yield rate is simulated with IBM's yield model [109, 111] as introduced in Section 9. For the performance evaluation, we adopt the total post-mapping gate count metric widely used in previous studies [77, 56, 18]. More gates lead to longer execution time and a larger probability of error on quantum computing devices. If a hardware architecture could execute the program with fewer gates, then its performance is considered to be better.

Yield Simulation Configuration The number of trials in the Monte-Carlo simula-

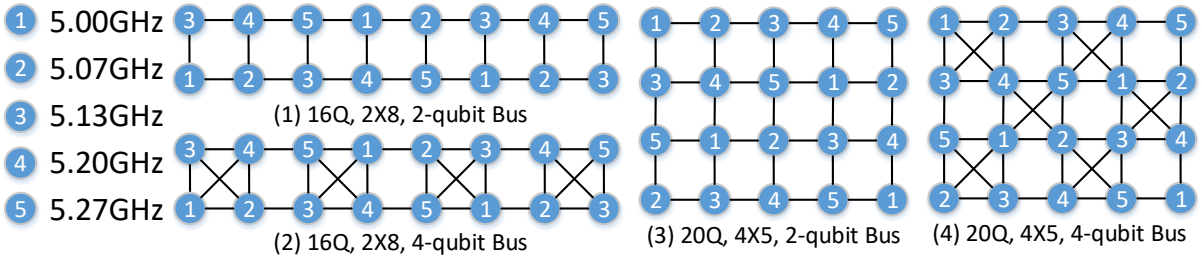


Figure 4.9: Baseline Qubit Frequency, Layout,and Connection Designs

tion for each architecture is 10,000∼ 100,000, which is 10 ∼ 100× of that used in IBM's experiments [111, 128, 125] to ensure the simulation accuracy. The fabrication precision parameter σ is set to be $30MHz$, a realistic extrapolation of progress in hardware by IBM [109, 125]. IBM has improved the σ from $200MHz$ [129] to $130MHz$ [109] in the last few years and $30MHz$ is a reasonable projection to achieve a useful yield as predicted by IBM [125].

4.5.2 Experiment Methodology

To illustrate the benefit of our design flow, five experiment configurations are designed to show the overall improvement and the performance/yield trade-off gain at each of the three subroutines in Section 4.4. Among them, **ibm** is a set of general-purpose architectures from IBM and they are not tailored for any applications. The remaining four configurations are application-specific architectures generated by the entire or part of the proposed design flow.

ibm We use IBM's design scheme as the baseline configuration. It has two layout options, a 2×8 lattice with 16 qubits, and a 4×5 lattice with 20 qubits. The qubit connection design can be either 2-qubit bus only or using 4-qubit buses as many as possible. In total, there are four architectures combining the layout and connection options (shown in Figure 4.9). The frequency allocation scheme is a 5-frequency scheme [109, 125]. The

five frequencies are an arithmetic progression from $5GHz$ to $5.27GHz$ and their arrangement is also in Figure 4.9.

eff-full We apply all three subroutines and generate a series of efficient superconducting quantum processor architectures by varying the number of 4-qubit buses. The number of designs we can obtain for a quantum program depends on the number of qubits as more qubits can provide more squares to apply 4-qubit buses in the generated layout. In this chapter, we obtain the **eff-full** data series through iterating over all possible numbers of 4-qubit buses in the second subroutine for bus selection. This experiment can show the overall architecture design improvement when comparing with the baseline **ibm**.

eff-5-freq We only apply the first two subroutines to generate qubit layout and connection design but the frequency allocation is done with IBM's 5-frequency scheme. The yield benefit from the proposed frequency allocation algorithm can be demonstrated by comparing with results from **eff-full**.

eff-rd-bus We keep the first and the third subroutines but randomly select some squares to employ 4-qubit buses with the prohibited condition constraint satisfied. This will demonstrate the effect of our filtered-weight-based 4-qubit bus selection algorithm by comparing with results from **eff-full**.

eff-layout-only We apply our profiling method and perform a layout design. The connection design has two options. One is only using 2-qubit buses. The other is using 4-qubit buses as much as possible. The frequency design follows the baseline **ibm**. The benefit of our layout optimization can be shown when comparing with the results from **ibm**.

For each benchmark, we run all the five configurations to generate different superconducting quantum processor architectures with different yield rates. Then we apply one state-of-the-art qubit mapping algorithm [18] on these architectures to obtain the total number of gates when running the generated or baseline architectures.

4.5.3 Overall Improvement

Figure 4.10 shows the result of yield and performance for all benchmarks and the five experiment configurations. There are 12 subfigures and one subfigure contains the results of the five experiment configurations for one benchmark. The X-axis represents the normalized reciprocal of post-mapping gate count and data points on the **right** have better performance. The Y-axis represents the yield rate and data points on the **top** have higher yield rates. The legend at the bottom of Figure 4.10 shows the markers for the five configurations. The data points for the four designs in the baseline are labeled by (1), (2), (3), and (4), according to Figure 4.9.

Optimality The optimal solution in this chapter means the Pareto-optimal solution in terms of post-mapping gate count and yield rate. A series of architectures with better Pareto-optimal results can be generated by our design flow as the data of **eff-full** is on the upper right of **ibm**. The most simplified designs (the most left top blue triangle data point in **eff-full**, zero 4-qubit buses) generated by our design flow outperforms the 16-qubit baseline design (data point (1) in **ibm**) without 4-qubit buses in both performance ($\sim 7.7\%$) and yield rate ($\sim 4\times$). Compared with the 16-qubit baseline with four 4-qubit buses (data point (2) in **ibm**), our designs with zero 4-qubit buses achieve over $100\times$ better yield rate with $< 1\%$ performance loss. On the other side, compared with IBM's 20-qubit chip design with six 4-qubit buses (the baseline design with the most hardware resources, data point (4) in **ibm**), the designs with the maximum number of 4-qubit buses generated from our design flow (the data points on the most bottom right in **eff-full**) have over $1000\times$ yield rate improvement on average with only about 3.5% performance loss.

Controllability The proposed design flow can easily control the trade-off between yield and performance by only changing the number of 4-qubit buses without traversing

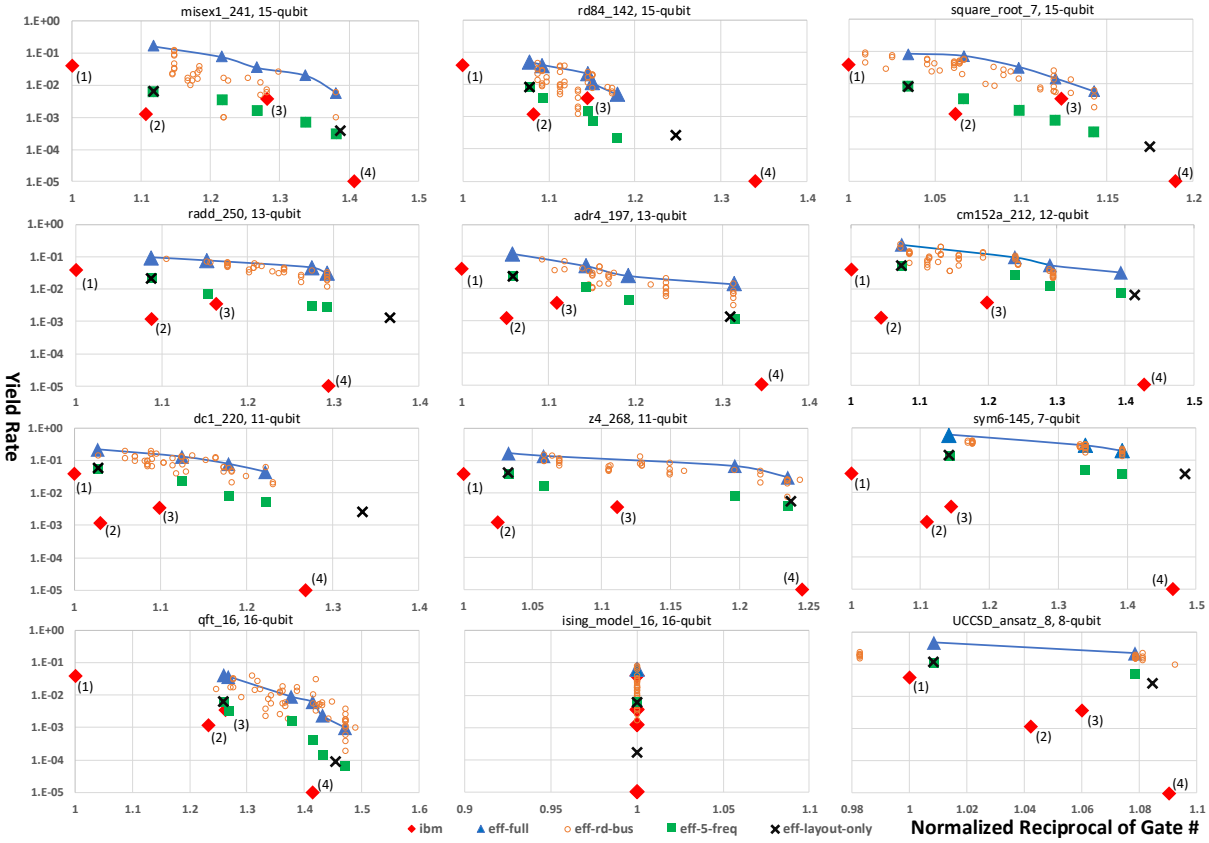


Figure 4.10: Yield v.s. Normalized Reciprocal of Post-mapping Gate Count

across, or sampling a large number of designs in, the entire search space. Depending on the number of qubits in different target programs, we can trade in around $10\times \sim 50\times$ yield rate for $10\% \sim 33\%$ performance improvement.

Special Case

The results of *ising_model* are significantly different because the logical qubit coupling in this benchmark forms a chain structure. The mapping algorithm can always find the perfect initial mapping without inserting additional operations. As a result, the post-mapping gate count is the same for all tested hardware architectures. All data points for this program lie in one vertical line. Only one architecture is generated from our design flow because there is no need to add 4-qubit bus. All the two-qubit gates can be executed

through the edges on the 2D lattice. There are no two-qubit gates applied on two qubits on a diagonal because of the chain coupling structure. In this case, 4-qubit buses can only lower the yield rate without improving the performance.

4.5.4 Effects from Individual Subroutines

The overall improvement has already been discussed, but one interesting question is how much improvement the layout and connection optimization contribute and how much comes from the optimized yield allocation directly. The five configurations decouple the proposed design flow and provide a breakdown of the effect of individual subroutines.

Effect of Layout Design

The difference between **ibm** and **eff-layout-only** illustrates the effect of layout design since the rest two subroutines are the same. An architecture with more hardware resources is expected to provide higher performance by allowing more flexibility in qubit mapping. But our optimized layout design could use comparable or fewer hardware resources while the performance can be even better. For example, we compare the 2-qubit bus only data point (the upper left one) with the 16-qubit baseline with four 4-qubit buses (labeled by (2) in each subfigure). **eff-layout-only** provides better or comparable performance most of the time with about 35× yield improvement on average. The improvement at this step depends on the program size and programs with fewer qubits will use fewer qubits and connections in an optimized architecture. This result proves that our layout design could generate qubit layout with high performance but using much fewer hardware resource for different programs.

4-qubit Bus Selection Quality

By comparing the results from **eff-full** and **eff-rd-bus**, we can see that the architectures generated from our bus selection algorithm are better than that of random selection in trading in yield for performance most of the time. The data points of **eff-rd-bus** reveal the distribution of the yield and performance sampled from random bus designs. Note that the performance of **eff-rd-bus** is usually confined by the two data points in **eff-layout-only** because adding connections can improve the performance most of the time. For most benchmarks except *qft*, the results from **eff-full** are close to the upper bound formulated by the random samples, which shows that our weight-based bus selection could generate a series of near Pareto-optimal hardware architectures with various numbers of qubit connections.

The result of *qft* is much worse than that of other programs due to the unique uniform two-qubit gate pattern in this program. The number of two-qubit gates between arbitrary two logical qubits is always two in *qft*, which makes all the logical qubit pairs are the same in the sense the coupling strength during profiling. Then in bus selection subroutine, all the squares share the same weight and the weight-based selection is the same as random selection.

For the two small benchmarks, *sym6* and *UCCSD_ansatz*, the number of available squares in the generated qubit layout is small and there are very few options when applying 4-qubit buses. Therefore, most of the architectures generated from the random 4-qubit bus selection are the same as those from the proposed design flow, which makes the results from **eff-full** and **eff-rd-bus** very close.

Frequency Allocation Optimization

By comparing **eff-full** and **eff-5-freq**, we can see that the proposed frequency allocation algorithm provides about 10× yield rate improvement on average. This improvement is slightly worse when the yield from the baseline 5-frequency is already high, e.g., results from *sym6* and *UCCSD_ansatz*. The fabrication variance makes the ideal yield 100% unreachable and it is hard to optimize yield when it is already high.

4.6 Discussion

This chapter studies application-specific efficient superconducting quantum processor design. In particular, we formalize the architecture design for superconducting quantum processors with three key steps, each of which comes with an optimization subroutine. This is the first attempt, to the best of our knowledge, to identify the optimization opportunity from the architecture level to push forward the balance between quantum computing performance and hardware yield rate. Effort towards this direction can be of significant demand in the near term quantum computing with limited computation resource and immature fabrication technology.

Although we show that improved Pareto-optimal designs can be generated with a static program analysis and three optimized design algorithms, several future research directions can be explored as with any initial research.

Improving Profiling Method This chapter focused on the logical qubit coupling topology in a quantum program but other patterns may also be leveraged. We omitted the temporal information of the two-qubit gates and all information about other program components. But the locations of two-qubit gates in a quantum program may also be leveraged for finer-grained evaluation of the coupling strength for different logical qubit pairs at different times during the execution. The single-qubit patterns can also help

with the basic gate set design.

Exploring More Design Space In the proposed design flow, the number of physical qubits is the same as that of logical qubits for higher yield rate. However, we can still add auxiliary physical qubits since they can also be used during the qubit routing, trading in more yield rate for higher performance. How to add auxiliary qubit to appropriate locations and how to connect them are interesting problems to explore in the future. To ensure modularity and scalability, the qubits are forced to be embedded in a 2D lattice and only consider two types of buses lying in the lattice. However, the qubit placement and connection could be more flexible if we trade in part of the scalability. For example, one bus could also connect more than four qubits [130]. The design space in this direction is not yet explored.

Optimizing Frequency Allocation This chapter tried to optimize the qubit frequency selection from the center to periphery and only searched for the optimal frequency for one qubit, resulting in a sub-optimal frequency allocation. A global optimization like formal methods can be explored to further optimize the frequency allocation result. One alternative approach to resolve the frequency collision issue is to use flux-tunable transmon qubits [82], of which the frequencies can be dynamically tuned with additional control signals. The design trade-off of different types of qubits is not yet explored and additional signals bring more noise and increase the control complexity. The proposed design flow is still valuable even with frequency-tunable qubits because the simplified architectures with fewer the on-chip connections can not only reduce the fabrication complexity but also benefit the overall performance by lowering the crosstalk error.

4.7 Related Work

This chapter ranges across multiple topics, i.e., program profiling, superconducting processor design, application-specific design, qubit mapping. We briefly introduce related work for all of them.

Application-specific Design The closest related work is SPARQS, a superconducting planar architecture proposed by Wilhelm *et al.* [123, 124] targeting a specific Fermi-Hubbard model simulation program. However, they only provide an implementation-independent design from theoretical physics level. This chapter formalizes a systematic end-to-end design flow with automatic program profiling and realistic physical constraints included, for the first time. With no limitation on the target program, we can generate a series of Pareto-optimal hardware architecture designs in a controllable way.

Quantum Program Profiling and Analysis Program profiling and analysis are very important for software and compiler optimization. Previous works on quantum program analysis [117, 118, 119, 120, 121, 122] have studied entanglement, termination, non-cloning checking, etc. The profiling method in this chapter is proposed to guide the hardware design, fulfilling a different goal.

Superconducting Quantum Processors As one of the most promising candidate technology to implement quantum computing, superconducting quantum techniques have been employed in two mainstream quantum computation models. The circuit model based processors [37, 52, 51] support quantum circuit model [25] and the quantum annealers [131] can implement adiabatic quantum computing [29]. Their programming model and hardware architecture are different for these two quantum computing approaches. The design flow in this chapter is proposed for circuit model based quantum processors while efficient quantum annealer design can be a future research direction.

Qubit Mapping Formal and heuristic methods have been attempted to solve this

problem [64, 60, 18, 77, 56] and minimize the total gate count. Recently several studies [85, 132, 104] have applied the actual gate error rates for fine-grained optimization. All these optimizations are pure software-level modification. This chapter attempts to improve the performance by reducing the mapping overhead from the hardware level. We adopt the gate count metric to estimate the mapping overhead since our experiments are performed on artificial hardware architectures.

4.8 Conclusion

The demand for larger computation capability in a superconducting quantum processor naturally calls for more hardware resources which will also increase the design complexity and lower the yield rate. This chapter explored application-specific architecture design for superconducting quantum processors to achieve both high performance and higher yield rate. Gate patterns in a quantum program can be extracted by the proposed profiling method and then utilized in the follow-up hardware architecture design. Three subroutines are designed to generate the qubit layout, connection, and frequency respectively with physical constraints taken into consideration. Experimental results show that the proposed design flow could deliver architectures with both high yield rate and performance automatically for different applications except those with extremely special gate patterns.

Chapter 5

Software-Hardware Co-Optimization for Computational Chemistry on Superconducting Quantum Processors

5.1 Introduction

Computational chemistry is an important domain in scientific computing that employs computer simulation to help understand and predict the properties of chemical systems like molecules [133]. It has broad applications in chemistry [134], biology [135], and material science [136]. However, simulations of large chemical systems quickly become intractable as the laws governing them lead to equations too complicated to solve efficiently on classical computers [137]. For example, more than 1 million node-hours on the Summit supercomputer were recently allocated to chemistry and materials simulation [138].

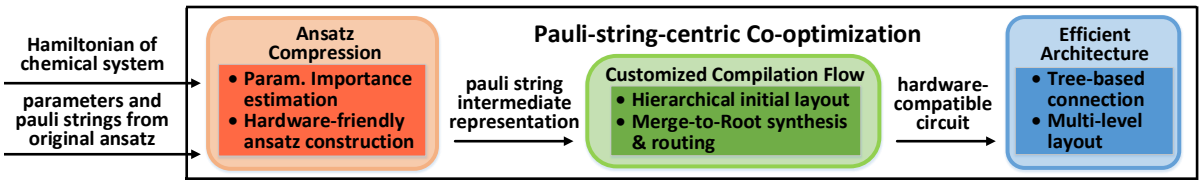


Figure 5.1: Overview of the proposed Pauli-string-centric software-hardware co-optimization

Fortunately, quantum computers are naturally suited to solve problems in computational chemistry. In fact, this was the original motivation for Feynman's proposal to build a quantum computer [2]. A leading algorithm for this task is known as the Variational Quantum Eigensolver (VQE), which has relatively modest requirements in terms of number of qubits and depth of computation, and shows some robustness to errors, all favorable properties for near-term quantum computing [48, 139]. Small-size molecular simulations using VQE have been experimentally demonstrated with superconducting quantum circuits [140, 47, 141, 142] and other technologies [143, 144, 145, 146].

Despite the recent progress, larger-scale chemistry simulations are not yet feasible on quantum devices. We argue that this is primarily due to three shortcomings in current quantum computing technologies: 1) large program (circuit) size, 2) inefficient hardware architecture, and 3) deficient compiler optimizations. Each piece is under active research, but rarely in a collaborative way, leading to insufficient overall improvement. To the best of our knowledge, there is no existing united co-optimization solution throughout the application, hardware, and compiler stacks in quantum computing. In this chapter we make the case that co-optimizing all three of them can dramatically optimize the overall execution, allowing quantum applications to scale much sooner.

While the co-design principle has been shown to be effective [147], it is challenging as the design objectives of different technology stacks may contradict each other. We briefly review some of these challenges below.

Application: Optimizations to reduce the size of VQE circuits have been mostly

done theoretically, ignoring the actual execution on the underlying hardware [148, 149, 150, 151, 152, 153]. VQE is an iterative optimization algorithm and more parameters to optimize over can result in better accuracy. However, this adversely leads to larger circuits and longer time to converge, both undesirable on near-term quantum hardware [149]. Making the program hardware-friendly [47] without keeping its general chemistry structure could prevent it from converging to the right solution effectively [154].

Hardware architecture: The quality of superconducting quantum processors has steadily improved in the past few years, while the progress is usually measured by metrics that are oblivious to application performance [78, 106, 82, 112, 155, 156]. Applications generally require high qubit connectivity, but this will cause adverse crosstalk and low yield during device fabrication [111, 157, 158]. Making connections sparse will lead to high qubit mapping overhead during application execution [108].

Compiler optimizations: State-of-the-art quantum compilers [159, 160, 161] mostly perform optimizations at the gate level where it is easier to reason about program optimization [162, 163, 164, 104], but they miss a large optimization space when compiling VQE programs because they do not exploit the synergy of domain knowledge and hardware information.

In this chapter, we co-optimize the algorithm, hardware, and compiler for VQE on superconducting quantum processors through a key observation that *optimizations at different technology stacks can be coordinated through* ***Pauli strings*** *and their simulation circuits*. Pauli strings arise naturally as fundamental building blocks in quantum chemistry simulation. Their unique semantics and structure can be carried through the stack to guide all aspects of the design. At the algorithm level, the VQE program is dominated by Pauli string simulation circuits. The molecule's Hamiltonian (energy operator to be estimated) is also represented by an array of weighted Pauli strings. We find that the geometrical interpretation of Pauli strings can effectively compress the VQE circuit to

estimate the same solution with much lower cost. At the hardware level, the gate pattern of Pauli string simulation circuits makes it possible to efficiently support their execution with very few on-chip connections. Moreover, Pauli string simulation circuit synthesis is flexible, allowing us to tailor the compilation flow when deploying VQE programs to the underlying hardware. Such property makes it possible to achieve very low execution overhead even on a sparsely-connected hardware architecture.

Our Pauli-string-centric software-hardware co-optimization is shown in Figure 5.1. **First**, we introduce a novel VQE circuit compression strategy that takes the Hamiltonian of the target chemical system as an additional input. The impor-tance of each parameter in the VQE circuit is estimated by *comparing the Pauli strings* of the circuit with the target Hamiltonian. Only those parameters that are expected to signi-ficantly affect the final result are kept in a hardware-friendly order. The output of this step is an array of Pauli strings and their parameters, which can be considered as a new interme-diate representation (IR) above quantum circuits. **Second**, we design an *X-Tree* superconducting quantum processor archi-tecture that is extremely sparse as it uses the minimal number of physical connections. The sparsity significantly boosts the processor reliability and yield rate. Yet, it does not sacrifice performance as the connectivity structure is well-suited for the structure of Pauli string simulation circuits that appear in various chemistry and physics applications. **Third**, we pro-pose a new compilation flow that converts the Pauli string IR directly into executable quantum circuits for the X-Tree ar-chitecture. We determine qubit layouts directly from the Pauli strings (termed *hierarchical initial layout*). We also perform synthesis and mapping in one step (termed *Merge-to-Root*). We show that relying on this higher-level IR, our compiler can map the program to hardware with negligible overhead, as it can adaptively synthesize *each* Pauli string according to the current mapping and the underlying X-Tree architecture.

Our co-designed stack is not limited in programmability and can accommodate a

wide range of problems in chemistry and physics that are naturally represented by Pauli strings. We show a comprehensive evaluation by simulating various molecules of different sizes and structures. Results show that our co-optimization outperforms conventional VQE setups with significant program size reduction, faster convergence speed, mild simulation accuracy loss, more efficient hardware design, and negligible compilation mapping overhead.

Our key contributions can be summarized as follows:

- We discover a Pauli-string-centric co-optimization opportunity that can broadly advance variational quantum chemistry simulation of various chemical systems on superconducting quantum processors.

- We propose three novel optimizations for VQE algorithms, quantum compilers, and superconducting hardware architectures, respectively. Each of them not only focuses on the design objectives of one individual technology but also considers the optimizations in other system stacks.

- Our experiments show that our approach outperforms conventional setups of VQE on superconducting quantum processors across a wide range of criteria from software to hardware. On average for nine molecules, when using a 50% parameter compression ratio, our technique can achieve about 2.5× convergence speedup and only incur 0.05% error in the simulated energy. It also achieves 99.7% mapping overhead reduction on an optimized architecture with 8× fabrication yield improvement.

5.2 Background

In this section, we introduce the necessary background to help understand the proposed co-optimization. The quantum computing basics concepts have been covered in previous chapters and we will focus on the background about quantum chemistry simulation.

5.2.1 Pauli String and Its Time Evolution Circuit

The central building blocks of chemistry simulation circuits are Pauli string operators. An n-qubit Pauli string P is an array $P = G_{n-1}G_{n-2}\dots G_0$ where $G_i \in \{I, X, Y, Z\}$ for the ith qubit and $0 \leq i < n$. X, Y, Z are the three Pauli operators and I is the identity operator.

Time evolution: In quantum physics, the time evolution of a system is determined by the system Hamiltonian H, and the unitary that represents this time evolution is $exp(i\theta H)$ where θ is a parameter to represent time. Usually, we do not directly implement $exp(i\theta H)$ in a quantum circuit since it is hard to directly synthesize $exp(i\theta H)$ into basic single-qubit and two-qubit gates efficiently. Instead, we first decompose H into a weighted sum of Pauli strings, i.e., $H = \sum_j w_j P_j$ where P_j is a Pauli string and $w_j \in \mathbb{R}$ is its weight. The time evolution of Pauli strings $exp(i\theta P_i)$ can be easily synthesized.

Pauli string simulation circuit: We introduce the synthesis of Pauli string simulation circuits with the examples in Figure 5.2. Suppose the Pauli string is $XIYZ$ on four qubits and the time parameter is θ. The circuit in Figure 5.2 (a) shows the synthesis result. The first layer consists of some single-qubit gates. The rule is that if the operator on one qubit is X (e.g., q3), then we apply an H (Hadamard) gate. If the operator on one qubit is Y (e.g., q1), then we apply a Y gate. If it is I (e.g., q2) or Z (e.g., q0), no single-qubit gate is required. After applying the single-qubit gates, several CNOT gates

will connect all qubits whose corresponding operators are not I in the Pauli string. In this example, the CNOT gates connect q0, q1, q3 because the operator on q2 is I. We can first connect q0 and q1 and then connect q1 and q3, as shown in Figure 5.2. Then, a rotation gate is applied to rotate angle 2θ along the Z axis on the last qubit in the CNOT connections (i.e., q3). Finally, the CNOT gates and single-qubit gates are applied again in the reverse order. In summary, the Pauli string will determine the outermost single-qubit gates and the CNOT gates. The parameter will only affect the rotation angle of Z-rotation gate in the middle.

Flexible synthesis: The most expensive components for executing a Pauli string simulation circuit on a near-term superconducting quantum processor are the CNOT gates before and after the middle rotation gate. Across various qubit technologies today, (non-local) CNOT gates have an order of magnitude larger latency and error compared to (local) single-qubit gates. During the synthesis of a Pauli string simulation circuit, there is flexibility in the pattern of CNOT gates used. For example, the three circuits in Figure 5.2 (b)(c)(d) show three equivalent synthesis result variants of $exp(i\theta ZZZZ)$. The requirement of the CNOT gates is that they must be connected in a *tree structure* and the CNOT gates are then applied from the leaves to the root (the center rotation gate is applied on the root qubit). The tree structures of the three variants are shown below the corresponding circuit examples. Each qubit is a node and the CNOT gates are represented by directed edges connecting the nodes. We leverage this flexibility to guide our hardware architecture design and compiler optimizations.

5.2.2 Variational Quantum Computational Chemistry

We use the example in Figure 5.3 to briefly introduce the basics of VQE algorithm for chemistry simulation. We recommend [5] for further details.

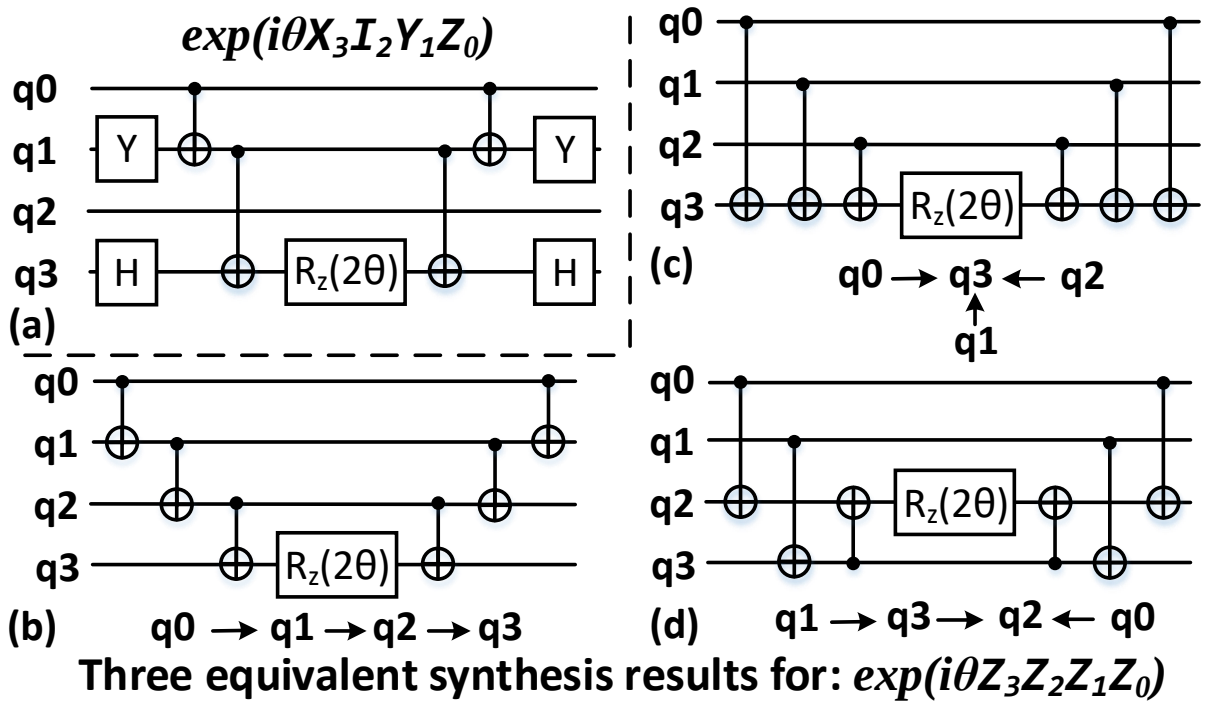


Figure 5.2: Pauli string simulation circuit synthesis examples

Problem encoding

The first step is to encode the simulation problem, for example a Hydrogen (H_2) molecule at a specific bond length (on the left of Figure 5.3). To simulate the state of the electrons, we map four candidate orbitals (basis states) that an electron may occupy, and then obtain the system Hamiltonian through standard chemistry tools like PySCF [165]. This step is not the focus of our work.

Circuit construction

After we map the orbitals to qubits, we need to construct a circuit that can generate a state to represent how the electrons occupy the orbitals. Figure 5.3 shows the overall structure of this circuit. After all the qubits are initialized to $|0\rangle$ state at the beginning, the first part on the left is a shallow circuit (applying X gates on some qubits) to prepare

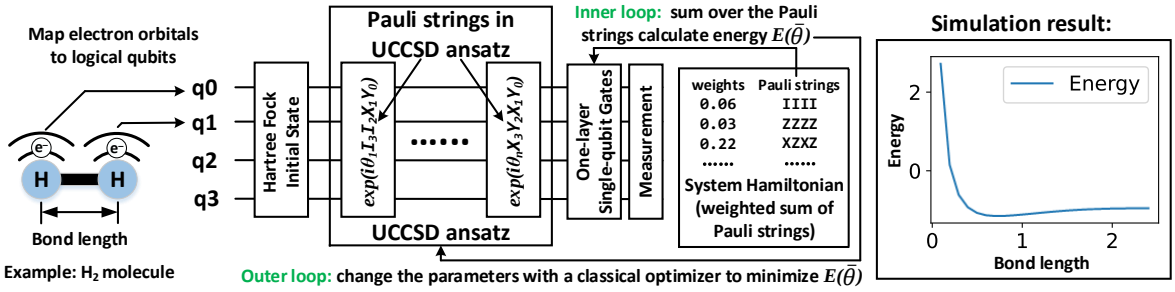


Figure 5.3: Example of variational quantum chemistry simulation flow and result

an initial state. We use the default Hartree-Fock initial state [166]. On the right is one layer of single-qubit gates to change the basis prior to measurement, based on the different terms present in the target molecule's Hamiltonian. These two components only make up a small portion of the entire simulation circuit. In this work, we focus on the middle part of the circuit: the parameterized state preparation circuit which is known as *ansatz* in the quantum computational chemistry. The parameters of this circuit are what get optimized during execution. *This ansatz part makes up the vast majority of the quantum subroutine and is the target of our co-optimization.*

Execution flow

The execution flow of VQE has two major loops. For a given set of parameters (denoted by $\bar{\theta}$), we first execute the circuit to generate state $\left|\psi(\bar{\theta})\right\rangle$. Then we measure the expectation value $\left\langle\psi(\bar{\theta})\right| P_i \left|\psi(\bar{\theta})\right\rangle$ where P_i is a Pauli string in the decomposition of H. We iterate over all P_is in H to obtain $E(\bar{\theta}) = \sum_i w_i \left\langle\psi(\bar{\theta})\right| P_i \left|\psi(\bar{\theta})\right\rangle$. Changing to measuring different P_is only needs to change the last layer of single-qubit gates and the parameters are not changed in this inner loop in Figure 5.3. After $E(\bar{\theta})$ is obtained, a classical optimizer will change the parameters $\bar{\theta}$ to minimize $E(\bar{\theta})$. This optimization may take many steps to converge and this is the outer loop in Figure 5.3. In this work, we optimize both the inner loop and outer loop: we discard less important parameters for

faster convergence and reduce the circuit cost at each iteration by focusing on important sub-circuits and better mapping. Finally, we obtain a minimal energy $E(\bar{\theta})$ (representing the ground state) of the H_2 molecule under the specified bond length. In a typical simulation task, we will simulate different bond lengths and record ground state energies for these different configurations.

Simulation result interpretation

The result of the H_2 simulation is on the right of Figure 5.3. The X- and Y-axis represent the bond lengths and the simulated ground state energies, respectively. The minimal simulated ground state energy is achieved when the bond length is around 0.7 Å (1Å= $10^{-10}m$ and we sample the bond length every 0.1Å). The actual bond length measured by physical experiments is 0.74Å, which is consistent with the simulation result.

5.2.3 UCCSD Ansatz

The widely-used UCCSD (Unitary Coupled Cluster Singles and Doubles), a chemistry-inspired ansatz [167, 168], is the 'standard' ansatz for variational chemistry simulation. The terms in a UCCSD ansatz are similar to those in the Hamilto-nian of a chemical system. Therefore, it is expected that tuning the parameters in UCCSD can make a 'good' guess about the ground state. A UCCSD ansatz of n qubit has $O(n^4)$ parameters and each parameter corresponds to some Pauli strings. When implementing UCCSD in a circuit, it becomes a series of Pauli string simulation circuits with parameters, as shown in the middle of Figure 5.3. Implementing a UCCSD ansatz is very expensive on a superconducting quantum processor due to its large number of parameters and CNOT gates in the synthesized circuit. Our techniques will tailor the ansatz, architecture, and compiler for each other to significantly reduce the cost.

5.3 Ansatz Compression

To enable chemistry simulation of larger size problems, we first propose to optimize the simulation program at the algorithm level. We will focus on optimizing the parameterized ansatz because it makes up most of the program. The objectives of the ansatz optimization are summarized as follows:

- **Small:** The constructed ansatz should have a small size, i.e., fewer parameters and gates, for shorter execution time and higher fidelity on near-term devices.
- **Accurate:** The simulation accuracy should not degrade too much using a smaller ansatz with fewer parameters.
- **Hardware friendly:** The generated ansatz can be mapped onto the target hardware without too much overhead.

Our optimization will start from the UCCSD ansatz, the well-accepted standard ansatz with a large number of parameters ($O(n^4)$ parameters for n qubits). We seek to eliminate those parameters that contribute the least to final measurement results. Doing this precisely for each parameter can be very complex. Fortunately, in variational algorithms, we do not have to be very precise, as long as the optimization can converge in a reasonable amount of time. The key is to have enough parameters to explore the optimization space and move towards the answer by adjusting those parameters at each iteration. Thus, we only need to *estimate* whether a parameter is more likely or less likely to affect the final measurement results. The ansatz can be compressed by only selecting those circuit components with critical parameters. The effectiveness of our parameter importance estimation method can be empirically verified later. In the rest of this section, we first study how to estimate the importance of each parameter in the UCCSD ansatz. Then we introduce how to construct the ansatz in a hardware-efficient manner.

5.3.1 Parameter Importance Estimation

In a VQE simulation, the final observable, which is the Hamiltonian of the target chemical system, is an array of weighted Pauli strings. The UCCSD ansatz itself is also an array of Pauli string simulation circuits with their corresponding parameters (one parameter can be shared by multiple Pauli strings). We first estimate how likely the parameter tuning of each Pauli string in the ansatz can affect the final measurement and then assemble the results to estimate the importance of each parameter. The pseudo code of this importance estimation is in Algorithm 5. For a given Pauli string (denoted by P_a) in the ansatz, we compare it with each Pauli string (denoted by P_H) in the Hamiltonian. We explain the Pauli string comparison method with an example of P_a and P_H shown on the left of Figure 5.4. For the two Pauli operators on the same qubit q_s in the two Pauli strings being compared, we have the following three cases that will make P_a less likely to affect the measurement result of P_H:

1. If the Pauli operator in the P_a is 'I' (e.g., q3), then this Pauli string simulation circuit will not apply any gate on q_s (as shown in Figure 5.2 (a)) and this will make P_a less likely to affect the measurement result of P_H.

2. If the Pauli operator in the P_H is 'I' (e.g., q2), then when measuring this Pauli string simulation circuit in the Hamiltonian, the measurement result on this qubit q_s will be always be 1 and will never be changed with respect to the parameter. This makes P_H less sensitive to parameter tuning in P_a.

3. If the two Pauli operators on q_s are the same (e.g., q1), then the effect of changing the parameter in P_a will be reduced on the measurement results of P_H. Figure 5.5 is a geometrical explanation. The state vector of a single qubit can be considered as a unit vector on the Bloch sphere in a three-dimensional Euclidean vector space

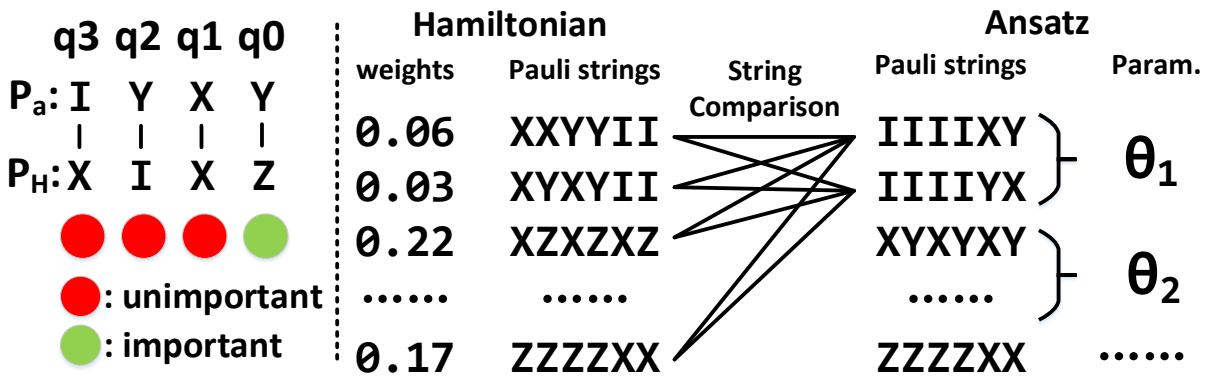


Figure 5.4: Importance estimation example

(Figure 5.5 (a)). X, Y, Z can represent three orthogonal axes. When applying $exp(-i\theta P)$ ($P \in \{X, Y, Z\}$) on a state vector $|\psi\rangle$, the state vector on the Bloch sphere will rotate around the corresponding axis. For example, in Figure 5.5 (b), the state is rotating around the X-axis after $exp(-i\theta X)$ is applied on it. Such rotation will not change the result when we project the state $|\psi\rangle$ onto the same axis, and therefore will not change the measurement result when the observable is X.

The only case left is when the two Pauli operators on q_s are the different (e.g., q0). In this case, changing the parameter is very likely to affect the measurement result because rotation along one axis can change the projection onto another axis. For example, in Figure 5.5 (b), the projection result on the Y axis is changed after a rotation along the X-axis is applied.

Suppose the number of qubits on which the Pauli operators satisfy any of the three conditions above is d and we have $d = 3$ in this example. How likely tuning the parameter of P_a will affect the measurement result of P_H is estimated to be the absolute value of the weight of P_H multiplied by an exponential decaying term 2^{-d}. We repeat this process for all P_Hs in the Hamiltonian and obtain a score of P_a in the ansatz. After we obtain the scores of each Pauli string in the ansatz, the importance of each parameter equals

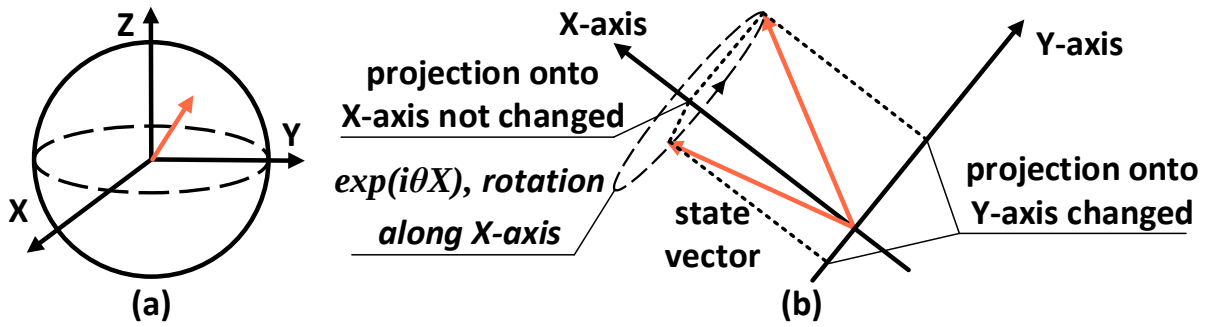


Figure 5.5: (a) Block sphere with three axes, (b) Effect of state vector rotation

Algorithm 5: Parameter Importance Estimation

Input: Weighted Pauli strings of target Hamiltonian H, Pauli strings of one parameter θ

Output: Importance score of parameter θ

```
importance_score = 0;
for P_a in all Pauli Strings of parameter θ do
    for P_H in all Pauli Strings in H do
        Obtain the importance decay factor d by comparing P_a and P_H;
        score += 2^{-d} × abs(weight of P_H);
    end
end
```

the sum of the scores of all that parameter's corresponding Pauli strings (note that one parameter can be shared among multiple Pauli strings). For example, the importance of θ_1 in the example ansatz on the right of Figure 5.4 is the sum of the scores of the first two Pauli strings ($IIIIXY$ and $IIIIYX$). The importance of the rest parameters can be calculated similarly. The time complexity of our ansatz compression algorithm is $O(n\#(P_a)\#(P_H))$ where $\#(P_a)$ and $\#(P_H)$ are the numbers of P_as and P_Hs, respectively, and n is the number of qubits.

5.3.2 Hardware-friendly Ansatz Construction

After the importance of each parameter is determined, we can construct the new ansatz and achieve the three objectives mentioned above. **First**, since a small size with

fewer parameters and Pauli string simulation circuits is expected, we will select only part of the parameters and Pauli string simulation circuits from the original UCCSD. The size of the constructed ansatz can be determined by a given compression ratio. **Second**, simulation accuracy is also desired. Therefore, we will select those components that are estimated to be more important than the remaining components. Changing the parameters in these important components is expected to have a large impact on the final simulated energy. Thus, a lower simulated ground state energy, which will be closer to the true ground state energy, is more likely to be achieved. For a given compression ratio α, if the total number of parameters in the original UCCSD is K, then we will select the top $\lceil \alpha K \rceil$ parameters and employ their corresponding Pauli string simulation circuits. **Third**, we will make the constructed ansatz hardware friendly by putting the Pauli strings in an *importance-decreasing* order. Such an order will reduce the overhead when mapping to the target hardware by the compiler because this approach can improve qubit locality in the generated ansatz as explained in the next paragraph.

Improving locality: The term qubit locality (similar to data locality in classical computing) in this chapter is that the CNOT gates are applied more frequently on some logical qubits in a period of time. In quantum chemistry simulation, each qubit represents an orbital but the wavefunction of the electrons is not uniformly distributed on all orbitals. Different orbitals represent the states with different energies and the electrons are more likely to occupy low energy orbitals because the energy minimum represents a stable ground state. Therefore, those Pauli string simulation circuits that involve low-energy orbitals are more important because changing their parameters will affect the occupancy of the low-energy orbitals. In our ansatz construction, those Pauli string simulation circuits at the beginning of the program mostly include the qubits representing low-energy orbitals. And these qubits will be frequently involved in the CNOT gates in these Pauli string simulation circuits. This creates gate locality in our constructed

ansatz, which makes it easier to be synthesized and mapped later in our compilation.

The output of our ansatz compression algorithm is a sequence of Pauli strings and their parameters, rather than a typical quantum circuit. Later in Section 5.5, we will have a customized compilation flow to compile the Pauli string sequence into an executable quantum circuit.

5.4 Architecture Design

After a chemistry simulation program is compressed, a quantum hardware platform is required to finally execute the optimized program. In this section, we propose a new superconducting quantum processor architecture to efficiently support variational quantum chemistry simulation. We first detail the design objectives and physical constraints. Then we introduce a new hardware architecture, namely *X-Tree*, and discuss the reasons why it can support VQE circuits with both high performance and high efficiency.

Design objectives: This architecture should support VQE programs with **high performance**, which means that the simulation programs can be synthesized into circuits and then mapped onto the proposed architecture with low overhead (i.e., no or few additional SWAP gates). It should have as **few connections** as possible because more connections will increase the probability of frequency collision, lower the yield rate, and also increase crosstalk error. The architecture should have **good programmability**, which means it can support programs from the entire UCCSD simulation family for various target chemistry systems.

Device modeling and physical constraints: We adopt IBM's fixed-frequency transmon qubit and cross-resonance qubit connections [111]. The following practical physical constraints are considered. Physical qubits are placed on a planar substrate. One physical qubit can only connect to a limited number of nearby physical qubits

directly via bus resonators. In this work we allow one qubit to connect to at most four neighbors to increase device reliability, but similar architectures with five or six direct connections per qubit have also been built [169].

5.4.1 X-Tree Architecture

As introduced in Section 5.2.1, the CNOT gates in the Pauli string simulation circuits form a tree structure. Therefore, if the physical qubits are connected in a tree, we can match them to the CNOT gates in the chemistry simulation program. Based on this observation, we propose an X-Tree superconducting quantum processor architecture after considering the design objective and physical constraints mentioned above.

X-Tree architecture construction: An X-Tree architecture starts from a *root* qubit. Then more qubits are placed and connected. The key is that the coupling graph formed by the connection is always a tree and there is no circle in the connections. Figure 5.6 shows several examples of X-Tree architecture with different numbers of qubits. We may connect four qubits to the root qubit and obtain the XTree5Q (5-qubit) architecture. We can add three more qubits to one leaf qubit of XTree5Q to obtain XTree8Q. Similarly, we can have XTree17Q and XTree26Q architectures by adding more physical qubits. The first generation of IBM's cloud-access quantum computers were compatible with the XTree5Q architecture, but they have since diverged. Next, we explain why X-Tree architecture can satisfy our three design objectives.

Fewer connections: The proposed X-Tree architecture is highly simplified and has only the smallest number of connections ($N-1$ connections for N qubits) to connect all qubits because the coupling graph of an X-Tree architecture is a tree. As our device yield rate simulations will show, this judicious lowering of connections results in higher *yield rate* in this architecture compared to conventional 2D-grid architectures (which roughly

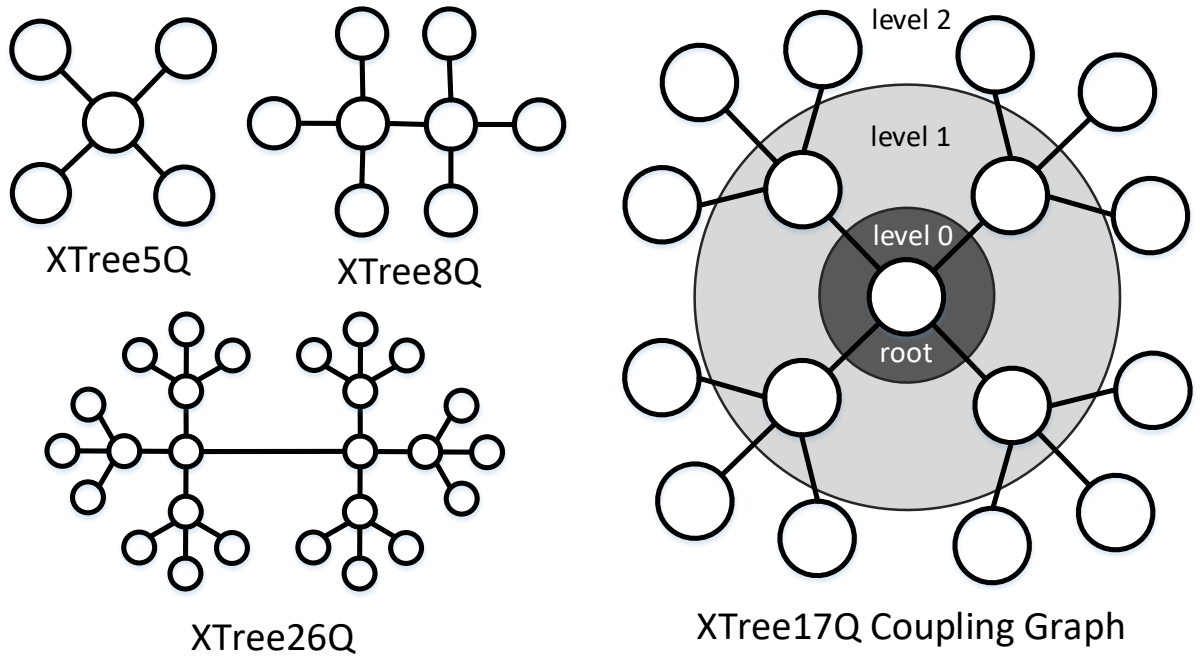


Figure 5.6: X-Tree architecture examples

have $2N$ connections for N qubits). Similarly, *gate crosstalk* errors will be significantly reduced too.

High performance and programmability: We expect the X-Tree architecture to support variational chemistry simulation applications with low mapping overhead since the physical qubit connections naturally fit the logical qubits' CNOT gate connections (both of them are trees). We also expect programmability since the X-Tree architecture is not tailored to specific any gate-level VQE circuit instances. Instead, our design is inspired by the properties of Pauli string, a high-level algorithm feature, without any assumptions about the simulated system. However, the physical connection tree is not identical to the CNOT gate connection trees since there are different Pauli strings on different qubits for different simulation programs. Compiler optimizations are still required to deploy the chemistry simulation program onto the X-Tree architecture, which will be explained in the next section.

5.5 Compiler Optimization

Although the X-Tree architecture has been designed to match the tree pattern of gates in a typical quantum chemistry program, we will show that state-of-the-art compilers are not well suited for taking maximum advantage of it. A traditional quantum compilation flow separates high-level synthesis from mapping onto the architecture. That is, it will first convert the Pauli strings and the parameters into concrete Pauli string simulation circuits using a uniform CNOT synthesis plan. For example, Qiskit [159] synthesizes the CNOTs in a Pauli string simulation circuit in a chain structure like Figure 5.2 (b). However, recall that there is great flexibility in how each Pauli string simulation circuit is synthesized: as long as the non-trivial qubits in the Pauli string are connected by a tree, it does not matter which connections we use. This is the key insight that allows us to adaptively synthesize and map each Pauli string simulation circuit in the larger ansatz. The approach taken by previous compilers fails to recognize this flexibility. Once the circuit is synthesized, it is exceedingly hard to find such high-level semantics, and mapping a poorly synthesized program on a sparse architecture can incur a very high cost.

In this section, we introduce the third optimization, a tailored compiler optimization to efficiently synthesize and map variational quantum chemistry simulation programs to X-Tree architectures with very low overhead. We will show that this tailored approach incurs an overhead of around 99% lower than a traditional compiler for the same architecture, and even 97.7% lower than mapping to a dense architecture but without leveraging such compiler optimizations.

Our compiler optimization performs circuit synthesis and qubit mapping collaboratively in two steps. First, we determine an initial qubit layout, based on the ansatz Pauli strings only, before the program is synthesized to gate sequences. Then we perform

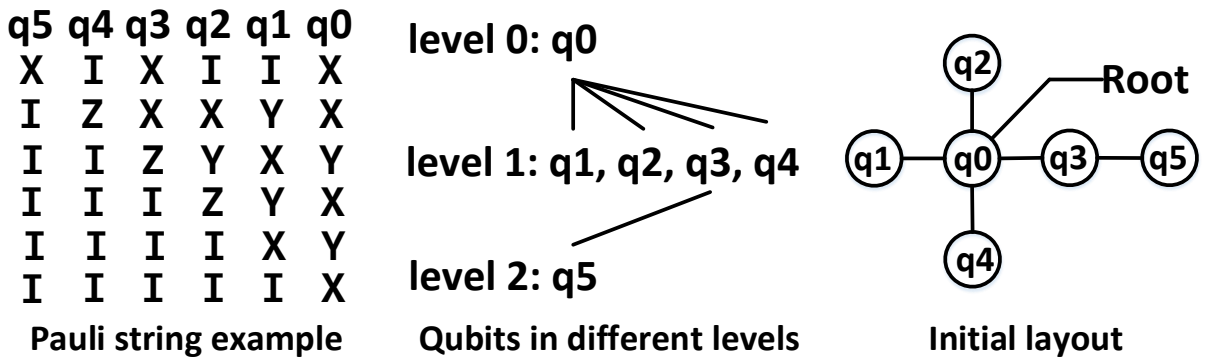


Figure 5.7: Initial layout example

circuit synthesis and qubit routing (inserting SWAPs) simultaneously onto the X-Tree architecture.

5.5.1 Hierarchical Initial Layout

Since the program is not converted to gates yet, our initial qubit layout algorithm will directly analyze the high-level program and provide an initial qubit layout. This is possible because our proposed X-Tree architecture has different physical qubit levels. For example, in the XTree17Q architecture in Figure 5.6, the center (root) qubit has level 0 as it is on average closer to all other qubits. The four qubits surrounding the root have level 1, and the leaves have level 2. Similarly, we can also discover different priorities for different logical qubits in a chemistry simulation program. The states represented by some orbitals are closer to the true ground state of the electrons, thus the logical qubits corresponding to these orbitals will appear in more Pauli string simulation circuits and will participate in more CNOT gates (as discussed in Section 5.3.2). We place these logical qubits on lower-level physical qubits, ensuring that they can reach other qubits with shorter paths.

Our hierarchical initial layout algorithm is based on such heterogeneity of the logical and physical qubits. The pseudocode is in Algorithm 6 and we explain the algorithm

Algorithm 6: Hierarchical Initial Layout

Input: Pauli strings in the simulation program, an X-Tree architecture with qubits at different levels

Output: Initial logical-to-physical qubit mapping

for P_i *in all Pauli Strings* **do**
  if *the qubit* $\boldsymbol{j}$ *and* $\boldsymbol{k}$ *appear in* P_i **then**
    $Mat(j,k) += 1$;
  end
end
$Qubit\_occurrence = \sum_k Mat(j,k)$;
$Logical\_qubit\_order = ArgSort(Qubit\_occurrence)$;
for *qubit j in Logical_qubit_order* **do**
  Map qubit j to the physical qubit in the lowest available level;
  if *there are multiple possible parent qubit k* **then**
    select k = argmax(Mat(j,k))
  end
end

with the example in Figure 5.7. We first determine which qubits appear in more Pauli strings. A matrix will record the number of instances when qubit i and j appear in the same Pauli string (the first loop). Then we can know which qubits connect to other qubits more by taking summation in one dimension. Finally, we sort the logical qubits by their connectivity requirements and place them on the X-Tree architecture from level 0 outwards. In Figure 5.7, we put q0, which appears in all Pauli strings, on the level 0 root and put q1, q2, q3, q4 in the four level 1 physical qubits. In case of multiple available spots, we attach to a parent qubit which shares the largest number of common Pauli strings with the logical qubit to be allocated (the parent is already allocated a physical spot because it is in a lower level). In the example in Figure 5.7, q5 has been assigned level 2 as it participates in only one Pauli string. Of the qubits it shares a Pauli string with, q3 is one level up and so chosen as q5's parent.

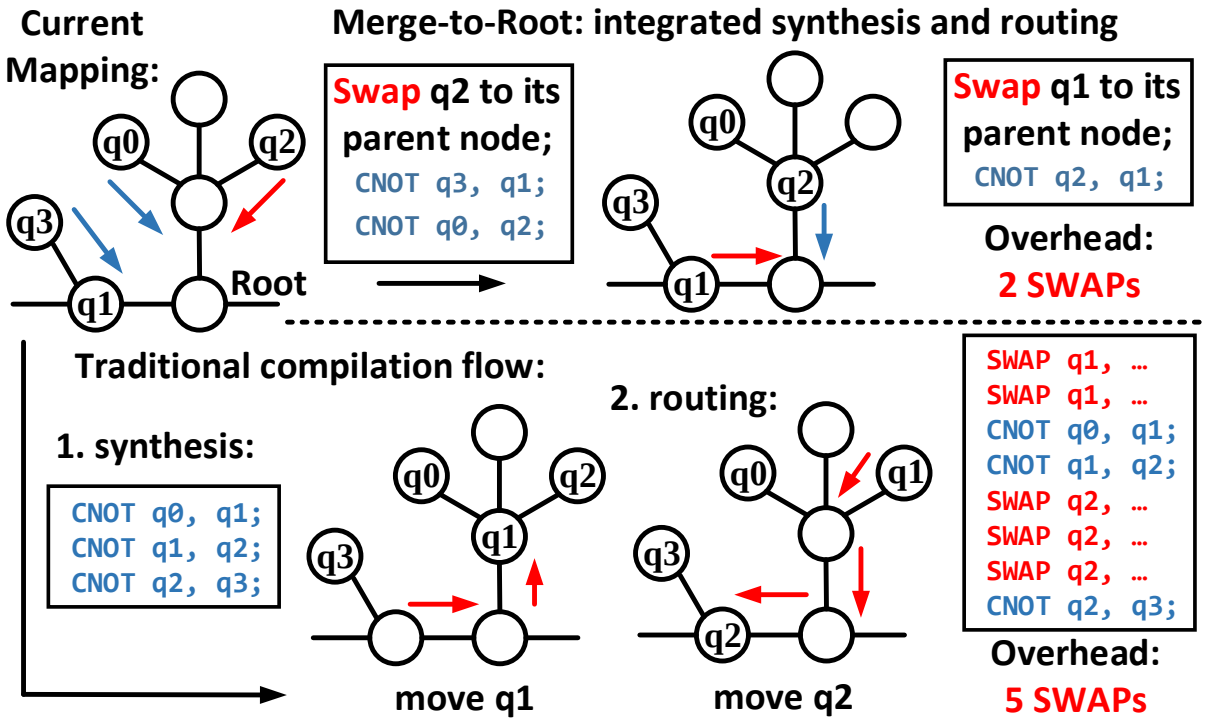


Figure 5.8: Merge-to-Root vs traditional compilation

5.5.2 Merge-to-Root Circuit Synthesis and Qubit Routing

After the initial qubit mapping is determined, we need to synthesize the Pauli string simulation circuits into concrete circuits and resolve all the CNOT gate dependency issues caused by the limited on-chip qubit connection. We propose a *Merge-to-Root* algorithm to synthesize the simulation circuits and determine how to insert SWAPs for remapping qubits. For each Pauli string simulation circuit, the two layers of the single-qubit gates at the beginning and the end are fixed. We only need to synthesize two CNOT trees and the center rotation gate as introduced in Section 5.2.1. The pseudocode is in Algorithm 7 and we explain it with the example in Figure 5.8.

Suppose we need to compile the simulation of Pauli string on four logical qubits, q0, q1, q2, and q3. Their current mapping on an X-Tree architecture is shown on the top left of Figure 5.8. Our merge-to-root compilation starts from the outermost physical

qubits. We can find that q0, q2, and q3 are currently mapped onto level 2 physical qubits. We check the parent qubits (at level 1) of these outermost qubits. If a parent qubit is holding a logical qubit in the simulation circuit, e.g., the parent qubit of q3 is the one q1 is mapped onto, then we can synthesize a CNOT between these two qubits. If not, we will find one qubit in the current level and swap it to this parent qubit. For example, the parent qubit of q0 and q2 is not in the Pauli string. We first select one of them and SWAP it to the parent qubit. We will select the qubit that will appear more times in the follow-up Pauli strings. Suppose we move q2 to the parent physical qubit. We can now synthesize a CNOT between q0 and q2. The procedure above synthesizes all CNOTs that are between level 2 and 1 with just one SWAP overhead. It will be repeated from the outer levels to the inner levels until the last qubit. For level 1 qubits (q1 and q2 in this example), we can move q1 and then synthesize the last CNOT between q2 and q1. This synthesis of the left CNOT tree is now completed with only two SWAPs in total. The center rotation gate can then be applied on q1. The right CNOT tree can be synthesized similarly in a reversed order from the inner levels to the outer levels. The time complexity of our compiler optimization algorithm is $O(n\#(P_a))$ where n is the number of qubits and $\#(P_a)$ is the number of Pauli strings in the ansatz.

Comparing with traditional compilation: The lower half of Figure 5.8 also shows the compilation results of the left CNOT tree from traditional compilation flow. The left CNOT tree will first be synthesized into three CNOT gates. Then a mapping algorithm will try to move the qubits to satisfy the dependencies of the three CNOT gates. In this example, we first move q1 by two SWAP gates to execute the first two CNOT gates. We then move q2 by three SWAP gates to execute the last CNOT gate. The total overhead is five SWAPs, which is much higher than that of our Merge-to-Root compilation. The key is that, comparing with traditional compilation, Merge-to-Root will synthesize entirely different CNOTs adapted to the current mapping and the architecture.

Algorithm 7: Merge-to-Root Synthesis and Routing

Input: Initial qubit layout, Pauli strings in the simulation program, a X-Tree architecture with qubits of K different levels

Output: A hardware compatible circuit

```
for P_i in all Pauli Strings do
    // Synthesize left CNOT tree
    for level k from K − 1 down to 1 do
        if a qubit at level k is in P_i but its parent qubit q_p at level k − 1 is not in P_i then
            select one of q_p's child qubits that are in P_i and SWAP it with q_p;
        end
        Synthesize all CNOTs from level k to k − 1;
    end
end
Apply the center rotation gate on the last qubit;
Synthesize the right CNOT tree accordingly;
```

5.6 Evaluation

We evaluate the proposed co-optimization with carefully designed experiments over a wide range of chemistry simulation benchmarks to show the improvements from the algorithm, hardware, and compiler levels.

5.6.1 Experiment Setup

Benchmarks: We select nine molecules of various sizes and geometrical structures. The names of the molecules and the information of their simulation circuits using the original full UCCSD ansatz are listed in Table 5.1. Note that '# of Pauli' means the number of Pauli strings.

Metric: The simulation accuracy is measured by the simulated ground state energy of the target molecule. We adopt atomic units that are more convenient for computational chemistry. The energy unit is Hartree (1 Hartree $\approx 4.36 \times 10^{-18}$ Joules). The bond length unit is Angstrom (1 Angstrom $= 10^{-10}$ meter). The convergence speed is indicated by the

Table 5.1: Benchmark molecules and their original cost

	# of Qubits	# of Pauli	# of Param.	# of Gates (CNOTs)
H_2	4	12	3	150 (56)
LiH	6	40	8	610 (280)
NaH	8	84	15	1476 (768)
HF	10	144	24	2856 (1616)
BeH_2	12	640	92	13704 (8064)
H_2O	12	640	92	13704 (8064)
BH_3	14	1488	204	34280 (21072)
NH_3	14	1488	204	34280 (21072)
CH_4	16	2688	360	66312 (42368)

number of iterations in the parameter optimization (outer loop in Figure 5.3). A smaller number of iterations means that the simulation converges faster. Compiler optimizations are evaluated by the gate count in the post-compilation circuit, a widely used metric in previous studies [77, 56, 18]. A more effective compiler optimization will result in a lower gate count in the post-compilation circuit. The CNOT count is of particular interest owing to the much higher error rate and longer latency compared to single-qubit gates.

Implementation: We implement the proposed optimizations based on Qiskit [159] and perform experiments with classical simulators in Qiskit. The Hamiltonian of the simulated molecule is generated by PySCF [165] with STO-3G orbitals [170] and Jordan-Wigner encoding [171]. We freeze the core electrons and only simulate the interaction of the outermost electrons. We use the default UCCSD ansatz from Qiskit Aqua library (version 0.8.0). The parameters are optimized using the Sequential Least Squares Programming [172] solver. The noise-free simulations are performed with Qiskit Aer statevector simulator and the noisy simulations are performed with Qiksit Aer qasm simulator (version 0.6.0). For the hardware yield rate, we adopt the yield simulation method and qubit frequency allocation algorithm in [19]. All experiments are performed on a MacBook Pro with 2.8 GHz Quad-Core Intel Core i7 CPU and 16GB 2133MHz

LPDDR3 memory.

5.6.2 Experiment Methodology

Baseline: The software baseline is the original UCCSD ansatz [48], denoted by 'Orig. UCCSD'. The true ground state energies for reference, denoted by 'Ground State', are obtained by directly calculating the eigenvalue of the Hamiltonian of the target system. The hardware baseline is IBM's 17-qubit device (Grid17Q) with a 2D grid connection [111] (shown on the left of Figure 5.11) for a fair comparison with our 17-qubit X-Tree device (XTree17Q) employing the same number of qubits. The compiler baseline is SABRE [18] (SAB), a state-of-the-art general-purpose mapping algorithm in Qiskit.

Configurations: We apply the parameter compression method in Section 5.3 with five compression ratios: 10%, 30%, 50%, 70%, 90%. They are denoted by '10% Param.' to '90% Param.' We also generated ansatzes by randomly selecting 50% parameters (denoted by 'Rand. 50%').

5.6.3 Simulation Accuracy and Convergence Speedup

Figure 5.9 shows the simulation accuracy and the convergence speed of our compressed ansatz. The results of H_2 is omitted since its circuit is small with only three parameters. There are three parts in Figure 5.9. The X-axes represent the bond lengths of the simulated molecules. The Y-axes represent the simulated energy, simulated energy difference, and the number of iterations steps as labeled on the left of Figure 5.9. The top part shows simulated energies at different bond lengths. The simulation results of the compressed ansatzes are close to that of the full UCCSD and the true ground states. The more parameters we keep, the more accurate simulation we can obtain. To better

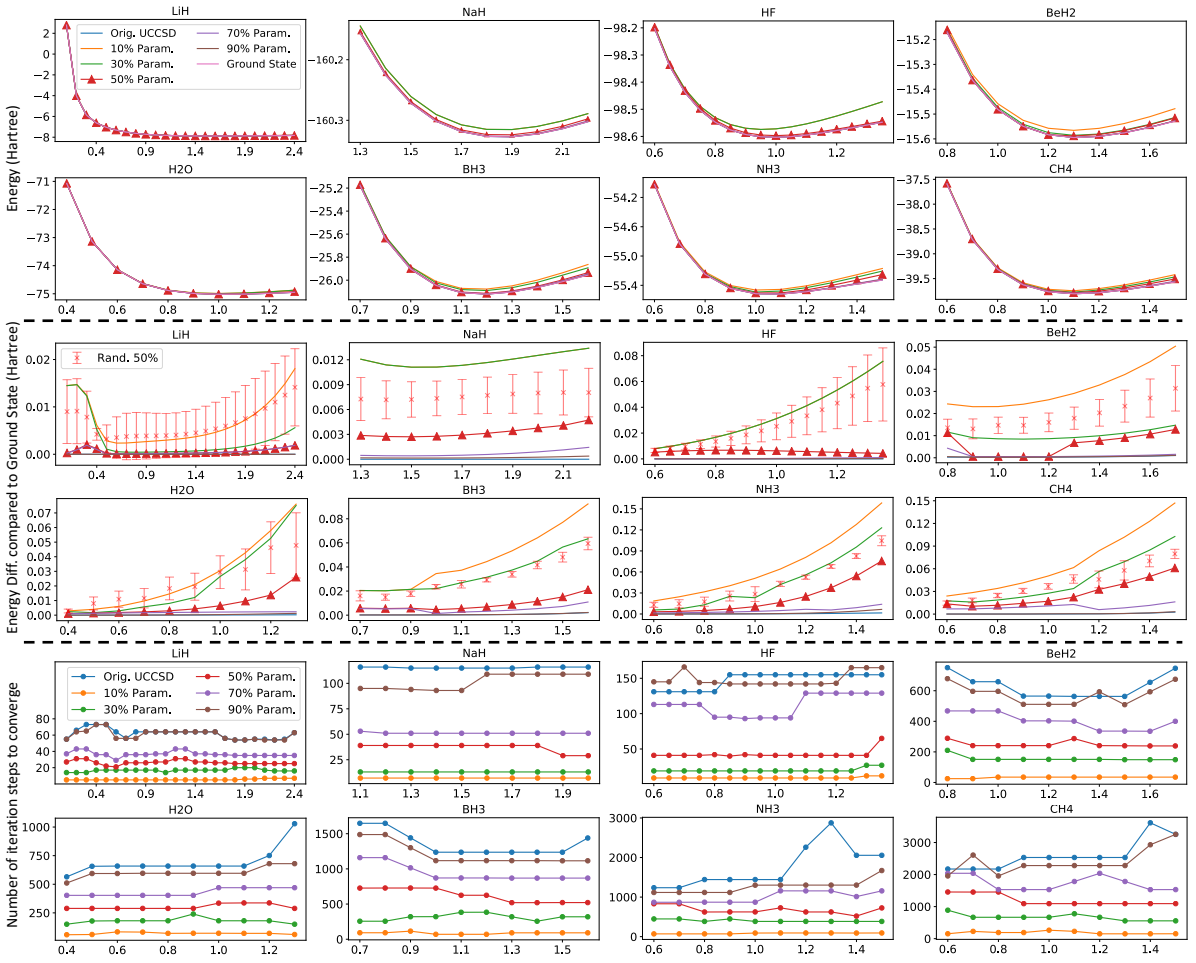


Figure 5.9: Accuracy and number of iterations vs various parameter reduction ratios

understand the amount of accuracy loss, the middle part of Figure 5.9 shows the energy difference between different experiment configurations and their corresponding true ground states. For example, the energy differences for '50% Param.' are usually only at the level of about 0.05%.

Effective parameter selection: We show the effectiveness of our parameter selection method by comparing the ansatzes generated by our compression method with those constructed by randomly selected parameters. For the ansatzes with 50% randomly selected parameters, we generate five different random parameter selections for each molecule at each simulated bond length. The simulation result distribution of

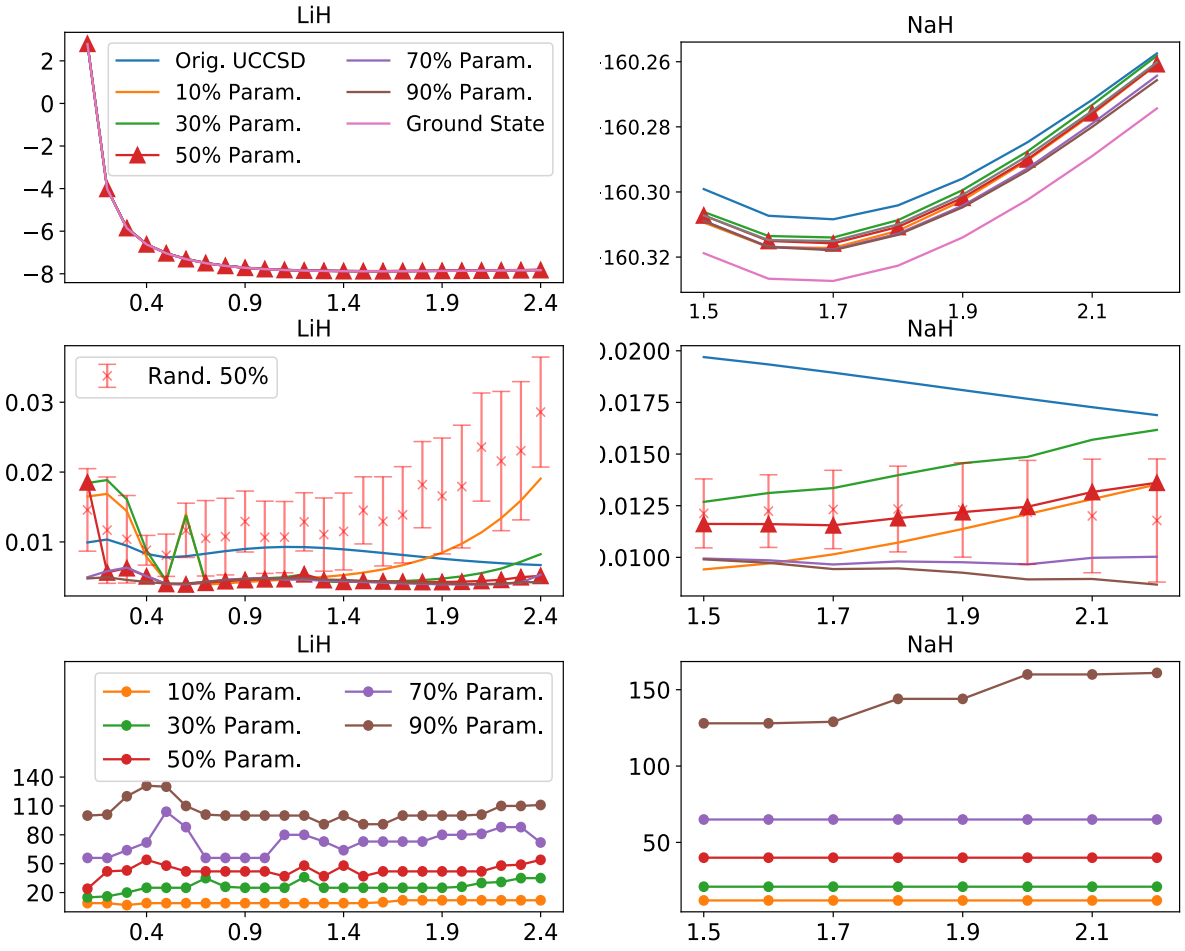


Figure 5.10: Noisy simulation case studies on LiH and NaH

'Rand. 50%' is demonstrated by the mean and standard deviation of the simulated energies. It can be observed that the '50% Param.' ansatzes generated by our optimization outperform the 'Rand. 50%' with better accuracy and the simulated energies are closer to the true ground state energies. The accuracy of 'Rand. 50%' is similar to that of '30% Param.', which means that our optimization can select 30% of parameters but achieve the same level of accuracy from randomly selecting 50% of the parameters. This comparison proves that our ansatz compression algorithm is very effective. The execution time of our ansatz compression is negligible compared with the VQE execution itself. For example, it requires several minutes to compress the ansatz for CH_4 while it takes over ten CPU

Table 5.2: Mapping overhead comparison of different compilation approaches

	Original # of CNOTs					MtR on XTree17Q (# of CNOTs)					SAB on XTree17Q (# of CNOTs)					SAB on Grid17Q (# of CNOTs)				
Ratio	10%	30%	50%	70%	90%	10%	30%	50%	70%	90%	10%	30%	50%	70%	90%	10%	30%	50%	70%	90%
H_2	48	48	52	56	56	0	0	0	6	6	0	0	0	0	0	0	0	0	0	0
LiH	80	208	256	272	280	0	6	6	12	18	48	126	132	150	168	0	6	9	15	18
NaH	176	448	672	736	764	0	0	0	3	21	162	777	1002	1197	1470	12	12	87	120	123
HF	400	912	1264	1552	1608	0	0	0	6	36	633	1863	2034	2163	2502	87	126	267	372	612
BeH_2	1504	3808	5696	7248	7984	3	6	24	51	228	3315	6513	13416	14268	17862	621	1395	4005	5253	8091
H_2O	1536	3840	5712	7280	7988	0	12	18	75	135	3132	7764	12495	13266	15618	1110	1725	2034	2514	3156
BH_3	3664	9632	14560	18368	20824	0	39	108	237	606	9489	23811	35289	45603	46395	2163	7632	9654	17010	21165
NH_3	3680	9696	14592	18480	20824	0	30	72	183	522	11646	20622	35523	42348	48447	1959	5844	8568	12375	13668
CH_4	7136	19040	28992	36656	41632	0	45	120	366	1005	23796	56799	79821	99831	111876	4788	18939	25173	33792	39729

hours to simulate CH_4 with VQE at one bond length.

Convergence speedup: The bottom part of Figure 5.9 shows the number of iterations to converge. The compressed ansatzes with fewer parameters can converge much faster with smaller numbers of parameter optimization steps. The numbers of parameter optimization steps are reduced by 14.3×, 4.8×, 2.5×, 1.6×, and 1.1× on average for the five parameter compression ratios of 10% to 90%, respectively.

There is also a subtle implication in computation reliability when the computation concludes faster. Quantum computers are calibrated to reduce gate errors. After a few hours, the physical properties of the system drift causing the calibration to become stale. At current experimental speeds, a full VQE experiment can easily take hours to converge, which makes these speedups a boost to reliability as well.

5.6.4 Noisy Simulation Case Studies

We study the effect of hardware noise on LiH and NaH as case studies. Our simulation adopts a depolarizing error model with realistic CNOT error rates of 0.0001 [173]. Figure 5.10 shows the simulation results. Similar to Figure 5.9, the first, second, and third rows are the overall simulated energies, energy differences, and number of iterations, respectively. We observe that our compressed VQE can still demonstrate the correct landscapes of the molecule energy under different bond lengths. We can also observe interesting trade-offs between parameter pruning and accuracy in noisy regimes. For LiH,

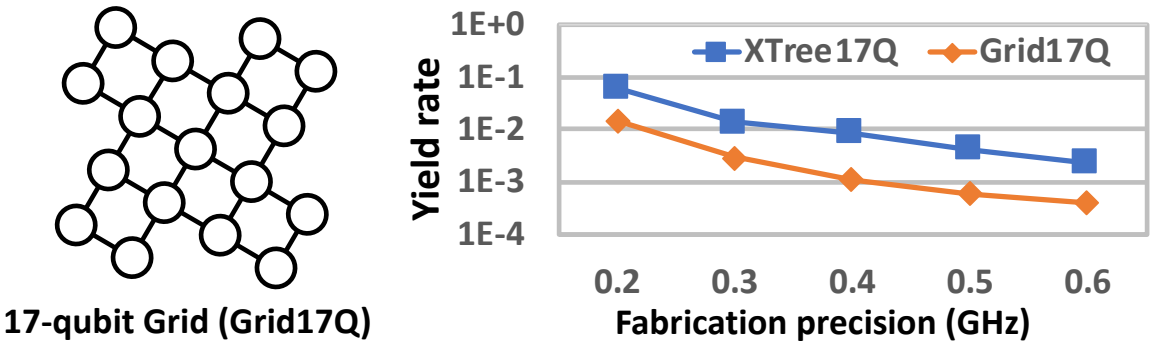


Figure 5.11: Grid17Q architecture and the yield rate comparison

the error first decreases from '10% Param.' to '50% Param.' due to the increasing parameters. After that, the error does not change significantly from '50% Param.' to '90% Param.' because the effect of more parameters is masked by the increasing gate error. '50% Param.' is a sweet spot for LiH. For NaH, the balance is different. The error first increases from '10% Param.' to '30% Param.' and then drops from '30% Param.' to '90% Param.' This suggests that we should either select '10% Param.' or '90% Param'. Such trade-offs depend on the molecule Hamiltonian, the bond length configuration, hardware noise strength, and maybe other factors. A comprehensive research into these trade-offs is left as future work.

5.6.5 Hardware Efficiency

We evaluate our hardware design by comparing the XTree17Q architecture with baseline Grid17Q. Both of them have 17 physical qubits. Figure 5.11 shows the yield rates of XTree17Q and Grid17Q for various fabrication precision parameters from 0.2 GHz to 0.6 GHz. The yield rate of the XTree17Q architecture is about 8× higher than that of the Grid17Q architecture. This is because Grid17Q has 24 connections while XTree17Q only employs 16 connections.

5.6.6 Mapping Overhead Reduction

Table 5.2 shows the mapping overhead (i.e., the number of additional CNOT gates) of our Merge-to-Root (MtR) compilation (including our initial layout algorithm) vs. the baseline compilation (SAB) on XTree17Q and Grid17Q architectures.

We first compare MtR on XTree17Q vs. SAB on XTree17Q. The sparse connectivity of XTree17Q makes the mapping overhead very high for the general-purpose SAB compiler. The number of additional CNOTs is about 177% of the CNOT count of the original circuits. This is even worse for larger benchmarks. For CH_4, the number of additional CNOTs for SAB is about 288% of the original CNOT count. However, our MtR compilation incurs dramatically smaller overhead. For all tested benchmarks, the number of additional CNOTs is on average 1.4% of the original CNOT count. Therefore, our MtR compilation reduces the mapping overhead to only about **1**% of the overhead from the state-of-the-art compilation.

The SAB compiler still cannot compete with our co-designed approach even if it targets a much denser architecture. Grid17Q employs more connections, of course at the cost of 8× lower yield rate compared to our XTree17Q. However, even then the CNOT overhead for MtR on XTree17Q is only about 2.3% of SAB on Grid17Q in most cases.

Locality improvement: Analysis of mapping overheads shows that our ansatz construction improves gate locality. At 10% ansatz compression ratio, MtR on XTree17Q does not require any additional CNOTs most of the time. We also observe that this mapping overhead jumps much faster from 70% to 90% compared to other gaps. For example, the mapping overhead increases from 70% to 90% is about 2.9× that of 50% to 70%, while the original CNOT count increment from 70% to 90% is only about 0.47× that of 50% to 70%. This is because our ansatz construction will first select Pauli string simulation circuits with more gate locality so that they can be synthesized and mapped

to XTree17Q efficiently. But at compression ratios close to 1 (i.e. little compression), Pauli string simulation circuits with poor locality will also be included in the ansatz, which makes the mapping overhead grow faster.

5.7 Discussion and Future Directions

In this chapter, we advance variational quantum computational chemistry through a holistic software-hardware co-optimization from the algorithm, compiler, and hardware levels, outperforming conventional setups with significant benefits of multiple aspects. This is the first attempt, to the best of our knowledge, that leverages the high-level application domain knowledge to coordinate the optimizations throughout the three levels from software to hardware in quantum computing. Also our software-hardware co-optimization is not targeting a particular program instance and can broadly accommodate the full family of computational chemistry problems with such structure. We believe that the co-optimization principle can also be applied to other promising application domains and hardware implementation technologies to boost the development of quantum computing. Several further research directions are briefly discussed as follows:

More physical systems: This chapter focused on chemical systems and the results can guide the development of new useful compounds. Many other physical systems are also worth simulating. For example, the Hubbard model [174] in condensed matter physics can explain the transition between conducting and insulating systems. These models may have different characteristics compared to a chemical system, e.g., periodic potential vs atomic potential, fermion vs boson. We expect that the Pauli-string-centric principle will still be applicable since the mathematics about simulating a Hamiltonian is invariant. But the actual optimizations may need to change according to the characteristics of these models.

Hardware architecture variants: This work focuses on the tree architecture with a minimized number of connections for a higher yield rate. However, it is not yet known how to find other Pareto-optimal designs. We may also need to change the number of connections per qubit when scaling up and to improve CNOT fidelity. It can be interesting to consider tree structures with different degrees at different levels. Moreover, for other hardware like ion traps, the main constraints can be different, and it is worth exploring how to extend the Pauli-string-centric principle to optimize quantum computational chemistry on other platforms.

Deeper compiler optimization: The compiler optimization in this chapter is for the circuit synthesis and qubit mapping passes, which are essential in compiling a program to an executable circuit on a superconducting quantum processor. Deeper compiler optimization is possible in at least two directions. First, other passes in the traditional compilation flow, e.g., gate cancellation [163], may be customized to variational quantum chemistry simulation programs. Second, the variational quantum simulation is a numerical optimization algorithm. It is thus possible to allow approximate compilation for more aggressive compiler optimization. Third, compiler-based error mitigation techniques [175, 176, 177] can also be incorporated to further reduce the simulation error.

5.8 Related Work

The techniques in this chapter range across the algorithm, hardware, and compiler for variational quantum computational chemistry. We briefly introduce related work for each of them.

5.8.1 Algorithmic Optimization

The major component in the VQE circuit is the parameterized ansatz. UCCSD [48] is the "standard" chemistry-inspired ansatz, but has a large size. There have been several optimizations to reduce its size [148, 149, 150, 151, 152, 153], but without considering specific hardware mapping overheads. At the other extreme, "hardware-efficient" ansatzes [47] have been proposed which only employ gates that are easy to implement on the underlying hardware. However, these ansatzes are unlikely to support large molecules since they do not consider any information about the chemical system to be simulated [154, 5]. Alternatively, several ansatz selection techniques rely on classical simulation of the molecule, and it is unclear how they scale to super-classical regimes [146, 178]. In contrast, the algorithm optimization proposed in this work exploits information about the target system through Pauli string comparison, and can maintain simulation accuracy as well as reduce hardware mapping overhead, and does not require classical simulation.

Additionally, there is prior work on optimizing the number of measurements required to evaluate the energy [179, 180, 181, 182, 183]. This type of optimization reduces the number of iterations of the inner loop in Figure 5.3 and is orthogonal to our techniques which reduce the number of iterations in the outer loop as well as the size of the circuit itself. These optimizations can be employed together with our techniques.

5.8.2 Compiler Optimization

A large body of work exists on mapping quantum circuits to hardware [104, 18, 77, 85, 184]. These algorithms are invoked after a quantum circuit is already synthesized and are general-purpose with little assumption regarding the input programs or the underlying hardware architectures.

High-level semantics have recently been considered in compiler optimizations. Cowtan

et al. recently proposed a method for compiling UCC ansatzes by partitioning Pauli strings into sets, but not considering the underlying architecture [185]. An architecture-aware synthesis for phase-polynomial quantum circuits was proposed in [186].

In contrast, this chapter uses the Pauli string simulation circuit to devise a new compilation flow based not only on the chemistry simulation domain knowledge but also on the underlying architecture. Starting from a Pauli string IR, it achieves unprecedented mapping overhead reduction by combining synthesis and mapping in a single pass.

5.8.3 Application-specific Quantum Processor Architecture

An application-specific quantum architecture was proposed by Wilhelm et al. for a specific Fermi-Hubbard model simulation, based on a superconducting planar architecture [123, 124]. Recently, an end-to-end design flow has also been proposed to generate optimized superconducting quantum processor architectures for different individual quantum programs [19]. These architectures are circuit-specific rather than domain-specific, as they exploit low-level gate patterns but not high-level domain knowledge and do not generalize to families of circuits. For trapped ion technology, [187] provided a forward-looking overview of co-designing trapped ion machines. Murali et al. also proposed a toolflow to evaluate the architecture design of trapped ion quantum computers over a benchmark suite [188]. Our architecture design, which integrates the algorithm-level domain knowledge, is a concrete optimized design with compiler support to accommodate various variational quantum chemistry programs with different simulation targets.

5.9 Conclusion

In this chapter, we advance variational quantum chemistry simulation through a holistic software-hardware co-optimization at the algorithm, compiler, and hardware levels.

We show that variational quantum chemistry programs can be significantly simplified without complex derivative calculation, and they can be efficiently mapped onto a high yield superconducting quantum processor with very sparse connections. The three proposed optimizations can accommodate simulating various chemical systems and bring a wide range of advantages from software to hardware. The design principle and the results from this chapter could guide future development of quantum software and hardware infrastructures.

Chapter 6

Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels

6.1 Introduction

In the previous chapter we introduce the software-hardware co-design for quantum chemistry simulation. Actually, quantum simulation is much more than chemistry simulation. It is can be generalized to many other domains. one of the most important quantum algorithm design principles and can be generalized to many other domains. Simulating a quantum physical system, including the chemical system and other quantum physics systems of interest, which motivated Feynman's proposal to build a quantum computer [2], is by itself an important application of quantum computing [189, 190]. Later, the idea of quantum simulation was extended to quantum algorithms for other

applications, e.g., linear systems [191], quantum principal component analysis [192], and quantum support vector machine [193]. These algorithms involve simulating an artificial quantum system crafted based on the target problem. In recently developed variational quantum algorithms for near-term quantum computers (e.g., VQE for chemistry [48] and QAOA for combinatorial optimization [6]), the program structures are also inspired by the simulation principle.

Because the quantum simulation principle is shared among many algorithms, one subroutine, which we term the *quantum simulation kernel* in this chapter, appears frequently in quantum programs. This kernel is to implement the operator (controlled-)exp(iHt) where H is the Hamiltonian of the simulated system and $t \in \mathbb{R}$ is system evolution time. Since it is hard in general to directly compile exp(iHt) into executable single- and two-qubit gates, a compiler usually decomposes H into the sum of local Hamiltonians [189] (simulation of which can be easily compiled to basic gates) and then synthesize them one-by-one. Consequently, the quantum simulation kernel will be compiled to a very long gate sequence and constitute the vast majority of cost in post-compilation quantum programs.

Optimizing the compilation of this kernel can immediately benefit a wide range of quantum applications. However, three key challenges have so far hindered deeper compiler optimizations for quantum simulation kernels.

First, existing quantum compilers (e.g., Qiskit [159], Quilc [194], t|ket⟩ [161]) lack a good formal high-level intermediate representation (IR). Once programs are converted to low-level gate sequences, the high-level semantics of quantum simulation kernels are lost and hard to reconstruct from assembly-style gate sequences. Moreover, simulation kernels face different constraints in different algorithms. Previous ad-hoc optimizations of quantum simulation [195, 196, 197, 198, 185, 186, 199, 103, 200, 201, 20, 202] are mostly algorithm-specific and do not generalize due to the lack of a formal IR that can

uniformly represent simulations kernels as well as varying constraints that are attached to them in different algorithms.

Second, most optimizations (e.g., circuit rewriting [162], gate cancellation [163], template matching [164], qubit mapping [104]) in today's quantum compilers [159, 194] are local program transformations at small scale. However, these passes are designed for generic input program and fail to leverage the deeper optimization opportunities present in quantum simulation kernels. These opportunities are mainly from the properties of Pauli strings which naturally appear in the (Suzuki-)Trotter Hamiltonian approximation [189, 203], Jordan-Wigner [204] or Bravyi-Kitaev [205] fermion-to-qubit transformation, etc.

Third, quantum simulation kernels appear in a wide range of algorithms. Some algorithms [189, 190, 191, 192, 193] are designed for fault-tolerant quantum computers with quantum error correction while others [6, 48] target near-term noisy quantum computers. The hardware models of these backends can be very different and one single optimization pass may not be suitable for all of them. Adapting the high-level algorithmic optimizations to the various (and ever-evolving) hardware platforms with different constraints and optimization objectives naturally invokes a reconfigurable compiler infrastructure.

To overcome these challenges, we propose Paulihedral, a compiler framework backed by a formal IR to deeply optimize quantum simulation kernels. A brief comparison between Paulihedral and conventional quantum compilers is shown in Figure 6.1. **First**, Paulihedral comes with a new IR, namely Pauli IR, to represent the quantum simulation kernels at the Pauli string level rather than the gate level. The syntax of Pauli IR has a novel block structure which can uniformly represent the simulation kernels of different forms and constraints. The semantics of Pauli IR is defined on the commutative matrix addition operation. Such semantics guarantees that the follow-up high-level algorithmic optimizations are always semantics-preserving and can be safely applied. **Second**, we

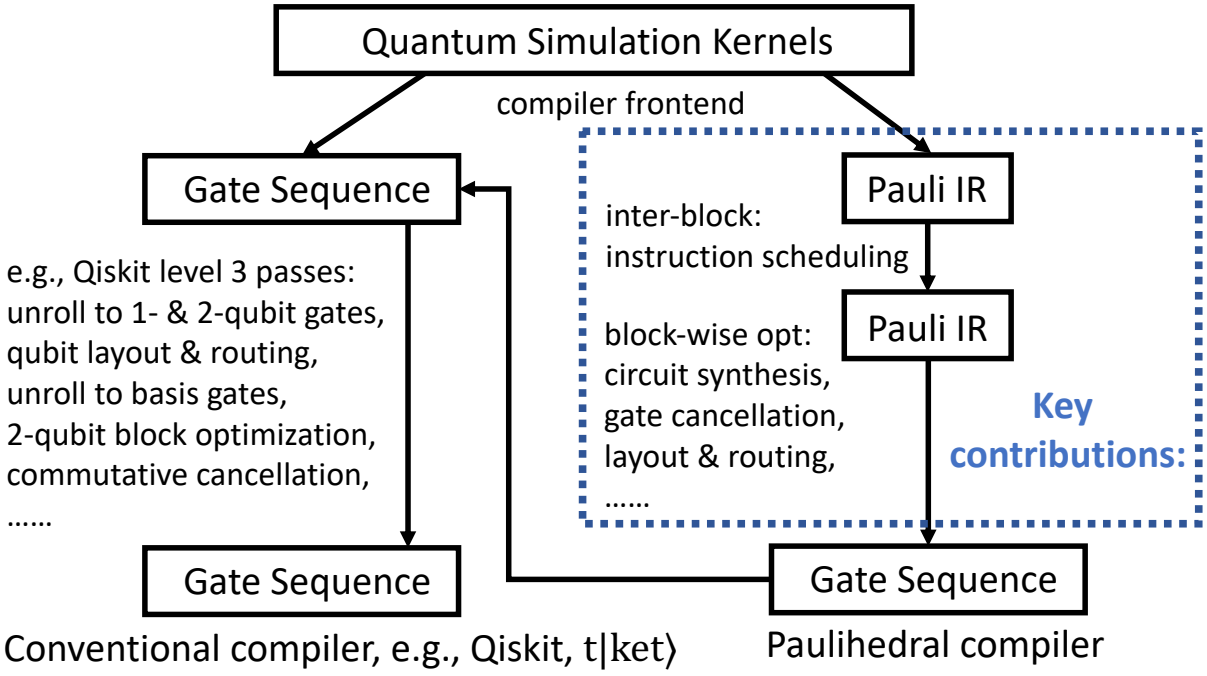


Figure 6.1: Paulihedral vs conventional compilers

propose several novel optimization passes to reconcile instruction scheduling, circuit synthesis, gate cancellation, and qubit layout/routing at the Pauli IR level. All these passes are much more effective than their counterparts in conventional gate-based compilers because they are operating in a large scope where the algorithmic properties of Pauli strings (quantum simulation kernel) are fully exploited. The optimization algorithms in these passes are also highly scalable since analyzing and processing Pauli strings are much easier than handling the gate matrices on a classical computer. **Third**, we decouple the technology-independent and technology-dependent optimizations at different stages and Paulihedral can be extended to different backends by adding/modifying the technology-dependent passes. To showcase, we develop technology-dependent optimizations for two different backends, the fault-tolerant quantum computer and the noisy near-term superconducting quantum processor.

Our comprehensive evaluations show that Paulihedral outperforms state-of-the-art

baseline compilers (Qiskit [159], t|ket⟩ [161] and algorithm-specific compilers [200, 206, 207]) with significant gate count and circuit depth reduction on both fault-tolerant and superconducting backends, and only introduces very small additional compilation time. We also perform real-system experiments to show that Paulihedral can significantly increase the end-to-end success rate of QAOA programs on IBM's superconducting quantum devices.

Our major contributions can be summarized as follows:

1. We propose Paulihedral, an extensible algorithmic compiler framework that can deeply optimize quantum simulation kernels and thus benefit the compilation of a wide range of quantum programs, with passes that make it retargetable to various backends and optimization objectives.

2. We define a new Pauli IR with formal syntax and semantics which can uniformly represent quantum simulation kernels and encode algorithmic constraints of seemingly very different algorithms, and safely expose high-level information to the compiler for optimizations.

3. We propose several compiler passes for different optimization objectives and backends. They can outperform previous works by systematically leveraging the algorithmic information and they are scalable to efficiently handle larger-size programs.

4. Our experiments on 31 different benchmarks show that Paulihedral can outperform state-of-the-art baseline compilers with significant gate count and circuit depth reduction. For example, compared with t|ket⟩ [161], Paulihedral achieves 53.1% gate count reduction and 53.3% circuit depth reduction on average on the superconducting backend, as well as 33.6% gate count and 65.0% circuit depth reduction on the fault-tolerant backend, using only $\sim 5\%$ additional compilation time. For QAOA on

a real quantum device, Paulihedral achieves end-to-end 1.24× success probability improvement on average (up to 1.87×) against the baseline Qiskit compiler [159].

6.2 Background

In the section we introduce the necessary background about quantum simulation kernels. We will first revisit the Pauli strings and then introduce the quantum algorithms related to quantum simulation.

6.2.1 Pauli String and Compilation

We start with the Pauli string, the basic concept in quantum simulation. For an n-qubit system, a Pauli string is defined as $P = \sigma_{n-1}\sigma_{n-2}\cdots\sigma_0$ where $\sigma_i \in \{I, X, Y, Z\}$, $0 \leq i \leq n-1$. X, Y, Z are the three Pauli operators, and I is the identity. σ_i corresponds to the i-th qubit. The operators in a Pauli string P can represent a Hermitian operator $\otimes_{i=0}^{n-1}\sigma_i$ ($\otimes$ is the Kronecker product), which can be denoted by P without ambiguity. In the rest of this chapter, we do not distinguish a Pauli string P and the Hermitian operator generated by P.

One important property of a Pauli string is that the operator $\exp(\mathrm{i}P\frac{\theta}{2})$ can be easily synthesized into basic gates. An example of synthesizing $\exp(\mathrm{i}Y_4Z_3I_2X_1Z_0\frac{\theta}{2})$ is shown in Figure 6.2. There are two identical layers of single-qubit gates at the beginning and the end of the synthesized circuit. In this single-qubit gate layer, there are H or Y gates on those qubits whose operators are X (i.e., $q1$) or Y (i.e., $q4$) in the Pauli string, respectively. In the middle is a left CNOT tree, a central $\mathrm{Rz}(\theta)$ gate, and a right CNOT tree. The left tree can be generated in different ways and the only requirement is to connect all the qubits whose operators are not the identity in P (e.g., $q0$, $q1$, $q3$, $q4$ in Figure 6.2). The lower half of Figure 6.2 shows three different but valid ways to generate

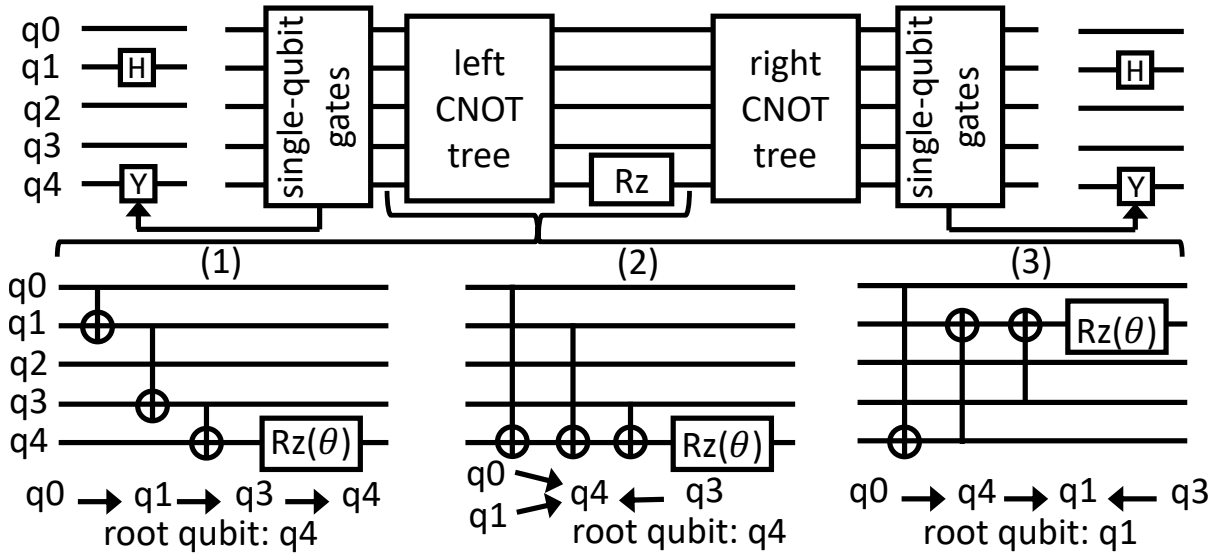


Figure 6.2: Synthesis example of $\exp(\mathrm{i}Y_4Z_3I_2X_1Z_0\frac{\theta}{2})$

the CNOT tree circuits and their corresponding tree graphs. In these trees, the CNOT gates should connect the qubits from the leaf nodes to the root node. Any qubit in the tree can become the root (e.g., $q4$ in Figure 6.2 (1) (2), $q1$ in Figure 6.2 (3)). The central $\mathrm{Rz}(\theta)$ gate is applied on the root qubit and the right CNOT tree has the same CNOT gates in the left tree but in a reversed order. Paulihedral uses this algorithmic flexibility in synthesis to increase gate cancellation and reduce the mapping overhead.

6.2.2 Quantum Simulation Kernels

The quantum simulation kernel is to (approximately) implement the operator $\exp(\mathrm{i}Ht)$ where H is the Hamiltonian of the simulated system and $t \in \mathbb{R}$. Since directly compiling $\exp(\mathrm{i}Ht)$ into single- and two-qubit gates is hard, a compiler usually expands H in the Pauli basis, i.e., $H = \sum_{j=1}^{N} w_j P_j$ where $w_j \in \mathbb{R}$ and P_j is a Pauli string. Then $\exp(\mathrm{i}Ht)$ is approximated using the Trotter formula [208]: $\exp(\mathrm{i}Ht) = \left[\prod_{j=1}^{N} \exp(\mathrm{i}P_j w_j \Delta t)\right]^{\frac{t}{\Delta t}} + O(t\Delta t)$. Δt is a parameter determined by the simulation accuracy. Figure 6.3 (a) shows the expansion process. $\exp(\mathrm{i}Ht)$ is first converted to $\frac{t}{\Delta t}$ terms of $\exp(\mathrm{i}H\Delta t)$. Each

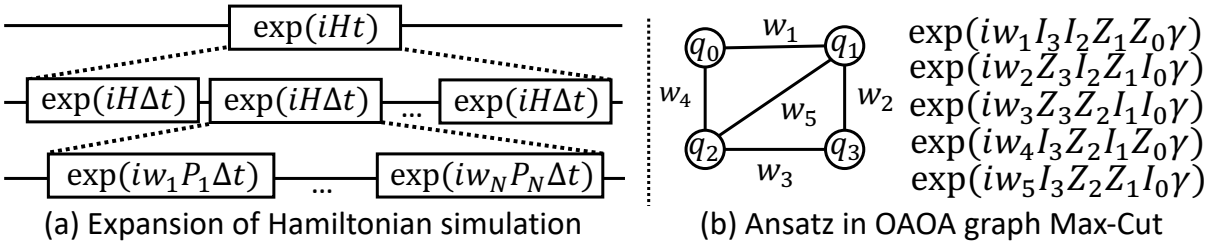


Figure 6.3: Example of quantum simulation kernels

$\exp(\mathrm{i}H\Delta t)$ is then expanded to an array of $\exp(\mathrm{i}P_j w_j \Delta t)$ and converted to basic gates.

Quantum simulation kernels also appear in recently developed variational quantum algorithms, in which the vast majority of the program is an ansatz (parameterized quantum circuit). One popular type of ansatz with good trainability is the application-inspired ansatz [209] which can be considered as a simulation kernel. Compared with implementing $\exp(\mathrm{i}H\Delta t)$, the only difference is that the Δt is changed to some tunable parameters associated with different Pauli strings and *the overall program structure remains the same*. For example, Figure 6.3 (b) shows the ansatz of QAOA algorithm [6] on a 4-node graph Max-Cut problem. The graph of the problem has 5 edges of different weights, and the Hamiltonian of this problem is the weighted sum of the 5 Pauli strings associated with the 5 edges. The majority of the QAOA ansatz [6] is to implement the 5 operators on the right (γ is the parameter).

6.3 Foundations of Paulihedral

In this section, we first introduce the opportunities and challenges of compiler optimizations for the simulation kernel. Then we formally introduce a new IR that maintains the high-level information and the algorithm constraints in Paulihedral.

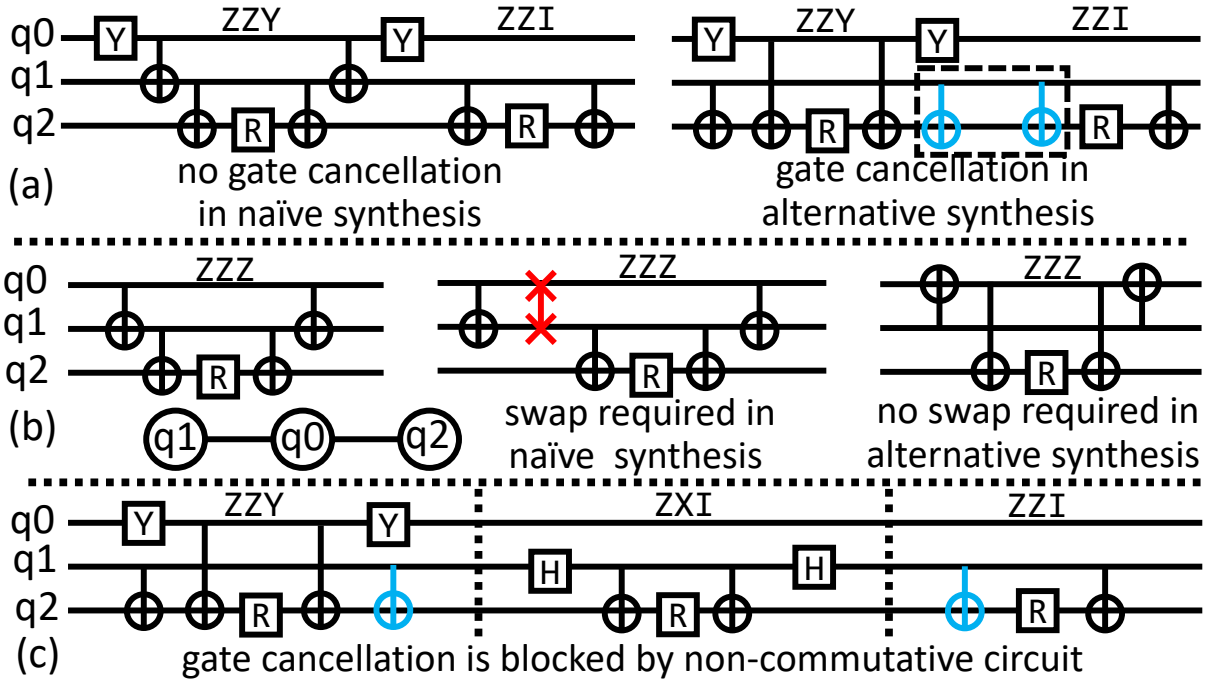


Figure 6.4: Optimization opportunities and challenges

6.3.1 Opportunities and Challenges

The optimization opportunities used in this chapter come from the properties of Pauli strings mentioned above. We introduce them by the examples in Figure 6.4. **1) Gate cancellation:** It is possible to have more gate cancellation by selecting a different synthesis plan for the $\exp(\mathrm{i}P\theta)$. Suppose the naive synthesis is the one in Figure 6.2 (1) and we have two Pauli strings, ZZY and ZZI. Under the naive synthesis (on the left of Figure 6.4 (a)) there is no gate cancellation. However, in an alternative synthesis of ZZY, we can have two CNOT gates cancelled (on the right of Figure 6.4 (a)). **2) Mapping:** The mapping overhead onto connectivity-constrained architectures can also be reduced. For example, we wish to map the ZZZ simulation circuit onto a linear architecture with the current mapping shown in Figure 6.4 (b). Under the naive synthesis we need to insert one SWAP between $q0$ and $q1$. While a better synthesis plan on the right of Figure 6.4 (b) does not require any SWAPs.

Although there is much optimization space for quantum simulation kernels, such optimizations are not yet widely deployed in today's quantum compiler infrastructures due to the following challenges. **1) Missing high-level information:** Once the program is converted to basic gates, where today's compilers perform most optimizations, it is hard to identify and reconstruct the high-level semantics of Pauli string simulation circuit blocks from an assembly-style gate sequence. **2) Non-semantics-preserving optimization:** To leverage some optimization opportunities would require non-semantics-preserving operations that are usually not allowed in a compiler. For example, consider the program in Figure 6.4 (c). It is known from Figure 6.4 (a) that gates can be cancelled between ZZY and ZZI but now there is an ZXI simulation circuit between them. We observe that the order of the simulation terms with respect to different Pauli strings is not specified in the Trotter formula or the variational form requirement. So, from an algorithmic perspective, the compiler may exchange the order of ZZI and ZXI, making ZZY and ZZI adjacent for gate cancellation. However, such operation is not semantics preserving from a gate-level perspective because, in general, $\exp(\mathrm{i}ZZI\theta_1)\exp(\mathrm{i}ZXI\theta_2) \neq \exp(\mathrm{i}ZXI\theta_2)\exp(\mathrm{i}ZZI\theta_1)$. This would be impossible to leverage without an IR that is able to encode such algorithmic knowledge.

6.3.2 Pauli IR: Syntax and Semantics

To overcome the challenges above, the objective of the new IR is to maintain high-level algorithmic information and make all transformations semantics-preserving. Our new IR, namely Pauli IR, realizes them with its syntax and semantics.

Syntax: The syntax is shown in Figure 6.5 and explained as follows. A *program* is recursively defined as a list of *pauli_block*s. Each *pauli_block* is a tuple with two elements. The first element is a list of weighted Pauli strings (*pauli_str_list*s) and the

$$
\begin{aligned}
\langle program \rangle &::= \langle pauli\_block \rangle \\
&\quad \mid \langle program \rangle \ ; \ \langle pauli\_block \rangle \\
\langle pauli\_block \rangle &::= \{ \langle pauli\_str\_list \rangle, parameter \} \\
\langle pauli\_str\_list \rangle &::= \langle pauli\_str, weight \rangle \\
&\quad \mid \langle pauli\_str\_list \rangle \ ; \ \langle pauli\_str, weight \rangle \\
\langle pauli\_str \rangle &::= \sigma_{n-1}\sigma_{n-2}\cdots\sigma_0 \\
\sigma_i &::= I \mid X \mid Y \mid Z, \ (0 \le i \le n-1) \\
parameter, \ weight \ &\in \mathbb{R}
\end{aligned}
$$

Figure 6.5: Formal syntax of an n-qubit Pauli IR program

second element is a real-valued *parameter* shared by all Pauli strings in this *pauli_block*. One element in the *pauli_str_list* is an n-qubit Pauli string and a real-value *weight*. Figure 6.6 shows the Pauli IR code of three example programs. Figure 6.6 (a) simulates the Hamiltonian of H_2 and each *pauli_block* has one *pauli_str*. Figure 6.6 (b)(c) are variational quantum algorithms so that parameters are labeled by θ and γ. In the UCCSD program (Figure 6.6 (b)), each *pauli_block* has multiple *pauli_str*s which share the same θ in the *pauli_block*. In the QAOA program (Figure 6.6 (c)), all *pauli_str*s are in one *pauli_block*, sharing the parameter γ.

Encoding constraints: One key advantage of the IR syntax is that the algorithmic constraints in all simulation kernels, as far as we know, can be naturally encoded. In some simulation kernels (e.g., UCCSD [48], QAOA for constrained optimization [210]), the algorithm requires that some Pauli strings should always appear together for some algorithmic purposes like symmetry preserving [211], parameter sharing [6, 48], error suppression [196], etc. Pauli IR employs a *pauli_block* structure to represent such constraints. The compiler can extract such information and all the *pauli_str*s inside one *pauli_block* are always scheduled together in follow-up optimization passes. In the rest of this chapter, *pauli_block* is denoted by block for simplicity.

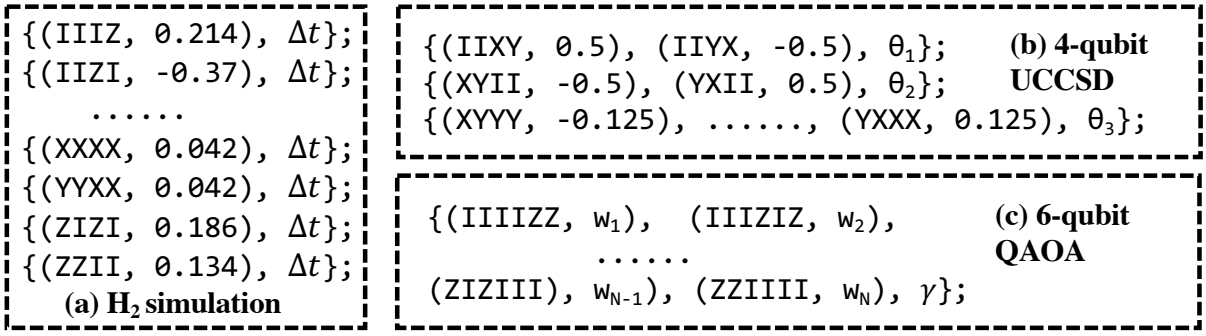


Figure 6.6: Example Puali IR programs

$$
\begin{aligned}
[\![\varnothing]\!] &= 0 \\
[\![\langle program\rangle; \langle pauli\_block\rangle]\!] &= [\![\langle program\rangle]\!] + [\![\langle pauli\_block\rangle]\!] \\
[\![\{\langle pauli\_str\_list\rangle, parameter\}]\!] &= parameter \times [\![\langle pauli\_str\_list\rangle]\!] \\
[\![\langle pauli\_str\_list\rangle; \langle pauli\_str, weight\rangle]\!] &= [\![\langle pauli\_str\_list\rangle]\!] \\
&\quad + [\![\langle pauli\_str, weight\rangle]\!] \\
[\![\langle pauli\_str, weight\rangle]\!] &= weight \times [\![pauli\_str]\!] \\
[\![\sigma_{n-1}\sigma_{n-2}\cdots\sigma_0]\!] &= \sigma_{n-1} \otimes \sigma_{n-2} \otimes \cdots \otimes \sigma_0
\end{aligned}
$$

Figure 6.7: Formal semantics of an n-qubit Pauli IR program

Semantics: The IR's semantics function, which is denoted by $[\![\langle program\rangle]\!]$, can be formally defined by the rules in Figure 6.7. This function is a mapping from the IR syntax to the set of all Hermitian operators in a 2^n-dimensional Hilbert space as our IR is to represent the Hamiltonian to be simulated. Note that the rules in the second and the fourth rows are defined based on *matrix addition* which is always commutative. As a result, exchanging the order of the *pauli_block*s in a *program* or the order of $\langle pauli_str, weight\rangle$s in a *pauli_block* will not change the semantics.

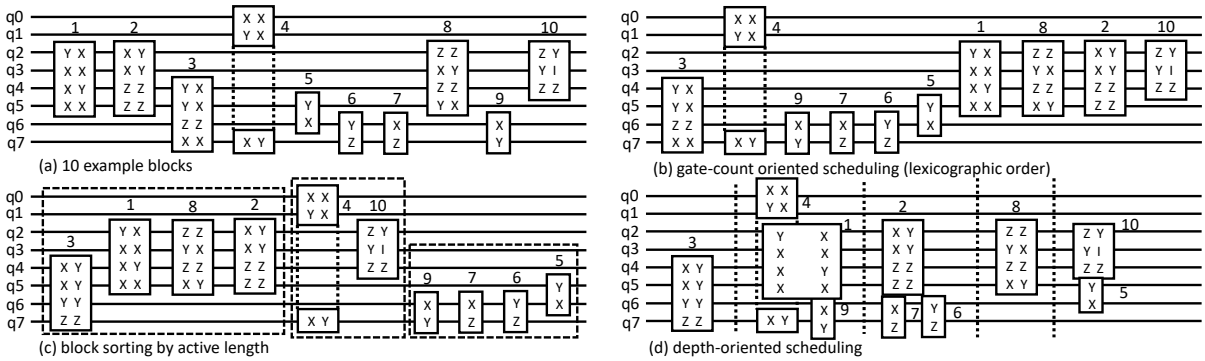


Figure 6.8: Example of block scheduling optimizations

6.4 Block-Wise Instruction Scheduling Passes

The first step in Paulihedral is to schedule the blocks and the instructions within each block. Intuitively for two adjacent Pauli strings, more gates can be cancelled if they share the same non-identity operators on more qubits. Trying to maximize the number of shared operators between consecutive strings would be desirable. Also, it is possible to execute multiple blocks which have non-identity operators on disjoint sets of qubits in parallel and reduce the final circuit depth. In this section, we present two block scheduling algorithms for two optimization objectives, reducing the total gate count or the circuit depth. We explain our block scheduling optimizations using the example in Figure 6.8. Suppose we will schedule 10 blocks on 8 qubits (Figure 6.8 (a)). In these blocks, each column is a Pauli string. The identity operators are omitted since they do not result in any circuit.

6.4.1 Gate-Count-Oriented Scheduling

Lexicographic ordering of Pauli strings has been shown to be effective at enabling gate cancellation between them [212, 196]. Here we adapt this principle to the multi-string-per-block case for our gate-count-oriented scheduling algorithm. In the lexicographic

Algorithm 8: Depth-oriented scheduling

Input: List of Pauli blocks.
Output: Pauli Layers L.

```
Sort Pauli blocks by active-length-decreasing order, then sort blocks of the same
  active length by lexicographic order;
R = the set of all Pauli blocks remaining; L = ∅;
Initialize the first layer;
while R is not empty do
    next_block = arg max_{block∈R} Overlap(block, last Pauli layer);
    pauli_layer = [next_block]; R.remove(next_block);
    while total depth of the small padding blocks ¡ the depth of next_block do
        find small Pauli block not overlapped with next_block;
        Append these blocks to pauli_layer;
        Remove these blocks from R;
    end
    L.append(pauli_layer);
end
```

order, the Pauli strings are scheduled in the alphabetical order. In Figure 6.8, we assume $X < Y < Z < I$ and use little-endian to lexicographically order from $q7$ down to $q0$. When a block has multiple strings, we first apply the lexicographic order on all strings in this block and then use the first string to represent this block when compared with other blocks. The first Pauli string can be representative because the strings in one block are usually mutually commutative in practical algorithmic constraints [48, 6, 196]. Two strings in a mutually commutative set can share the same operators on many qubits and all strings in one block are similar. Figure 6.8 (b) shows the result of gate-count-oriented scheduling.

6.4.2 Depth-Oriented Scheduling

The blocks can also be scheduled for reducing circuit depth. For example, in Figure 6.8, $q0$ to $q5$ are idle when executing block 9, 7, and 6. We may execute block 1 with them in parallel so that the overall circuit depth can be reduced. We propose a new

depth-oriented block scheduling algorithm, whose pseudocode is in Algorithm 8. For the example in Figure 6.8, we first sort all blocks by the active length of the Pauli strings of the blocks in a decreasing order. The active length of a block is defined by the number of qubits which have a non-identity operator in at least one Pauli string of this block. This is an over-approximated estimation on how a block will occupy the qubits. The blocks of the same active length are ordered by the lexicographic order above. Figure 6.8 (c) shows the sorting result. Block 3, 1, 8, 2 have the largest active length of 4 so they are at the beginning. Block 9, 7, 6, 5 have the smallest active length of 2 and they are at the end.

Then we begin to schedule all blocks and put the blocks in different layers to increase the parallelism. For each layer, we first schedule a large active length block. Then we search for small active length blocks that can be executed in parallel with the large block. For the example in Figure 6.8 (d), we initialize the first layer by selecting the first block after the sorting. We place the block 3 at the beginning. Then we search for small blocks that have no overlapped active qubits with the large block and can be executed in parallel. There are no such small blocks for block 3 so we continue by start another layer with block 1. In this layer, block 4, 9, 7, 6 can be placed in parallel with block 1. We iterate over the sorted block and find the first few blocks that can padded in this layer. In this example, we select block 4 and 9. We also estimate the depth of these small blocks so that total depth of these blocks will not exceed the depth of the original large block in this layer. We repeat this padding process until we cannot find any new blocks that can be added in this layer. We then continue to the next layer and start with block 2 because its first Pauli string has the most overlapped Pauli operators with the Pauli strings at the end of the previous layer. We iterate until all blocks are scheduled. Figure 6.8 (d) shows the final result of our depth-oriented scheduling and we can expect that the circuit depth can be reduced even if we do not convert the program to the gates.

This is another benefit of Pauli IR because the compiler can operate on a fairly compact description of the program. Once the program is lowered to gates, then the size blows up and parallelizing gates becomes much more expensive.

6.5 Block-Wise Optimization Passes

In this section, we introduce two optimization passes that can exploit the gate cancellation potential created by our scheduling passes in the last section, and convert the Pauli IR programs to gate sequences with different optimization objectives onto the fault-tolerant quantum computer (FT) backend and the near-term superconducting quantum computer (SC) backend.

6.5.1 On the Fault-Tolerant Backend

Our strategy for the FT backend is to adaptively find the synthesis plan that can maximize gate cancellation since the mapping overhead can usually be neglected after applying quantum error correction [213]. The pseudocode is shown in Algorithm 9, and we explain it with Figure 6.9. To capture the major gate cancellation opportunities, we scan over all layered blocks and try to select consecutive layer pairs that share the most Pauli operators. There should be significant operator overlap between consecutive layers since this was considered in our scheduling. The blocks on the left of Figure 6.9 are in five layers. Layer 1, 2, 3, 5 have one block in each and layer 4 has two blocks. We will pair the layer 3 and 4 together first since they share the same Pauli operators on 6 qubits. Then the first two layers are paired since they share Pauli operators on only 2 qubits. The last layer is left alone.

We first realize gate cancellation between the paired layers. For all layer pairs, we synthesize the Pauli strings at the end of the first layer and the Pauli strings at the be-

Algorithm 9: Optimization for FT backend

Input: List of Pauli layers pls
Output: A quantum circuit of basic gates

```
pl_paired = []; // paired Pauli layer list
while neighboring layers exist in pls do
    i = argmax_{i∈IndexSet(pls)} Overlap(pl_i, pl_{i+1}) ;
    pls.remove(pl_i); pls.remove(pl_{i+1});
    pl_paired.append((pl_i, pl_{i+1}));
end
for (pl_1, pl_2) in pl_paired do
    ps_list_1 = last Pauli string of pl_1;
    ps_list_2 = first Pauli string of pl_2;
    analyze string overlap then do synthesis on (ps_list_1, ps_list_2);
    pl_1.remove(ps_list_1); pl_2.remove(ps_list_2);
    for pb in (pl_1 + pl_2) do
        most_overlap_sort(pb); // find overlap at Pauli-string-level
        analyze string overlap then do synthesis on the sorted strings in pb;
    end
end
for pb in pls do
    most_overlap_sort(pb); analyze string overlap then do synthesis on the sorted
     strings in pb;
end
```

ginning the second layer in the pair. For the layer 3 (block 3) and the layer 4 (block 4 and 5), we need to handle the $IYXXYXXI$ in the layer 3 and ($IIIIYXXX$,$YYXXIIII$) in the layer 4. There are two sets of overlapped operators, YXX on qubit 3-1 and YXX on qubit 6-4. For each set, most gates can be directly cancelled, and we can select one qubit from each set and connect them with CNOT gates. The synthesis result for these two layers with gates cancelled is shown on the right of Figure 6.9. We repeat this process to optimize the synthesis of Pauli strings at the junction of two paired layers. Here we will synthesize the last string in layer 1 and the first string in layer 2.

We then realize the gate cancellation between strings inside a block. For those Pauli strings in the paired layer but not synthesized (one block with multiple strings), we

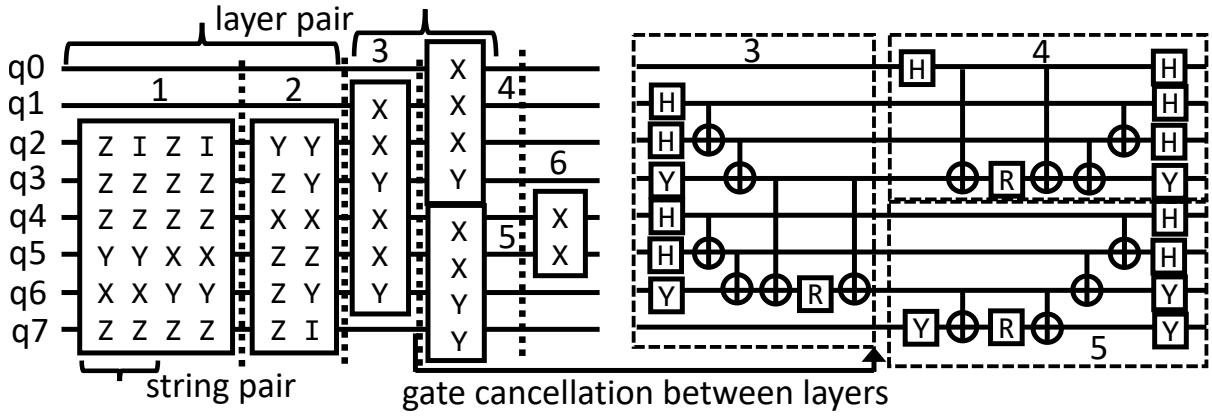


Figure 6.9: Example of compilation onto FT backend

employ a similar strategy at the string level for all Pauli strings inside one block. For each block, we search for string pairs that share the same Pauli operators on the most qubits and then synthesize these pairs first. In the block 1 in Figure 6.9, the first three Pauli strings are not yet synthesized. We will pair and synthesize the first two Pauli strings since they share 5 Pauli operators and a lot of gates can be cancelled. For the individual Pauli strings left, they are not paired with other strings (e.g., the third string in block 1). We check if it shares more Pauli operators with its left neighbor string or right neighbor string. Then we select the one with more gate cancellation and synthesize the Pauli string accordingly. For the blocks that are not paired with other blocks at the beginning of this algorithm (e.g., block 6), we treat them as unsynthesized Pauli strings and apply the same strategy, pairing and synthesizing the strings with high gate cancellation potential first then dealing with individual strings. Finally, all Pauli strings are compiled and we obtain a gate sequence of the input Pauli IR program. The final gate count is substantially reduced because the gate cancellation potential created by our block scheduling passes is maximally exploited through the adaptive synthesis plan in our block-wise optimization pass.

6.5.2 On the Near-Term Superconducting Backend

The compilation is more complicated for the SC backend because the SWAP gates are necessary to change the qubit mapping due to the qubit connectivity constraints. The gates are not uniform as they have different error rates on different qubits. We assume that the device calibration information (qubit coupling graph and the gate error rates on each qubit and qubit pair) is provided by the vendor. The major objective on the SC backend is to reduce the mapping overhead.

Our key idea is to find a tree embedding in the coupling map that can support the Pauli strings in the current layer and also minimize the mapping transition overhead between layers. Algorithm 10 shows the pseudocode, and we explain it using the example in Figure 6.10. For the initial qubit layout, we map all qubits to the most connected subgraph in the device coupling map. Suppose the coupling map and the current mapping of Figure 6.10 (b). We then begin to generate the simulation circuits and insert SWAPs for the blocks that appear in the critical path. In our block scheduling, we have already placed the blocks in different layers. In each layer, the largest block (involving the most qubits) is most likely on the critical path. Our optimization pass will first process the largest block in each layer, followed by the small blocks remaining. The program in Figure 6.10 (a) has two layers in which block 3 and 4 are the largest blocks.

For each block, we first select a root qubit. We define that the core qubit list of a block contains the qubits which have a non-identity operator on all Pauli strings in the block (e.g., q2-5 for block 3, q(2,4,6) for block 4). For block 3, since it is the first layer, we only need to consider itself. For q2-5 in its core list, they are already in a connected subgraph (Figure 6.10 (b)). We select any one of them (e.g., q2) as the root. And we only need to attach q6 to this subgraph by connecting it to any node of this graph. Suppose we swap q6 with q0 and now all active qubits in this block are connected in a

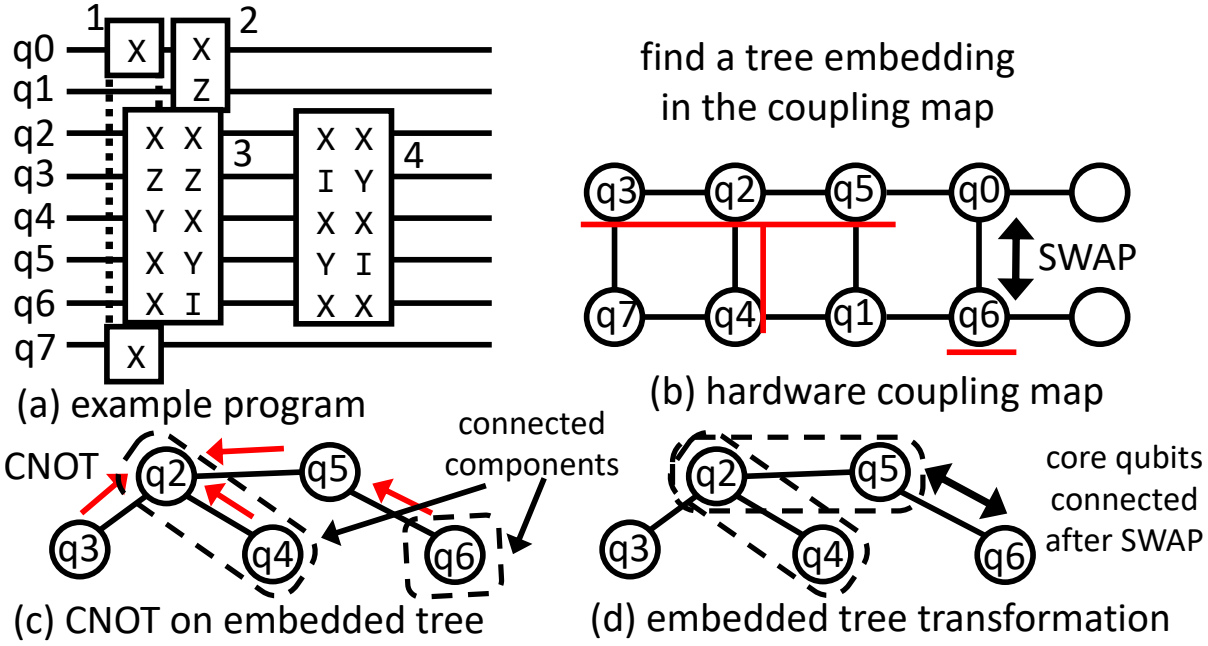


Figure 6.10: Example of compilation onto SC backend

subgraph. Active qubits are those qubits that have a non-identity operator in at least one string in this block. We can naturally generate an embedded tree from the coupling map (Figure 6.10 (c)).

Next we can synthesize the strings in block 3. The key idea is to naturally implement the CNOT tree in the Pauli circuits on the embedded tree so that we do not need to insert SWAPs for all individual CNOTs. We generate CNOT gates and single-qubit gates from the outermost qubits to the root for all the Pauli strings in the current block. If a qubit is active in the current Pauli string, we will check if its parent node is also active in the current Pauli string. If so, we insert a CNOT between the qubit and its parent. Otherwise, we swap it with its parent so that the qubit can get closer to the root and will be connected by CNOT later. In Figure 6.10 (c), the generated CNOTs are labeled by red arrows. After we determine the left CNOT tree, the right CNOT tree can be generated by reversing the order of CNOTs in the left tree.

After we process block 3, we will compile block 4, the next block in the critical path.

As our block scheduling passes tend to maximize the overlap between two consecutive layers, the core lists of two consecutive layers are similar. For example, $q(2,4,6)$ are in the core list of block 4 and they all appear in the core list of block 3. We evaluate all these qubits to select the root qubit with the largest connected component (within the core list) in the current mapping (Figure 6.10 (c)) to minimize the transition overhead. For $q(2,4,6)$, we will select $q2$ or $q4$ since they are in a size-2 connected component while $q6$ is in a size-1 connected component. Similarly, we then move all other active qubits to the tree through the path with the smallest error rate estimated by the device information. Here we select $q2$ as the root and then swap $q6$ and $q5$ to transit from block 3 to 4 with only 1 SWAP (Figure 6.10 (d)). After that the core qubits in block 4 are connected and we can begin synthesizing all strings in block 4.

The procedure above is to process the largest blocks in each layer. For other small blocks in the same layer, we follow a similar strategy and attempt to construct the trees for active qubits in those small blocks. If the trees of the small blocks do not affect the tree construction of the large block, we just process the small blocks in parallel with the large block since they will not affect each other. This will create parallelism and reduce the depth of the generated circuit. For example, block 2 and 3 can be processed in parallel because $q0$ and $q1$ are connected after swapping $q6$ with $q0$. However, if the trees of the small blocks affect the processing of the large block, we will put it in *remain_layer* and process them at the end. For example, block 1 will be in the *remain_layer* because connecting $q0$ and $q7$ will affect block 3.

After we process all the large blocks in the critical path and those small blocks that can executed in parallel, we will compile the blocks in the *remain_layer*. The order of processing these blocks is determined by whether the active qubits are close in the current mapping. We compute the cumulative distance between active qubits in a block and then compile the block with the smallest cumulative distance and update the qubit

mapping. This process is repeated until all the blocks are processed.

6.6 Evaluation

In this section, we evaluate Paulihedral by comparing with state-of-the-art baselines, analyze the effects of individual passes, and perform real system study.

6.6.1 Experiment Setup

Backend: The optimizations in this chapter target two different backends, the fault-tolerant backend (FT) and near-term superconducting backend (SC). We will cover both of them. We select IBM's latest 65-qubit Manhattan architecture [157] as the SC backend. For real system study, we use IBM's 16-qubit Melbourne chip, the largest publicly available one.

Metric: We use the CNOT/single-qubit gate count, and the circuit depth in the post-compilation program to evaluate Paulihedral. For the SC backend, the CNOT gate count is more important due to its higher error rate and latency. The depth is also important due to short qubit coherence times. For the FT backend, T gate is usually more expensive but for the simulation kernels, the ratio between the H, Y, CNOT gate count and the T gate count grows linearly as the number of qubits increases. Because a Pauli string of length n will have $O(n)$ H, Y, and CNOT gates but the number of Rz gates (the only source of T gates) is always one. It has also been shown that CNOT count is a significant cost in fault-tolerant algorithms and should not be neglected compared to T gates [214]. Hence, we estimate the performance with total gate count and circuit depth, following convention in previous work [197, 195, 198, 185].

Benchmark: We select 31 benchmarks of different sizes and various applications. For the SC backend, we select VQE UCCSD ansatzes [48] of six sizes, and the QAOA

programs [6] for graph max-cut on regular (REG) graphs of degrees 4, 8, 12, and random (Rand) graphs of edge probability 0.1, 0.3, 0.5, as well as traveling salesman problem (TSP) of different sizes. These benchmarks are generated by Qiskit [159]. For the FT backend, we first generate the Hamiltonians of five molecules using PySCF [165] (N_2, H_2S, MgO, CO_2, NaCl). We also prepare the Hamiltonians of three Ising models and three Heisenberg models, both of which are widely used in condensed matter physics, of different dimensions. We finally generate random Hamiltonians (Rand) of various sizes (30 to 80 qubits) for a more comprehensive evaluation. For a Hamiltonian of n qubits, we prepare $5n^2$ Pauli strings. In each Pauli string, we first randomly select one integer m between 1 and n. Then we randomly select m qubits and assign random Pauli operators to them. The rest $n - m$ qubits will be assigned with the identity. Table 6.1 shows the details of these benchmarks. Note that 'Pauli #' represents the number of Pauli strings. We include the CNOT and single-qubit gate counts when naively converting these benchmarks into gates without any optimization/transformations, and neglecting mapping overhead.

Implementation: We prototype Paulihedral in Python 3.8 (denoted by 'PH'). The entire compilation flow has two stages. The first stage is the quantum simulation program optimizations. The baselines include the Quantinuum's t|ket⟩ compiler [161] which employs the simultaneous diagonalization [198, 185, 186], a popular technique for optimizing quantum simulation programs ('TK'), and the QAOA compiler [200, 206, 207], an algorithm-specific compiler for unconstrained optimization QAOA on graphs ('QAOA compiler'). The second stage is the generic compilation and we have two industry generic compilers, the IBM's Qiskit [159] at the highest optimization level 3 ('Qiskit_L3') and the Quantinuum's t|ket⟩ generic compiler [161] at the highest optimization level 2 ('tket_O2'), The experiments are performed on a server with a 28-core Intel Xeon Platinum 8280 CPU and 1TB RAM. Note that due to the limited representation ability of t|ket⟩, the algorith-

Table 6.1: Benchmark information

Backend	Type	Name	Qubit #	Pauli #	CNOT #	Single #
SC	UCCSD	UCCSD-8	8	144	1134	1240
		UCCSD-12	12	1476	16192	15588
		UCCSD-16	16	4200	56558	47044
		UCCSD-20	20	8316	132326	109248
		UCCSD-24	24	9300	146312	115584
		UCCSD-28	28	20724	353984	270196
	QAOA	REG-20-4	20	40	80	40
		REG-20-8	20	80	160	80
		REG-20-12	20	120	240	120
		Rand-20-0.1	20	18	37	18
		Rand-20-0.3	20	56	113	56
		Rand-20-0.5	20	93	187	93
		TSP-4	16	112	192	112
		TSP-5	25	225	400	225
FT	Ising	Ising-1D	30	29	58	29
		Ising-2D	30	49	98	29
		Ising-3D	30	59	118	59
	Heisenberg	Heisen-1D	30	87	174	319
		Heisen-2D	30	147	294	539
		Heisen-3D	30	177	354	649
	Molecule	N_2	20	2951	39594	32151
		H_2S	22	4582	66026	52686
		MgO	28	24239	388258	310519
		CO_2	30	16154	252402	202282
		NaCl	36	67667	1249768	945935
	Random	Rand-30	30	4500	132939	99123
		Rand-40	40	8000	316039	229240
		Rand-50	50	12500	618763	441532
		Rand-60	60	18000	1068153	754071
		Rand-70	70	24500	1699771	1190101
		Rand-80	80	32000	2540640	1768117

mic constraints are hard to be encoded in 'TK'. To run our experiments and perform a fair comparison at our best, we relax those constraints in 'TK' and this relaxation allows a larger optimization space.

6.6.2 Comparing with t|ket⟩ and the QAOA Compiler

Table 6.2 shows the compilation time and results of the four configurations of all benchmarks on the two backends. Note that '>72 hrs' indicates that the 'tket_O2' takes

Table 6.2: Compilation time and results compared with t|ket⟩ [161]

	Time(s)		PH+Qiskit_L3				Time(s)	PH+tket_O2				Time(s)		TK+Qiksit_L3				Time(s)	TK+tket_O2			
	PH	Qiskit	CNOT	Single	Total	Depth	tket	CNOT	Single	Total	Depth	TK	Qiskit	CNOT	Single	Total	Depth	tket	CNOT	Single	Total	Depth
UCCSD-8	0	9	1165	667	1832	1404	16	1282	538	1820	1415	0	17	2247	1188	3435	1975	1	1775	346	2121	1527
UCCSD-12	1	120	17322	8607	25929	17808	485	20794	6816	27610	19439	0	331	35130	16113	51243	28910	31	26713	4007	30720	21747
UCCSD-16	3	471	58441	27607	86048	55566	7300	77274	19157	96431	65749	2	1525	140932	55678	196610	107177	179	106868	11837	118705	87064
UCCSD-20	8	827	106464	54951	161415	97168	11033	128195	37972	166167	107750	4	3171	315911	123823	439734	235862	655	223106	23736	246842	179618
UCCSD-24	9	1235	126013	68258	194271	113293	33155	187653	40357	228010	156431	6	4015	383444	145586	529030	285638	944	260282	29101	289383	213557
UCCSD-28	25	2759	280514	145672	426186	258326	69940	473093	92118	565211	386285	17	8933	870722	326600	1197322	644877	3313	595631	58782	654413	482484
REG-20-4	0	1	329	108	437	161	2	311	43	354	128	0	0	1594	523	2117	762	3	1441	40	1481	710
REG-20-8	0	2	519	266	785	290	7	574	84	658	267	0	0	1985	663	2648	894	3	1726	80	1806	816
REG-20-12	0	2	694	393	1087	396	12	842	125	967	404	0	0	1893	705	2598	765	3	1675	120	1795	775
Rand-20-0.1	0	1	188	46	234	97	1	167	20	187	67	0	0	577	184	761	255	0	448	18	466	214
Rand-20-0.3	0	1	414	181	595	236	5	439	60	499	202	0	0	1539	502	2041	688	2	1355	56	1412	644
Rand-20-0.5	0	2	571	313	884	335	9	674	98	772	333	0	0	1812	622	2434	735	3	1614	93	1707	746
TSP-4	0	2	500	381	881	257	8	583	112	695	283	0	0	1006	556	1562	446	2	1207	112	1319	530
TSP-5	0	3	1093	683	1776	636	24	1457	225	1682	534	0	0	2886	1250	4136	1030	9	3057	225	3282	1249
N_2	6	255	16632	11223	27855	17486	37	16002	10861	26863	17682	2	116	19762	9785	29547	21087	15	18928	9702	28630	20446
H_2S	16	450	25726	17580	43306	27676	94	24813	16836	41649	28051	3	209	35248	17453	52701	36450	47	33968	17075	51043	35329
MgO	615	2489	116973	87654	204627	126259	3291	113575	78585	192160	127619	20	1840	198428	92012	290440	203064	3818	192499	85694	278193	194905
CO_2	293	1656	96829	65243	162072	93679	2663	93391	61386	154777	95302	15	1182	126634	58368	185002	129697	1699	121768	56277	178045	124557
NaCl	7377	7492	316456	247630	564086	338763	28920	307038	223261	530299	342775	63	7262	626692	267424	894116	626665	87838	605201	247500	852701	599322
Ising-1D	0	0	58	29	87	6	0	58	29	87	6	0	2	508	29	537	450	0	508	29	537	450
Ising-2D	0	0	98	49	147	18	0	98	49	147	18	0	1	306	49	355	219	0	306	49	355	219
Ising-3D	0	0	118	59	177	18	0	118	59	177	18	0	1	290	59	349	188	0	290	59	349	188
Heisen-1D	0	1	87	204	291	13	0	87	190	277	13	0	1	172	176	348	100	0	169	200	369	126
Heisen-2D	0	1	216	315	531	43	0	212	284	496	47	0	1	293	239	532	89	0	293	294	587	98
Heisen-3D	0	2	305	366	671	65	0	295	335	630	67	0	2	365	271	636	118	0	364	328	692	125
Rand-30	13	1386	94222	47315	141537	70787	2329	89152	50490	139642	78429	5	779	114043	55665	169708	87196	141	108943	62158	171101	97897
Rand-40	42	4253	233266	108988	342254	166989	175842	223946	119225	343171	189856	11	1775	270928	125264	396192	199567	961	259985	142805	402790	227284
Rand-50	104	9144	470240	211605	681845	329388	55853	455607	234635	690242	379553	24	3575	533631	239379	773010	388722	5220	514235	276401	790636	447315
Rand-60	203	14007	834418	364816	1199234	575973	>72hrs	N.A.				44	5930	926982	404279	1331261	666338	22096	895895	471005	1366900	768714
Rand-70	393	27849	1345439	576378	1921817	924730	>72hrs	N.A.				74	9672	1476952	636284	2113236	1050970	138031	1430214	746249	2176463	1220344
Rand-80	709	39322	2033283	856105	2889388	1386577	>72hrs	N.A.				116	17092	2205935	938304	3144239	1555808	285101	2138232	1104321	3242553	1811440

over 72 hours and was shut down in the middle. In summary, 'PH' outperforms 'TK' with substantial gate count and circuit depth reduction while only introducing $\sim 5\%$ additional time ('PH' vs 'Qiskit/tket') in the entire compilation flow.

On the SC backend, 'PH' can reduce the CNOT, single-qubit, total gate count, and circuit depth by 66.2% (43.3%), 53.4% (-22.7%), 62.6% (41.2%), and 60.8% (44.3%), respectively on average, compared with 'TK' using 'Qiskit_L3' ('tket_O2') generic compilation. 'PH' can achieve such significant improvement because 'TK' does not support mapping-aware optimization for general Pauli strings and can only do a inefficient generic qubit mapping. The single-qubit gate count increases when using 'tket_O2' but this does not affect the overall improvement since the CNOT gates have much higher error rates on the SC backend and latency and the total single-qubit gate count is still relatively low.

On the FT backend, 'PH' can reduce the CNOT, single-qubit, total gate count, and circuit depth by 38.7% (44.5%), 18.6% (3.0%), 32.8% (34.4%), and 61.7% (68.0%), respectively on average, compared with 'TK' using 'Qiskit_L3' ('tket_O2'). The circuit depth reduction is significant due to the depth-oriented scheduling in 'PH'. Our

Table 6.3: Comparing with QAOA compiler [200]

	PH+Qiskit_L3					QAOA_Compiler+Qiskit_L3				
Benchmark	CNOT	Single	Total	Depth	Time(s)	CNOT	Single	Total	Depth	Time(s)
REG-20-4	329	108	437	161	0.14	394	101	495	171	6.32
REG-20-8	519	266	785	290	0.23	727	141	868	297	10.27
REG-20-12	694	393	1087	396	0.29	1020	181	1201	399	14.55
Rand-20-0.1	188	46	234	97	0.08	212	80	292	111	4.52
Rand-20-0.3	414	181	595	236	0.1	546	118	664	230	7.74
Rand-20-0.5	571	313	884	335	0.12	842	155	997	334	12.3

block-wise optimization is also much effective compared with 'TK' strategy. The details of 'TK' are not public and what we can infer, at our best, from their limited documents [161, 198, 185, 186] is that the simultaneous diagonalization may introduce too much overhead. For example, the 'Ising-1D' program has even more gates after 'TK'. One possible reason is that all Pauli strings in Ising-1D are mutually commutative and it takes many additional gates to simultaneously diagonalize all these Pauli strings.

Table 6.3 shows the compilation results of 'PH' and the QAOA compiler [200] on the 6 MaxCut problems. We ran the QAOA compiler with 20 random seeds for each program and collected the averaged compilation results. Comparing with the QAOA compiler, Paulihedral can achieve 24.6%, 12.2%, and 3.2% reduction in CNOT count, total gate count, and circuit depth, respectively on average, using only 1.7% compilation time. The overhead is about 40% in single-qubit gate count, but in QAOA the CNOT count is usually over $3-4\times$ higher than single-qubit gate count and CNOT error rate is usually $10\times$ higher on the SC backend. Therefore, 'PH' significantly outperforms QAOA compiler, even though it is more general purpose and not tailored to a single algorithm. This is because 'PH' employs a block-wise optimization for searching SWAPs and the search scope is much larger than that of the QAOA compiler's greedy search.

6.6.3 Pass Option Comparison

Now we study the effect of different pass options in Paulihedral. We first compare the two block scheduling passes.

DO vs GCO scheduling: On the left of Table 6.4 we show the difference between the depth-oriented (DO) scheduling and the gate-count-oriented (GCO) scheduling (in Section 6.4). Overall, across the 17 benchmarks on the FT backend, 'DO' can yield low-depth circuits while 'GCO' can reduce the gate count more. The circuit depth of DO is 46.7% (geomean) compared with that of GCO and the gate count overhead is 5.9%, 0.64%, and 3.3% for CNOT, single-qubit, and total gate count, respectively. For benchmarks on the SC backend, the effect of the block scheduling is largely amortized by mapping overhead reduction since the tested Manhattan architecture has very sparse qubit connection. For the UCCSD benchmarks, 'DO' and 'GCO' share similar overall performance. For the QAOA benchmarks, there is no difference between 'DO' and 'GCO' since the entire kernel has only one block.

BC improvement: Our block-wise compilation (BC) passes (in Section 6.5) can significantly reduce the gate count and circuit depth. On the right of Table 6.4 we show the comparison between using BC against a naive synthesis and Qiskit_L3. For the 17 benchmarks on the FT backend, BC reduces the circuit depth, the CNOT, single-qubit, and total gate counts by 15.5%, 6.0%, 3.1%, and 5.0%, respectively. On the SC backend, the BC pass is even more effective since the large mapping overhead can be greatly reduced. For the UCCSD (QAOA) benchmarks, BC can reduce the CNOT, single-qubit, total gate count, and circuit depth by 60% (33%), 45% (8%), 56% (26%), and 53% (−14%), respectively on average. The circuit depth of QAOA benchmarks is increased since the BC focus more on SWAP reduction, leading to fewer gates but deeper circuits because the effect of SWAP reduction is relatively limited in the small-size QAOA

Table 6.4: Pass option effect comparison

	DO vs GCO				Block-Wise Compilation improvement			
	CNOT	Single	Total	Depth	CNOT	Single	Total	Depth
UCCSD-8	-4.43%	-1.19%	-3.27%	0.72%	-51.07%	-51.14%	-51.09%	-37.93%
UCCSD-12	6.32%	-8.41%	0.93%	2.37%	-57.38%	-55.35%	-56.73%	-50.06%
UCCSD-16	-2.62%	-6.32%	-3.84%	-1.94%	-59.61%	-45.26%	-55.90%	-51.89%
UCCSD-20	-3.81%	-8.55%	-5.48%	-6.49%	-72.40%	-56.43%	-68.47%	-67.66%
UCCSD-24	-5.25%	3.90%	-2.23%	-10.60%	-68.72%	-49.02%	-63.80%	-63.38%
UCCSD-28	-1.19%	5.00%	0.84%	-2.34%	-73.97%	-56.31%	-69.80%	-67.22%
REG-20-4	N.A.				-21.85%	-6.90%	-18.62%	27.78%
REG-20-8	N.A.				-34.63%	1.14%	-25.73%	18.37%
REG-20-12	N.A.				-36.62%	-12.86%	-29.69%	13.79%
Rand-20-0.1	N.A.				-11.74%	-16.36%	-12.69%	36.62%
Rand-20-0.3	N.A.				-28.99%	-2.16%	-22.53%	26.20%
Rand-20-0.5	N.A.				-38.60%	-4.28%	-29.67%	10.20%
TSP-4	N.A.				-43.12%	-12.81%	-33.05%	-19.18%
TSP-5	N.A.				-47.53%	-7.58%	-37.07%	-2.30%
N_2	13.30%	3.13%	8.97%	-7.59%	-4.54%	-2.93%	-3.90%	-6.77%
H_2S	17.43%	6.24%	12.61%	-3.09%	-6.32%	-2.94%	-4.98%	-8.25%
MgO	31.16%	8.51%	20.40%	-0.48%	-6.89%	-8.17%	-7.44%	-9.45%
CO_2	26.08%	6.43%	17.36%	-8.84%	-5.13%	-1.34%	-3.64%	-6.04%
NaCl	25.03%	6.55%	16.18%	-5.46%	-12.18%	-8.48%	-10.60%	-13.62%
Ising-1D	0.00%	0.00%	0.00%	-93.10%	0.00%	0.00%	0.00%	0.00%
Ising-2D	0.00%	0.00%	0.00%	-68.42%	0.00%	0.00%	0.00%	0.00%
Ising-3D	0.00%	0.00%	0.00%	-71.43%	0.00%	0.00%	0.00%	0.00%
Heisen-1D	0.00%	0.00%	0.00%	-92.57%	0.00%	7.37%	5.05%	0.00%
Heisen-2D	-19.10%	-12.01%	-15.04%	-82.30%	0.00%	3.28%	1.92%	0.00%
Heisen-3D	-8.41%	-13.68%	-11.36%	-80.83%	0.00%	1.95%	1.05%	0.00%
Rand-30	7.25%	6.95%	7.15%	-9.76%	-8.74%	-3.53%	-7.06%	-29.45%
Rand-40	6.11%	5.55%	5.93%	-9.46%	-9.68%	-2.99%	-7.65%	-31.80%
Rand-50	4.83%	4.98%	4.88%	-9.21%	-10.48%	-2.19%	-8.06%	-33.15%
Rand-60	4.10%	4.36%	4.18%	-9.30%	-10.75%	-1.90%	-8.23%	-33.44%
Rand-70	3.60%	3.79%	3.66%	-8.80%	-11.07%	-1.63%	-8.44%	-33.76%
Rand-80	3.27%	3.34%	3.29%	-8.70%	-11.19%	-1.54%	-8.54%	-34.17%

benchmarks.

Pauli string pattern effects: It can be observed that the effect of the passes vary on different benchmarks. The reason is that the Pauli strings in the benchmarks have different patterns which can be classified into two categories based on the numbers of non-identity operators in each Pauli string. As mentioned in Section 6.2, a Pauli string with more non-identity operators on more qubits will in general be converted to a larger circuit block involving more qubits and gates. The first category includes the molecule

Hamiltonians, the random Hamiltonians, and the UCCSD. In these Hamiltonians, many Pauli strings have non-identity operators on various numbers of qubits (up to all qubits). The second category includes the Ising, Heisenberg, and the selected QAOA benchmarks, of which the Hamiltonians only have Pauli strings with non-identity operators on at most two qubits. Such a difference in the operator distribution affects the compilation results.

On the FT backend, benchmarks in the first category (molecule and random Hamiltonians) benefit more from the BC optimizations since Pauli strings with more non-identity operators have larger potential in gate cancellation and depth reduction. Benchmarks in the second category (Ising and Heisenberg) cannot benefit from BC since those Pauli strings with only two non-identity operators can only be synthesized in a single way and there is no space BC can explore to further reduce the gate count and circuit depth. However, these benchmarks can benefit a lot from DO. GCO turns out to be inefficient in both gate count and circuit depth for them because GCO cannot create gate count reduction while DO can create additional single-qubit gate reduction opportunities between consecutive layers by putting many small-size blocks in one layer. On these benchmarks, DO completely outperforms GCO with on average 84.2% circuit depth reduction and 7.5% total gate count reduction. Similarly on the SC backend, the BC improvement on the UCCSD benchmarks (first category) is also more significant compared with the QAOA benchmarks (second category) because more gate can be cancelled and more SWAPs in the mapping overhead can be eliminated when the tree sizes are large for Pauli strings with more non-identity operators.

6.6.4 Pass Benefit Breakdown

To show the separate effect of scheduling passes and optimization passes, we prepare breakdown experiments on four benchmarks (two random Hamiltonian RAND-40 and

Table 6.5: Benefit breakdown of the block ordering and block-wise optimization passes

Benchmark	Metric	Baseline order	Baseline order + BC	DO order	DO order + BC
RAND-40	CNOT	313011	279165 (-10.8%)	259477 (-17.1%)	234264 (-25.1%)
	Single	157853	149234 (-5.5%)	143686 (-9.0%)	131302 (-16.8%)
	Depth	275175	188267 (-31.6%)	245675 (-10.7%)	168003 (-38.9%)
RAND-50	CNOT	613714	548499 (-10.6%)	529220 (-13.8%)	474999 (-22.6%)
	Single	300955	283647 (-5.8%)	279465 (-7.1%)	255104 (-15.2%)
	Depth	529844	360004 (-32.1%)	490793 (-7.4%)	329169 (-37.9%)
N_2	CNOT	36484	23371 (-35.9%)	17423 (-52.2%)	15981 (-56.2%)
	Single	20124	14713 (-26.9%)	11557 (-42.6%)	11366 (-43.5%)
	Depth	42525	25216 (-40.7%)	18749 (-56.0%)	16788 (-60.5%)
H_2S	CNOT	61127	38213 (-37.5%)	27752 (-54.6%)	24792 (-59.4%)
	Single	32885	24173 (-26.5%)	18154 (-44.8%)	17307 (-47.4%)
	Depth	70618	40985 (-42.0%)	30430 (-56.9%)	26581 (-62.4%)

RAND-50, two molecule Hamiltonians N_2 and H_2S). Their information is in Table 6.1. We have four configurations by combining two ordering options and whether to apply block-level optimizations. The baseline order is the original order of the problem Hamiltonians, which for RAND-40 and -50 is random and for N_2 and H_2S is determined by the Hamiltonian generation module in PySCF [165] (which is based on ordering the electron orbitals from low energy to high energy level). The DO order is the depth-oriented order in Section 6.4.2. 'BC' means that the block-level optimization in Section 6.5.1 is applied. All configurations are followed by Qiskit Level 3 optimization by default. The following table shows the CNOT gate count, single-qubit gate count, circuit depth, and their corresponding reduction percentage of each configuration compared to the 'Baseline order'.

In Table 6.5 we can find that both the DO order and the BC have substantial contributions to the final CNOT/single-qubit gate count and circuit depth reduction. Averaging over the results of the four selected benchmarks, the effect of BC is 69% of that of DO order scheduling in CNOT reduction, 65% in single-qubit gate reduction, and 1.62x in circuit depth reduction.

6.6.5 Real System Study

Finally, we evaluate 'PH' on IBM's 16-qubit Melbourne chip with 8 QAOA MaxCut programs. We generate 4 regular graphs of 7 to 10 nodes with 4 edges per node ('REG-n(7-10)-d4'), and 4 random graphs of 7 to 10 nodes with edge probability 0.5 ('RD-n(7-10)-p0.5'). We prepare 1-level QAOA circuits on these graphs and then optimize the parameters in the simulator. Those circuits with the optimized parameters are then evaluated on the Melbourne chip (40960 shots per circuit). The baseline is 'Qiskit_L3' with the Pauli strings ordered by iterating over the adjacency matrix (Qiskit default configuration).

Figure 6.11 shows the improvement of the success probability after applying 'PH' optimizations. The 'Estimated Success Probability' (ESP), a widely used metric in guiding the compiler optimization [215, 176, 104], is a theoretical estimation of the success probability based on the program and the hardware noise model. The 'Real System Success Probability' (RSP) is the number of trials with correct measurement results divided by the total number of trials when executing on the real machine. Applying 'PH' can improve the ESP by 2.11× on average (up to 3.00×) based on the noise model of the tested device, by reducing the CNOT count and circuit depth by 15.1% and 36.2%, respectively on average. On the real machine, 'PH' can improve the RSP by 1.24× on average (up to 1.87×). There is a gap between the results from ESP and RSP because the noise model only provides limited hardware information. We expect that the compilation can be further improved with more detailed hardware models.

Table 6.6 shows the detailed compilation results onto the IBM's 16-qubit Melbourne chip It can be observed that Paulihedral optimization leads to significant reduction in both gate count and circuit depth. More importantly, the reduction grows as the benchmark size increases, showing that Paulihedral will be more effective on larger size input

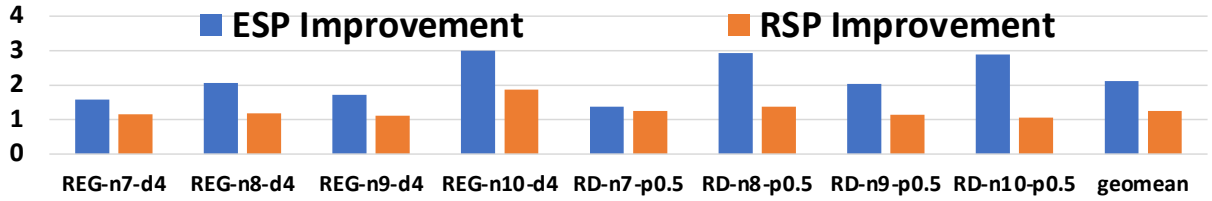


Figure 6.11: Success Probability Improvement for QAOA on IBM's 16-qubit Melbourne Chip

Table 6.6: Detailed compilation results onto IBM Melbourne

	Qiskit				Paulihedral+Qiskit			
Metric	CNOT	Single	Depth	RSP	CNOT	Single	Depth	RSP
REG-n7-d4	45	218	86	23.81%	43	131	56	27.50%
REG-n8-d4	78	202	100	10.97%	66	170	78	12.89%
REG-n9-d4	86	247	113	22.05%	67	124	70	24.69%
REG-n10-d4	122	238	114	10.22%	70	134	72	19.08%
RD-n7-p0.5	35	149	64	0.67%	40	82	39	0.85%
RD-n8-p0.5	68	166	102	0.09%	61	94	59	0.13%
RD-n9-p0.5	76	209	107	0.37%	71	126	67	0.42%
RD-n10-p0.5	141	314	174	0.08%	93	211	106	0.09%

programs. Such reduction can be turned into the final success probability improvement. We show the absolute success probabilities and the relative improvement in the second table. We can also observe that there is a significant difference between regular graphs and random graphs. This is because regular graphs have some symmetries and the solution space (the number of valid solutions) is much larger than that of random graphs (that is, the number of valid solutions are larger). This makes QAOA on regular graphs much more noise tolerant. The low absolute success probability comes from the fact that the Melbourne chip is an old device with higher error rates. But it is the only publicly accessible chip with 14 qubits and all other public devices are of 5-7 qubits.

6.7 Discussion

It would be always desirable to have more effective quantum compiler optimizations to fully exploit the potential of quantum computing. One common approach is to model the hardware more precisely (e.g., from coarse-grained gate count [56, 18, 216] to independent non-uniform gate error [104, 85], then correlated crosstalk error [158], and finally low-level pulse optimizations [217, 218]). The compiler can naturally exploit more potential from the hardware with more detailed hardware information.

Different from these compiler innovations that are mostly driven by the underlying technologies, Paulihedral takes another approach which is to enable deeper compiler optimizations by leveraging the algorithmic properties of the high-level quantum programs. Relatively little attention has been given to this direction because 1) it is exceedingly difficult to extract useful high-level semantics from the gate-sequence representation in today's compiler infrastructures, and 2) scalable yet effective static analysis of quantum programs is also very hard as the size of the operation matrices grows exponentially with the number of qubits. We believe that these are two critical yet difficult open problems in the future development of quantum compiler/software infrastructure since they prevent the compiler from automatically detecting high-level and large-scale optimization opportunities.

Paulihedral tackles these two problems for the quantum simulation kernel, a widely used subroutine, and thus can benefit the compiler optimization for many quantum algorithms. In particular, we define a new Pauli IR which can capture the high-level semantics of simulation kernels. The domain knowledge of quantum simulation can thus be exploited by the compiler automatically, yielding optimizations that are hard to be implemented in the conventional gate-based representation. We then design several new compiler passes, all of which are scalable block-wise circuit transformations since the

analysis on Pauli strings can be efficiently handled by classical computers. The evaluation in this work has covered a wide range of quantum simulation kernels and we expect that Paulihedral will continue to benefit future quantum algorithms since the quantum simulation has been a long-living algorithm design principle in the last few decades.

Looking forward, although Paulihedral is designed from an algorithmic perspective, it can incorporate those technology-driven optimizations. In this chapter, we have supported two different backends with two technology-dependent optimization passes targeting different objectives and hardware constraints. These passes can also be further optimized once we have a deeper understanding of the quantum devices and come up with more comprehensive hardware models. Paulihedral can be further extended to other quantum architectures (e.g., ion-trap-based architectures [188, 219], photonics [220]) by adding new optimization passes.

It is also possible to make Paulihedral more intelligent by automatically managing the passes based on the input program characteristics. Currently Paulihedral has four passes and we have already observed that the different Pauli string patterns can affect the final improvement under different pass configurations as discussed in Section 6.6. In the future, more passes can be included to cover more backends, error resources, architectural constraints, and optimization objectives. How to automatically select the most suitable combination of passes from a pool of compiler passes is worth to explore.

Finally, the idea of quantum algorithmic compiler can be extended to other promising quantum algorithm domains. There are several other important common techniques in quantum algorithm design (e.g., quantum phase estimation [25], amplitude amplification [221]) and promising quantum application domains (e.g., quantum machine learning [222]). How to design new programming languages to maintain the high-level semantics of these programs and then propose corresponding algorithmic compiler optimizations is still an open problem which can be left as future work.

6.8 Related Work

Paulihedral is a compiler framework with a new IR abstraction and deeper optimizations for general quantum simulation kernels. We first review the program representation and optimizations in quantum compilers. Then we discuss existing optimizations for quantum simulation programs.

IR in quantum compilers: Modern classical compilers employ multiple IRs (e.g., control flow graph, static single assignment) from high level to low level and different optimizations are applied on different IRs. Today's quantum compilers [159, 161, 223, 224, 225], on the other hand, are mostly built around low-level representations [226, 227, 228], which makes it difficult to extract high-level information about the semantics of the algorithm and discover non-commutative yet semantics-preserving re-orderings. The most recent version of open quantum assembly language (OpenQASM) [229] recognizes the need for higher-level semantics such as control, inverse, and power operations, but is still incapable of representing Pauli-level semantics which are prevalent in quantum simulation kernels. As we have shown, our Pauli IR can carry high-level semantics through multiple optimization stages, encode all known algorithm constraints, and is compatible with further low-level optimizations by these tools.

Quantum compiler optimizations: The state-of-the-art quantum compilers [159, 227] usually have multiple passes to execute different optimizations, (e.g., circuit rewriting [162], gate cancellation [163], template matching [164], qubit mapping [104]). These passes applied on the low-level gate sequences usually only rewrite the circuit locally on very few qubits or gates every time and only focus on one optimization objective in each pass. Different from these optimizations, all passes in Paulihedral performance program transformations at a much larger scope in a scalable way and multiple optimization opportunities can be reconciled because the high-level algorithmic information is lever-

aged. This makes Paulihedral optimizations more effective than simply combining those small-scale single-objective passes.

Optimizations for simulation algorithms: One common optimization technique is to group the Pauli strings into sets of mutually commutative strings and then apply simultaneous diagonalization [198, 185, 186, 197]. This technique, adopted by t|ket⟩ [161, 198, 185, 186], can simplify the circuit inside each set while the simultaneous diagonalization step introduces substantial overhead before and after the circuit of each set, limiting the overall optimization performance. Some other works [195, 196, 199, 20, 230, 103, 200, 201] explore the simulation program optimization or synthesis but these works are mostly ad-hoc, limited to specific algorithms/architectures, and not easily generalizable to a broader range of programs and employed by a compiler infrastructure. In Paulihedral, the Pauli IR's recursive, block-wise structure can support simulation kernels in all related algorithms, as far as we know. And our optimization algorithms have been shown to be much more effective in the evaluation above.

6.9 Conclusion

We propose Paulihedral, an algorithmic quantum compiler targeting the quantum simulation kernel, a subroutine widely used in many quantum algorithms. Paulihedral enables deep compiler optimizations by defining a new Pauli-string-based IR, which can encode high-level algorithmic information and constraints of many seemingly different quantum algorithms in a unified manner. All follow-up optimizations in Paulihedral operate at a large scope with good scalability and can reconcile multiple optimization opportunities. Paulihedral can be extended to different backends by adding or modifying technology-dependent passes. Comprehensive experimental results show that Paulihedral can significantly outperform state-of-the-art quantum compilers with more effective,

scalable optimizations and better reconfigurability.

Algorithm 10: Optimization for SC backend

Input: List of Pauli layers pls, device information
Output: Hardware compatible circuit Q

Map logical qubits to the most connected subgraph of the device coupling map;
for *pauli_layer in pls* **do**
  pb = the largest Pauli block in *pauli_layer*; s = core qubit list of pb;
  T_1 = node in s with largest connected component;
  connect active qubits in pb to tree T_1 through shortest path (lowest error rate); wl =leaves of T_1 sorted by depth;
  for *ps in pb* **do**
    while $wl \neq \varnothing$ **do**
      $n = wl.deque();\ np = n.parent$;
      if *n is the root of* T_1 **then** continue;
      if $ps[n] \neq I$ *and* $ps[np] \neq I$ **then**
        add single-qubit gates based on $ps[n]$ and $ps[np]$;
        $Q.append(CNOT(n, np))$;
      else if $ps[n] \neq I$ *and* $ps[np] == I$ **then**
        $Q.append(SWAP(n, np))$;
      end
      $wl.append(np)$;
    end
    generate the right half circuit of ps reversely;
  end
  for *spb in remaining blocks of pauli_layer* **do**
    T_2 = try_construct_tree(spb); `// Return NULL if changes` T_1
    synthesize spb with T_2 if T_1 not changed; otherwise add spb to *remain_layers*;
  end
end
while *remain_layers is not empty* **do**
  Sort *remain_layers* by cumulative distance between active qubits;
  Synthesize first layer of *remain_layers* with the same strategy and remove it from *remain_layers*;
end

Chapter 7

Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs

7.1 Introduction

Quantum computing is a promising computing paradigm with great potential in cryptography [3], database [7], linear systems [191], chemistry simulation [48], etc. Several quantum program languages [159, 231, 92, 232, 233, 234, 235] have been published to write quantum programs for quantum computers. One of the key challenges that must be addressed during quantum program development is to compose correct quantum programs since it is easy for programmers living in the classical world to make mistakes in the counter-intuitive quantum programming. For example, Huang and Martonosi [236, 237] reported a few bugs found in the example programs from the ScaffCC compiler project [117]. Bugs have also been reported in IBM's OpenQASM project [238] and Rigetti's PyQuil project [239]. These erroneous quantum programs, written and

reviewed by professional quantum computing experts, are sometimes even of very small size (with only 3 qubits)[1]. Such difficulty in writing correct quantum programs hinders practical quantum computing. Thus, effective and efficient quantum program debugging is naturally in urgent demand.

In this chapter, we focus on runtime testing and debugging a quantum program on a quantum computer, and revisit *assertion*, one of the basic program testing and debugging approaches, in quantum computing. There have been two quantum program assertion designs in prior research. Huang and Martonosi proposed statistical assertions, which employed statistical tests on classical observations [237] to debug quantum programs. Motivated by indirect measurement and quantum error correction, Liu et al. proposed a runtime assertion [240], which introduces ancilla qubits to indirectly detect the system state. As early attempts towards quantum program testing and debugging, these studies suffer from the following drawbacks:

1) **Limited applicability with classical style predicates:** The properties of quantum program states can be much more complex than those in classical computing. Existing quantum assertions [237, 240], which express the quantum program assertion predicates in a classical logic language, can only assert some simple quantum states of three special cases (detailed later in Section 7.5). A lot of complex intermediate program states cannot be tested by these assertions due to their limited expressive power. Hence, these assertions can only be injected at some special locations where the states are within the three supported types. Such restricted assertion types and injection locations will increase the difficulty in debugging as assertions may have to be injected far away from a bug.

2) **Inefficient assertion checking:** A general quantum state cannot be duplicated [241], while the measurements, which are essential in assertions, usually only probe

[1]We checked the issues raised in these projects' official GitHub repositories for this information.

part of the state information and will destroy the tested state immediately. Thus, an assertion, together with the computation before it, must be repeated for a large number of times to achieve a precise estimation of the tested state in Huang and Martonosi's assertion design [237]. Another drawback of the destructive measurement is that the computation after an assertion will become meaningless. Even though multiple assertions can be injected at the same time, only one assertion could be inspected per execution, which will make the assertion checking more prolonged [237].

3) **Lacking theoretical foundations:** Different from a classical deterministic program, a quantum program has its intrinsic randomness and one execution may not cover all possible computations of even one specific input. Moreover, some quantum algorithms (e.g., Grover's search [7], Quantum Phase Estimation [25], qPCA [192]) are designed to allow approximate program states, and the quantum program assertion checking itself is also probabilistic. Consequently, testing a quantum program usually requires multiple executions for one program configuration. It is important but rarely considered (to the best of our knowledge) what statistical information we can infer by testing those probabilistic quantum programs with assertions. Existing quantum program assertion studies [237, 240], which mostly rely on empirical study, lack a rigorous theoretical foundation.

Potential and problem of projections: We observe that projection can be the key to address these issues due to its potential logical expressive power and unique mapping property. The logical expressive power of projection operators comes from the quantum logic by Birkhoff and von Neumann back in 1936 [242]. The logical connectives (e.g., conjunction and disjunction) of projection operators can be defined by the set operations on their corresponding closed subspaces of a Hilbert space. Moreover, projections naturally match the projective measurement, which may not affect the measured state when the state is in one of its basis states [243]. However, only those projective measurements with

a very limited set of projections can be directly implemented on a quantum computer due to the physical constraints on the measurement basis and measured qubit count, impeding the full utilization of the logical expressive power of projections.

To overcome all the problems mentioned above and fully exploit the potential of projections, we propose **Proq**, a projection-based runtime assertion for quantum programs. **First**, we employ projection operators to express the predicates in our runtime assertion. The logical expressive power of projection-based predicates allows us to assert much more types of states and enable more flexible assertion locations. **Second**, we define the semantics of our projection-based assertions by turning the projection-based predicates into corresponding projective measurements. Then the measurement in our assertion will not affect the tested state if the state satisfies the assertion predicate. This property leads to more efficient assertion checking and enables multi-assertion per execution. **Third**, we quantitatively show that after a sufficient number of testing executions with projection-based assertions, the semantics of the tested program can be guaranteed with a high confidence level. This result can serve as the theoretical foundation of quantum program testing with projection-based assertions. **Finally**, we consider the physical constraints on a quantum computer and introduce several transformation techniques, including *additional unitary transformation*, *combining projections*, and *using auxiliary qubits*, to make all projection-based assertions executable on a measurement-restricted quantum computer. We also propose *local projection*, which is a sound simplification of the original projections, to relax the constraints in the predicates for simplified assertion implementations.

The major contributions of this chapter can be summarized as follows:

1. We, first the time, propose to use projection operators to design runtime assertions that have strong logical expressive power and can be efficiently checked on a

quantum computer.

2. On the theory side, we prove that testing quantum programs with projection-based assertions is statistically effective in debugging or assuring the program semantics for both exact and approximate quantum programs.

3. On the practice side, we propose several assertion transformation techniques to simplify the assertion implementation and make our assertions physically executable on a measurement-restricted quantum computer.

4. Both theoretical analysis and experimental results show that our assertion outperforms existing quantum program assertions [237, 240] with much stronger expressive power, more flexible assertion location, fewer executions, and lower implementation overhead.

7.2 Preliminary

In this section, we introduce the necessary preliminary to help understand the proposed assertion scheme. We will start from our base quantum programming language and then introduce the projection operator as well as the hardware constraints considered in this chapter.

7.2.1 Quantum Programming Language

For simplicity of presentation, this work adopts the quantum **while**-language [244] to describe the quantum algorithms. This language is purely quantum without classical variables but this selection will not affect the generality since the quantum **while**-language, which has been proved to be universal [244], only keeps basic quantum computation elements that can be easily implemented by other quantum programming lan-

guages [159, 231, 92, 232, 233, 234, 235]. Thus, our assertion design and implementation based on this language can also be easily extended to other quantum programming languages

Definition 7.2.1 (Syntax [244]) *The quantum **while**-programs are defined by the grammar:*

$$\begin{aligned} S ::= \ & \textbf{skip} \mid S_1; S_2 \mid q := |0\rangle \mid \overline{q} := U[\overline{q}] \\ & \mid \textbf{if} \ \ (\square m \cdot M[\overline{q}] = m \rightarrow S_m) \ \ \textbf{fi} \\ & \mid \textbf{while} \ M[\overline{q}] = 1 \ \textbf{do} \ S \ \textbf{od} \end{aligned}$$

The language grammar is explained as follows. q represents a quantum variable while $\overline{q}$ means a quantum register, which consists of one or more variables with its corresponding Hilbert space denoted by $\mathcal{H}_{\overline{q}}$. $q := |0\rangle$ means that quantum variable q is initialized to be $|0\rangle$. $\overline{q} := U[\overline{q}]$ denotes that a unitary transformation U is applied to $\overline{q}$. Case statement $\textbf{if} \cdots \textbf{fi}$ means a quantum measurement M is performed on $\overline{q}$ to determine which subprogram S_m should be executed based on the measurement outcome m. The loop $\textbf{while} \cdots \textbf{od}$ means a measurement M with two possible outcomes $0, 1$ will determine whether the loop will terminate or the program will re-enter the loop body.

The *semantic function* of a quantum **while**-program S (denoted by $[\![S]\!]$) is a mapping from the program input state to its output state after executing program S. For example, $[\![S]\!](\rho)$ represents the output state of program S with input state ρ. A formal and comprehensive introduction to the semantics of quantum **while**-programs can be found in [245].

7.2.2 Projection and Projective Measurement

One type of quantum measurement of particular interest is the projective measurement because all measurements that can be physically implemented on quantum computers are projective measurements. We first introduce projections and then define the projective measurement.

For each closed subspace X of $\mathcal{H}$, we can define a projection P_X. Note that every $|\psi\rangle \in \mathcal{H}$ ($|\psi\rangle$ does not have to be normalized) can be written as $|\psi\rangle = |\psi_X\rangle + |\psi_0\rangle$ with $|\psi_X\rangle \in X$ and $|\psi_0\rangle \in X^\perp$ (the orthocomplement of X).

Definition 7.2.2 (Projection) *The projection $P_X : \mathcal{H} \mapsto X$ is defined by*

$$P_X|\psi\rangle = |\psi_X\rangle.$$

for every $|\psi\rangle \in \mathcal{H}$.

In the rest of this chapter, we denote P_X as P because there is a one-to-one correspondence between the closed subspaces of a Hilbert space and the projections in it. For simplicity, we do not distinguish a projection P from its corresponding subspace. Note that P is Hermitian ($P^\dagger = P$) and $P^2 = P$. If a pure state $|\psi\rangle$ (or a mixed state ρ) is in the corresponding subspace of a projection P, we have $P|\psi\rangle = |\psi\rangle$ ($P\rho P = \rho$). The **rank** of a projection P (denoted by $\operatorname{rank} P$) is defined by the dimension of its corresponding subspace.

Definition 7.2.3 (Projective measurement) *A projective measurement M is a quantum measurement in which all the measurement operators are projections ($0_\mathcal{H}$ is the zero*

operator on $\mathcal{H}$):

$$M = \{P_m\},\ \textit{where} \sum_m P_m = \mathcal{I}_{\mathcal{H}}\ \textit{and}\ P_m P_n = \begin{cases} P_m & \textit{if } m = n, \\ 0_{\mathcal{H}} & \textit{otherwise.} \end{cases}$$

Note that if a state $|\psi\rangle$ (or ρ) is in the corresponding subspace of P_m, then a projective measurement with observed outcome m will not change the state since:

$$|\psi_m\rangle = \frac{P_m |\psi\rangle}{\sqrt{\langle\psi| P_m^\dagger P_m |\psi\rangle}} = \frac{|\psi\rangle}{\sqrt{\langle\psi|\psi\rangle}} = |\psi\rangle\,, \quad \left(\text{resp. } \rho_m = \frac{P_m \rho P_m^\dagger}{\text{tr}\left(P_m^\dagger P_m \rho\right)} = \frac{\rho}{\text{tr}(\rho)} = \rho\right)$$

7.2.3 Projection-Based Predicates and Quantum Logic

In addition to defining projective measurements, projection operators can also define the predicates in quantum programming. We introduce the definition of projection-based predicates.

Definition 7.2.4 (Projections-based predicates) *Suppose P is a projection operator on $\mathcal{H}$ and its corresponding closed subspace is X. A state ρ is said to satisfy a predicate P (written $\rho \models P$) if* $\text{supp}(\rho) \subseteq X$, *where* $\text{supp}(\rho)$ *is the subspace spanned by the eigenvectors of ρ with non-zero eigenvalues. Note that $\rho \models P \implies P\rho = \rho$.*

Some quantum algorithms (e.g., qPCA [192]) are not exact and their program states may only approximately satisfy a projection-based predicate. We first introduce two concepts, trace distance D and fidelity F, to evaluate the distance between two states. Then we define the approximate satisfactory of projection-based predicates.

Definition 7.2.5 (Trace distance of states) *For two states ρ and σ, the trace distance D, which measures the "distinguishability" of two quantum states, between ρ and*

σ is defined as

$$D(\rho,\sigma)=\frac{1}{2}tr|\rho-\sigma|$$

where $tr|X| = tr\sqrt{X^\dagger X}$ and $\sqrt{X^\dagger X}$ refers to the positive square root which is unique because X is a density matrix which is Hermitian. Note that $0 \leq D(\rho,\sigma) \leq 1$ and $D(\rho,\sigma) = 0 \Leftrightarrow \rho = \sigma$. For two normalized states ρ and σ (pure states or density operators with trace 1), $D(\rho,\sigma) = 1 \Leftrightarrow \rho$ and σ are orthogonal. Trace distance is a metric and it satisfies the triangle inequality.

Definition 7.2.6 (Fidelity) *For two states ρ and σ, the fidelity F, which is not a metric but measures the "closeness" of two quantum states, between ρ and σ is defined as*

$$F(\rho,\sigma)=tr\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}$$

where $\sqrt{\rho}$ is the unique positive square root given by the spectral theorem (the same with the square root in the above definition). For example, suppose the spectrum decomposition of ρ is $\sum_i p_i|\psi_i\rangle\langle\psi_i|$, then $\sqrt{\rho}=\sum_i \sqrt{p_i}|\psi_i\rangle\langle\psi_i|$ (we have $p_i \geq 0$ since a state ρ must be a positive semi-definite operator.). Note that $0 \leq F(\rho,\sigma) \leq 1$ and $F(\rho,\sigma) = 1 \Leftrightarrow \rho = \sigma$. $F(\rho,\sigma) = 0 \Leftrightarrow \rho$ and σ (may not be normalized) are orthogonal. Note that fidelity does not satisfy the triangle inequality. A frequently used metric induced by fidelity is the arccos of fidelity and it satisfies the triangle inequality.

Definition 7.2.7 (Approximate satisfactory of projection-based predicates) *A state ρ is said to approximately satisfy (projective) predicate P with error parameter ϵ, written $\rho \models_\epsilon P$ if there exists a σ with the same trace such that $\sigma \models P$ and $D(\rho,\sigma) \leq \epsilon$.*

In the rest of this chapter, all predicates are projection-based predicates and we do not distinguish a predicate P, a projection P, and its corresponding closed subspace P. A quantum logic can be defined on the set of all closed subspaces of a Hilbert space [242].

Definition 7.2.8 (Quantum logic on the projections [242]) *Suppose $\mathcal{S}(\mathcal{H})$ is the set of all closed subspaces of Hilbert space $\mathcal{H}$. Then $(\mathcal{S}(\mathcal{H}), \wedge, \vee, ^{\perp})$ is an orthomodular lattice (or quantum logic). For any $P, Q \in \mathcal{S}(\mathcal{H})$, we define:*

$$P \wedge Q = P \cap Q,\ P \vee Q = \overline{\text{span}(P \cup Q)},\ P^{\perp} = \{|\psi\rangle \in \mathcal{H} : \langle\psi| P |\psi\rangle = 0\}$$

and the notations are defined as follows. Suppose T is a set in $\mathcal{H}$. Then $\text{span}(T)$ *is the subspace spanned by T, and $\overline{T}$ is the closure of T. That is, in this quantum logic, the logic operations on the predicates are defined by the set operations on their corresponding subspaces.*

7.2.4 Measurement-Restricted Quantum Computer

Although projective measurement has restricted all the measurement operators to be projection operators, most quantum computers which run on the well-adopted quantum circuit model [25] usually have more restrictions on the measurement.

First, they only support projective measurement in the **computational basis**. That is, only projective measurements with a specific set (which only contains all the computational basis states) of projection operators can be physically implemented. For example, such a projective measurement on n qubits can be described as $M = \{P_t\}$, where $P_t = |t\rangle\langle t|$ is the projection onto the 1-dimensional subspace spanned by the basis state $|t\rangle$, and t ranges over all n-bit strings; in particular, for a single qubit, this measurement is simply $M = \{P_0, P_1\}$ with $P_0 = |0\rangle\langle 0|$ and $P_1 = |1\rangle\langle 1|$.

Second, only projective measurements with projection operators of **special ranks** can be physically implemented. Suppose we have an n-qubit program with a 2^n-dimensional state space. After we measure one qubit, the state of that qubit will collapse to one of its basis states. The overall state space is reduced by half and becomes a 2^{n-1}-dimensional

space. A projection P with $\operatorname{rank} P = 2^{n-1}$ can be implemented by measuring one qubit. If k qubits are measured, the remaining space will have 2^{n-k} dimensions, and projections with $\operatorname{rank} P = 2^{n-k}$ can be implemented by measuring k qubits. In reality, we can only measure an integer number of qubits but cannot measure a fraction number of qubits. For an n-qubit system, we can measure $\{1, 2, \cdots, n\}$ qubits so that only projections with $\operatorname{rank} P \in \{2^{n-1}, 2^{n-2}, \cdots, 1\}$ can be directly implemented.

7.3 Projection-based assertion: design and theoretical foundations

The goal of this chapter is to provide a design of assertions which the programmers can insert in their quantum programs when testing and debugging their programs on a quantum computer. In particular, our design aims to achieve two objectives:

1. The assertions should have strong logical expressive power and can be efficiently checked.

2. The assertions should be executable on a quantum computer with restricted measurements.

In this section, we will focus on the first objective and introduce how to design quantum program assertions based on projection operators. We first discuss the reasons why projections are suitable for expressing predicates in a quantum program assertion. Then we formally define the syntax and semantics of a new projection-based **assert** statement. Finally, we rigorously formulate the theoretical foundations of program testing and debugging with projection-based assertions. We prove that running the assertion-injected program repeatedly can narrow down the potential location of a bug or assure that the semantics of the original program is close to what we expect.

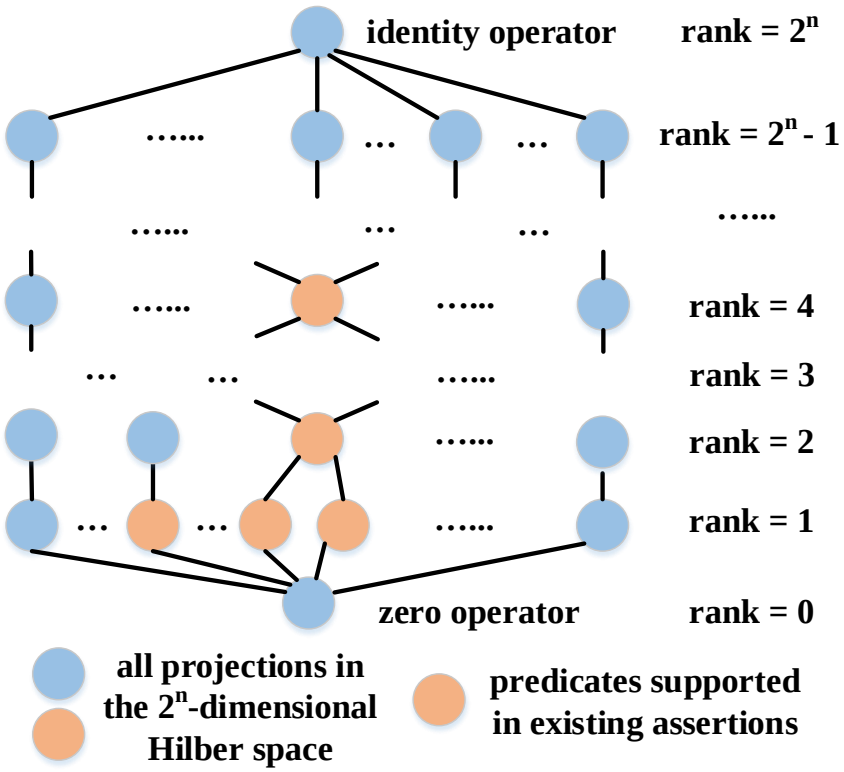


Figure 7.1: Logical expressive power comparison

7.3.1 Checking the Satisfaction of a Projection-Based Predicate

An assertion is a predicate at a point of a program. The key point of designing assertions for quantum programs is to first determine how to express predicates in the quantum scenario. Projection-based predicates has been used widely in static analysis and logic for quantum programming. For the first time, we employ projection-based predicates in runtime assertions for two reasons.

Strong logical expressive power: Figure 7.1 shows the orthomodular lattice based on all projections in a 2^n-dimensional Hilbert space and compares the logical expressive power of the predicates in existing assertions and the projections. All predicates expressed using a classical logical language in existing quantum program assertions [237, 240] can be represented by very few elements of special ranks in this lattice (detailed discussion is in

Section 7.5.1). But projections can naturally cover all elements in Figure 7.1. Therefore, projections have a much stronger expressive power compared with the classical logical language used in existing quantum assertions.

Efficient runtime checking: A quantum state ρ can be efficiently checked by a projection P because ρ will not be affected by the projective measurement with respect to P if it is in the subspace of P. We can construct a projective measurement $M = \{M_{\mathbf{true}} = P, M_{\mathbf{false}} = I - P\}$. When ρ is in the subspace of P, the outcome of this projective measurement is always "true" with probability of 1 and the state is still ρ. Then we know that ρ satisfies P without changing the state. When ρ is not in the subspace of P, which means that ρ does not satisfy P, the probability of outcome "true" or "false" in the constructed projective measurement is $tr(P\rho)$ or $1-tr(P\rho)$, respectively. Suppose we perform such procedure k times, the probability that we do not observe any "false" outcome is $tr(P\rho)^k$. Since $tr(P\rho) < 1$, this probability approaches 0 very quickly when $tr(P\rho)$ is not close to 1 and we can conclude if ρ satisfies P with high certainty within very few executions. Moreover, even if the state ρ is not in the subspace of P, the projective measurement with outcome "true" will change the incorrect state ρ to a correct state that is in the subspace of P so that the following execution after the assertion is still valid.

When $1 - tr(P\rho) < \epsilon$ and ϵ is small, it is possible that we do not observe any 'false' outcome in very few executions because the probability of observing a 'false' outcome is small. In this situation, we have the following two cases. **First**, the program itself has some real bugs that makes a tested state very close to what we expect. Given $1 - tr(P\rho) < \epsilon$, the trace distance (Definition 7.2.5) of the tested state ρ and at least one desired state is bounded by a small number $\epsilon + \sqrt{\epsilon(1-\epsilon)}$ if we realize that $\frac{P\rho P}{tr(P\rho P)}$ is a desired state since it satisfies P (in Lemma 7.3.1). It is almost impossible to prove that no such bugs ever exist but such a bug is not severe since the final state of the

program will also be close to the expected final state. This is because the trace distance is contractive under trace-nonincreasing quantum operations (the semantic function of any quantum program), i.e., $D([\![S]\!](\rho), [\![S]\!](\sigma)) \leq D(\rho, \sigma)$ where $[\![S]\!]$ is the semantic function of program S and D is the trace distance. Therefore, the trace distance between the final state of the tested program and the expected final state is also bounded by the small number $\epsilon + \sqrt{\epsilon(1-\epsilon)}$. Moreover, we have checked and confirmed that all types of bugs reported by Huang and Martonosi [236] (the only systematic report about bugs in real quantum programs to the best of our knowledge) can make $tr(P\rho)$ significantly smaller than 1. Therefore, checking a projection-based predicate is effective for these known quantum program bugs. **Second**, the program itself is not an exact quantum program and its correct program states are supposed to only approximately satisfy the predicates. We will prove that projection-based assertions can still test and debug such approximate quantum programs later in Section 7.3.4.

7.3.2 Assertion Statement: Syntax and Semantics

We have demonstrated the advantages of using projections as predicates. Now we add a new runtime assertion statement to the quantum **while**-language grammar.

Definition 7.3.1 (Syntax of the assertion) *The **syntax** of the quantum assertion is defined as:*

$$\mathbf{assert}(\overline{q}; P)$$

where $\overline{q} = q_1, ..., q_n$ is a collection of quantum variables and P is a projection in the state space $\mathcal{H}_{\overline{q}}$.

As the original quantum **while**-language is already universal, we define the semantics

of the new assertion statement using the quantum **while**-language. An auxiliary notation **abort** is employed to denote that the program terminates immediately and reports the termination location. The formal semantics of **abort** is $[\![\mathbf{abort}]\!](\rho) = 0_{\mathcal{H}}$ for all any input state ρ [245]. Intuitively, this definition means we do not have any quantum state after **abort**.

Definition 7.3.2 (Semantics) *The **semantics** of the new assertion statement is defined as*

$$\begin{aligned} \mathbf{assert}(\overline{q}; P) \equiv \mathbf{if}\ & M_P[\overline{q}] = m_0 \rightarrow\ \mathbf{skip} \\ \square\ & m_1 \rightarrow\ \mathbf{abort} \\ \mathbf{fi}\ & \end{aligned}$$

where $M_P = \{M_{m_0} = P, M_{m_1} = I_{\mathcal{H}_{\overline{q}}} - P\}$.

The semantics of the assertion statement is explained as follows: We construct a projective measurement $M_P = \{M_{m_0} = P, M_{m_1} = I_{\mathcal{H}_{\overline{q}}} - P\}$ based on the projection operator P in the assertion. We apply this measurement of the corresponding qubit collection $\overline{q}$. If the measurement result is m_0, which means that the tested state is in the closed subspace of P, then we continue the execution of program without doing anything because the tested state satisfies the predicate in the assertion. If the measurement result is m_1, which means the tested state is not in the closed subspace of P, the program will terminate and report the termination location. Then we can know that the state at this location does not satisfy the corresponding predicate. Here the semantics of **abort** is slightly different from the original one because we need to report the termination location.

7.3.3 Statistical Effectiveness of Testing and Debugging with Projection-Based Assertions

As with classical program testing, quantum program testing can show the presence of bugs, lowering the risking of remaining bugs, but cannot assure the behavior of all possible computation. One testing execution cannot even check the program behavior thoroughly for one input due to the intrinsic randomness of quantum systems. Therefore, multiple executions are required to test a quantum program with one input. In this section, we show that, for a program with projection-based assertions and one specific input, running it repeatedly for enough times can locate bugs or statistically assure the behavior of the program under the specific input with high confidence.

We consider a quantum program S. When the programmers try to test a program with assertions, multiple assertions could be injected so that a potential bug could be revealed as early as possible. Suppose we insert l assertions whose predicates are $P_1, P_2, \ldots, P_l$ (P_l is the predicate for the final state). We define that a bug-free standard program S_{std} is a program that can satisfy all the predicates throughout the program. We will show that after running the program with assertion inserted for a couple of times, we can locate the incorrect program segment if an error message occurs or conclude that output of the tested program S and the standard program S_{std} (under a specific input ρ) is close. We first formally define a debugging scheme for a quantum program.

Definition 7.3.3 *A debugging scheme for S is a new program S' with assertions being*

added between consecutive subprograms S_i and S_{i+1}:

$$\begin{aligned} S' \equiv\ & S_1;\ \mathbf{assert}(\overline{q}_1; P_1); \\ & S_2;\ \mathbf{assert}(\overline{q}_2; P_2); \\ & \cdots; \\ & S_{l-1};\ \mathbf{assert}(\overline{q}_{l-1}; P_{l-1}); \\ & S_l;\ \mathbf{assert}(\overline{q}_l; P_l) \end{aligned}$$

where $\overline{q}_i$ is the collection of quantum variables and P_i is a projection on $\mathcal{H}_{\overline{q}_i}$ for all $0 < i \leq l$.

In this debugging scheme, assertions are injected after every statement while this may not be necessary in practice. The assertion injection is flexible, and the programmers can inject assertions only on those locations where they hope to have assertions.

Now we discuss the statistical properties of this debugging scheme. A program segment S_i is considered to be correct if its output satisfies the predicate P_i when its input satisfied P_{i-1} as specified by the assertions. We show that running the program S' (defined in Definition 7.3.3) with assertions injected could effectively check the program by proving that the tested program S and a standard program S_{std} will have a similar semantic function under the tested input state. A quantitative and formal description of the effectiveness of our debugging scheme is illustrated by the following theorem.

Theorem 7.3.1 (Effectiveness of debugging scheme) *Suppose we repeatedly execute S' (with l assertions) with input ρ and collect all the error messages.*

1. *If an error message occurs in* $\mathbf{assert}(\overline{q}_i; P_i)$*, then subprogram S_i is not correct, i.e., with the input satisfying precondition P_{i-1}, after executing S_i, the output can violate postcondition P_i.*

2. *If no error message is reported after executing S' for k times ($k \gg l^2$), program S is close to the bug-free standard program; more precisely, with confidence level* 95%,

   (a) *the confidence interval of* $\min_{S_{\text{std}}} D\left([\![S]\!](\rho), [\![S_{\text{std}}]\!](\rho)\right)$ *is* $\left[0, \frac{0.9l+\sqrt{l}}{\sqrt{k}}\right]$,

   (b) *the confidence interval of* $\max_{S_{\text{std}}} F\left([\![S]\!](\rho), [\![S_{\text{std}}]\!](\rho)\right)$ *is* $\left[\cos\frac{0.9l+\sqrt{l}}{\sqrt{k}}, 1\right]$,

   where the minimum (maximum) is taken over all bug-free standard programs S_{std} that satisfy all assertions with input ρ. Here D is the trace distance (Definition 7.2.5) and F is the fidelity (Definition 7.2.6).

Moreover, within one testing execution, if the program s_m is not correct but $\mathbf{assert}(\overline{q}_m; P_m)$ *is passed, then follow-up assertion* $\mathbf{assert}(\overline{q}_{m+1}; P_{m+1})$ *is still effective in checking the program S_{m+1}.*

By Theorem 7.3.1, we conclude that we can use projection-based assertions to test a quantum program and find the locations of potential bugs with the proposed debugging scheme. When an error message occurs in $\mathbf{assert}(\overline{q}_i; P_i)$, we can know that there is at least one bug in the program segment S_i. Although we could not directly know how the bug happens nor repair a bug, our approach can help with debugging in practice, by narrowing down the potential location of a bug from the entire program to one specific program segment. After applying the proposed debugging scheme, programmers can manually investigate the target program segment to finally find the bug more quickly without searching in the entire program. If we could not have any error message after running the assertion checking program S' for a sufficiently large number of times, we can conclude that the semantics of the original program S for the tested input is at least close to what we expected (specified by the assertions) with high confidence. In the proposed theorem, we require $k \gg l^2$ because we hope to achieve the confidence level 95%. In practice, the number of test executions can be reduced with a lower confidence level.

Only one input tested: It can be noticed that only one input is tested when using the proposed debugging scheme in Theorem 7.3.1. However, in classical program testing, we usually prepare a large number of testing cases to increase the testing thoroughness. Here we argue that considering one input is already useful in testing many quantum programs because the input information of many practical quantum algorithms (e.g., Shor's algorithm [3], Grover algorithm [7], VQE algorithm [48], HHL algorithm [191]) are only encoded in the operations and the input state is always a trivial state $|00\cdots00\rangle$. Consequently, we do not need to check different inputs when testing these quantum algorithms. Checking for one specific input $\rho = |00\cdots00\rangle\langle 00\cdots00|$ will be sufficient.

7.3.4 Testing and Debugging Approximate Quantum Programs

We have shown that projection-based assertions can be used to check exact quantum programs but there are also other quantum algorithms (e.g., qPCA [192], Grover's search [7], Quantum Phase Estimation [25]) of which the correct program states sometimes only approximately satisfy a projection. We generalize Theorem 7.3.1 by adding error parameters on all the program segments to represent the approximation throughout the program, and prove that we can still locate bugs or conclude about the semantics of the tested program with high confidence by checking projection-based assertions.

We first study how much a state ρ is changed after a projective measurement by proving a special case of the gentle measurement lemma [246] with projections. The result is slightly stronger than the original one [246] under the constraint of projection.

Lemma 7.3.1 (Gentle measurement with projections) *For projection P and density operator ρ, if $\mathrm{tr}(P\rho) \geq 1-\epsilon$, then we have*

1. $D\left(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\right) \leq \epsilon + \sqrt{\epsilon(1-\epsilon)}$, *$D$ is the trace distance (Definition 7.2.5).*

2. $F\left(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\right) \geq \sqrt{1-\epsilon}$, *$F$ is the fidelity (Definition 7.2.6).*

Suppose a state ρ satisfies P with error ϵ, then $\mathrm{tr}(P\rho) \geq 1-\epsilon$ which ensures that, applying the projective measurement $M_P = \{M_{\mathbf{true}} = P,\ M_{\mathbf{false}} = I - P\}$, we have the outcome "true" with probability at least $1-\epsilon$. Moreover, if the outcome is "true" and ϵ is small, the post-measurement state $\frac{P\rho P}{\mathrm{tr}(P\rho P)}$ is close to the original state ρ in the sense that their trace distance is at most $\epsilon + \sqrt{\epsilon(1-\epsilon)}$.

Consider a program $S = S_1; S_2; \cdots; S_l$ with l inserted assertions $\mathbf{assert}(\overline{q}_m, P_m)$ after each segments S_m for $1 \leq m \leq l$. Unlike the exact algorithms, here each program segment S_m is considered to be correct if its input satisfies P_{m-1}, then its output approximately satisfies P_m with error parameter ϵ_m. The following theorem states that the debugging scheme defined in Definition 7.3.3 is still effective for approximate quantum programs.

Theorem 7.3.2 (Effectiveness of debugging approximate quantum programs) *Assume that all ϵ_m are small ($\epsilon_m \ll 1$). Execute S' for k times ($k \gg l^2$) with input ρ, and we count k_m for the occurrence of error message for assertion $\mathbf{assert}(\overline{q}_m, P_m)$.*

1. *The 95% confidence interval of real ϵ_m is $[w_m^-, w_m^+]$. Thus, with confidence 95%, if $\epsilon_m < w_m^-$, S_m is incorrect; and if $\epsilon_m > w_m^+$, we conclude S_m is correct. Here, w_m^-, w_m^+ and w_m^c are $B\left(\alpha, k_m + 1, k - \sum_{i=1}^{m} k_i\right)$ with $\alpha = 0.025, 0.975$ and 0.5 respectively, where $B(P, A, B)$ is the Pth quantile from a beta distribution with shape parameters A and B.*

2. *If no segment appears to be incorrect, i.e., all $\epsilon_m \geq w_m^-$, then after executing the original program S with input ρ, the output state σ approximately satisfies P_l with error parameter δ, i.e., $\sigma \models_\delta P_l$, where $\delta = \sum_{m=1}^{l} \sqrt{w_m^c} + \sqrt{\sum_{m=1}^{l} (\sqrt{w_m^+} - \sqrt{w_m^c})^2}$.*

With this theorem, we can test and debug approximate quantum programs by counting the number of occurrences of the error messages from different assertions. If the

observed assertion checking failure frequency is significantly higher or lower than the expected error parameter of a program segment, we can conclude that this program segment is correct or incorrect with high confidence. If all program segments appear to be correct, we can conclude that the final output of the original program approximately satisfies the last predicate within a bounded error parameter.

7.3.5 An Example of Using the Effectiveness Theorems

We give an example to illustrate using Theorem 7.3.1 and 7.3.2 in practical debugging. Suppose a bug-free standard program S has two qubits p, q:

$$S \equiv p := Z[p]; p := R_y(\pi/2)[p]; p, q := \mathrm{CNOT}[p, q]$$

where $R_y(\theta)$ is the rotation about the Y-axis, i.e.,

$$R_y(\theta) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

When the input is $|00\rangle_{pq}$, the program produces a Bell state $|\Phi^+\rangle_{pq} = \frac{|00\rangle_{pq}+|11\rangle_{pq}}{\sqrt{2}}$. Now we consider a real program written by a careless programmer:

$$S_{\mathrm{real}} \equiv p := Z[p]; p := R_y(1.7)[p]; p, q := \mathrm{CNOT}[p, q]$$

which can be decomposed into two segments, $S_{\mathrm{real},1} \equiv p := Z[p]; p := R_y(1.7)[p]$, $S_{\mathrm{real},2} \equiv p, q := \mathrm{CNOT}[p, q]$. The careless programmer understands the program correctly and knows that if the input is $|00\rangle_{pq}$, the state after the first two unitary transformations on p should be $|+\rangle_q = \frac{|0\rangle_q+|1\rangle_q}{\sqrt{2}}$ and the final state should be Bell state. Thus he adds two

assertions to S_{real}:

$$S'_{\text{real}} \equiv p := Z[p]; p := R_y(1.7)[p]; \textbf{assert}(p; P_1); p, q := \text{CNOT}[p, q]; \textbf{assert}(p, q; P_2)$$

where $P_1 = |+\rangle\langle+|$ and $P_2 = |\Phi^+\rangle\langle\Phi^+|$.

Theoretically it can be proved that S'_{real} is close to but not equal to the bug free program in the sense that:

$$D([\![S]\!](|00\rangle_{pq}), [\![S_{\text{real}}]\!](|00\rangle_{pq})) = 0.065, \quad F([\![S]\!](|00\rangle_{pq}), [\![S_{\text{real}}]\!](|00\rangle_{pq})) = 0.9979$$

Note that the final state is a pure state and thus all bug-free standard programs are equivalent to S when input is $|00\rangle_{pq}$.

We then consider three different testing cases. In the first two cases the programmer thinks the program should be accurate and in the last case the program is considered to be approximate.

Case 1. The programmer executes S'_{real} for 1000 times and surprisingly no error is reported. What can he learn from the result? By Theorem 7.3.1 (2) with $k = 1000$ and $l = 2$, we have: with confidence level 95%,

1. the confidence interval of $D([\![S]\!](|00\rangle_{pq}), [\![S_{\text{real}}]\!](|00\rangle_{pq}))$ is $[0, 0.101]$;

2. the confidence interval of $F([\![S]\!](|00\rangle_{pq}), [\![S_{\text{real}}]\!](|00\rangle_{pq}))$ is $[0.995, 1]$;

We can see that the real trace distance 0.065 and fidelity 0.9979 are indeed in the corresponding intervals $[0, 0.101]$ and $[0.995, 1]$, respectively.

Case 2. The programmer executes S'_{real} for 10,000 times and 37 errors are reported by the first assertion $\textbf{assert}(p; P_1)$ but no error is reported by the second assertion. Now what can he conclude?

1. By Theorem 7.3.1 (1), the first segment $S_{\text{real},1} \equiv p := Z[p]; p := R_y(1.7)[p]$ is not correct; there must be some bugs;

2. By Theorem 7.3.1 (2), the second segment $S_{\text{real},2} \equiv p, q := \text{CNOT}[p, q]$ is very likely to be true in the sense that: there exists a bug free standard program $S_1; S_2$ with two segments S_1 and S_2 such that: with confidence 95% ($k = 9963, l = 1$)

   (a) the confidence interval of $D([\![S_1; S_2]\!](|00\rangle_{pq}), [\![S_1; S_{\text{real},2}]\!](|00\rangle_{pq}))$ is $[0, 0.019]$;

   (b) the confidence interval of $F([\![S_1; S_2]\!](|00\rangle_{pq}), [\![S_1; S_{\text{real},2}]\!](|00\rangle_{pq}))$ is $[0.99982, 1]$;

   In fact, segment $S_{\text{real},2}$ is exactly correct.

Case 3. Now, the programmer thinks that the program is approximate and a small error is acceptable. In detail, for both segments $S_{\text{real},1}$ and $S_{\text{real},2}$, he selects the same acceptable error parameters $\epsilon_1 = \epsilon_2 = 0.01$.

Fact: A straightforward calculation gives the *real value* of $\epsilon_{1,\text{real}} = 0.0042$ and $\epsilon_{2,\text{real}} = 0$, and the output $[\![S_{\text{real}}]\!](|00\rangle_{pq})$ approximately satisfies P_2 with error parameter 0.065.

Consider the execution results in the **case 2** above. According to Theorem 7.3.2 (1), we first calculate the 95% confidence intervals of real ϵ_1 and ϵ_2 are $[w_1^-, w_1^+]$ and $[w_2^-, w_2^+]$ where parameters are:

$$w_1^- = 0.0027, \quad w_1^+ = 0.0051, \quad w_1^c = 0.0038,$$
$$w_2^- = 0.00000, \quad w_2^+ = 0.00037, \quad w_2^c = 0.00007$$

Obviously, $\epsilon_1 > w_1^+$ and $\epsilon_2 > w_2^+$ and thus with confidence 95%, both of the segments are acceptable. Now, by Theorem 7.3.2 (2), we further know that the output $[\![S_{\text{real}}]\!](|00\rangle_{pq})$ approximately satisfies P_2 with error parameter $\delta = 0.0845 > 0.065$. All of these confidence intervals and parameters given by our theorems are consistent with the Fact.

7.4 Transformation techniques for implementation on quantum computers

In the previous section, we have illustrated how to test and debug a quantum program with the proposed projection-based assertions and proved its effectiveness. However, there exists a gap that makes the assertions not directly executable on a real quantum computer. There are two reasons for this incompatibility as explained in the following:

1. **Limited measurement basis:** Not all projective measurements are supported on a quantum computer and only projective measurement that lie in the computational basis can be physically implemented directly with today's quantum computing underlying technologies (in Section 7.2.4). But there is no restriction on the projection operator P in the assertions so that P could be arbitrary projection operator in the Hilbert space. For example, $P = |+\rangle\langle+| = \frac{1}{2}(|0\rangle + |1\rangle)(\langle 0| + \langle 1|)$ is on a basis of $\{|+\rangle, |-\rangle\}$. These assertions with projections not in the computational basis cannot be directly executed on a real quantum computer.

2. **Dimension mismatch:** A projective measurement, which is already in the computational basis, may still not be executable because the number of dimensions of its corresponding subspace cannot be directly implemented by measuring an integer number of qubits. For an n-qubit system, only projections with $\operatorname{rank} P \in \{2^{n-1}, 2^{n-2}, \cdots, 1\}$ can be directly implemented (in Section 7.2.4). But the rank of the projection in an assertion can be any integer between 0 and 2^n. For example, a projection in a 2-qubit system can be $P = |00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 11|$. An assertion with such projection cannot be directly implemented because $\operatorname{rank} P = 3$ and $\operatorname{rank} P \notin \{2, 1\}$.

In this section, we introduce several transformation techniques to overcome these two

obstacles. The basic idea is to use the conjunction of projections and auxiliary qubit to convert the target assertion into some new assertions without dimension mismatch. Then some additional unitary transformations are introduced to rotate the basis in the projective measurements. These transformation techniques can be employed to compile the assertions and make a quantum program with projection-based assertions executable on a measurement-restricted real quantum computer.

7.4.1 Additional Unitary Transformation

We first resolve the limited measurement basis problem without considering the dimension mismatch problem. Suppose the assertion $\mathbf{assert}(\overline{q}; P)$ we hope to implement is over n qubits, that is, $\overline{q} = q_1, q_2, \cdots, q_n$, each of q_i is a single qubit variable. We assume that $\operatorname{rank} P = 2^m$ for some integer m with $0 \leq m \leq n$ so there is no dimension mismatch problem.

Proposition 7.4.1 *For projection P with $\operatorname{rank} P = 2^m$, there exists a unitary transformation U_P such that (here $I_{q_i} = I_{\mathcal{H}_{q_i}}$):*

$$U_P P U_P^\dagger = Q_{q_1} \otimes Q_{q_2} \otimes \cdots \otimes Q_{q_n} = \bigotimes_{i=1}^{n} Q_{q_i} \triangleq Q_P,$$

where $Q_{q_i} \in \{|0\rangle_{q_i}\langle 0|, |1\rangle_{q_i}\langle 1|, I_{q_i}\}$ for each $1 \leq i \leq n$. U_P and Q_P can be obtained immediately after we diagonalize the projection P.

We call the pair (U_P, Q_P) an **i**mplementation in the **c**omputational **b**asis (ICB for short) of $\mathbf{assert}(\overline{q}; P)$. ICB is not unique in general. According to this proposition, we have the following procedure to implement $\mathbf{assert}(\overline{q}; P)$:

1. Apply U_P on $\overline{q}$;

2. Check Q_P in the following steps: For each $1 \leq i \leq n$, if $Q_{q_i} = |0\rangle_{q_i}\langle 0|$ or $|1\rangle_{q_i}\langle 1|$, then measure q_i in the computational basis to see whether the outcome k is consistent with Q_{q_i}; that is, $Q_{q_i} = |k\rangle_{q_i}\langle k|$. If all outcomes are consistent, go ahead; otherwise, we terminate the program with an error message;

3. Apply $U_P^\dagger$ on $\overline{q}$.

The transformation for $\mathbf{assert}(\overline{q}; P)$ with ICB (U_P, Q_P) when $\operatorname{rank} P = 2^m$ is:

$$\mathbf{assert}(\overline{q}; P) \equiv\ \overline{q} := U_P[\overline{q}];\ \mathbf{assert}(\overline{q}; Q_P);\ \overline{q} := U_P^\dagger[\overline{q}]$$

Since Q_P is now a projection in the computational basis, $\mathbf{assert}(\overline{q}; Q_P)$ can be executed by Definition 7.3.2 and the projective measurement constructed by Q_P is executable.

Example 7.4.1 *Given a two-qubit register $\overline{q} = q_1, q_2$, if we want to test whether it is in the Bell state (maximally entangled state) $|\Phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, we can use the assertion $\mathbf{assert}(\overline{q}; P = |\Phi\rangle\langle\Phi|)$. We apply proposition 7.4.1 and diagonalize the projection P.*

$$P = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)\frac{1}{\sqrt{2}}(\langle 00| + \langle 11|) = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

$$U_P P U_P^\dagger = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & \frac{-1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 \end{bmatrix} \cdot P \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = Q_P$$

The generated diagonal matrix Q_P is actually $|0\rangle_{q_1}\langle 0| \otimes |0\rangle_{q_2}\langle 0|$ and the unitary U_P can

be implemented with first an CNOT *gate and then a* H *gate. Therefore, we have:*

$$H[q_1]\text{CNOT}[q_1, q_2] \cdot P \cdot \text{CNOT}[q_1, q_2]H[q_1] = |0\rangle_{q_1}\langle 0| \otimes |0\rangle_{q_2}\langle 0|$$

we can first apply CNOT *gate on* $\overline{q}$ *and* H *gate on* q_1*, then measure* q_1 *and* q_2 *in the computational basis. If both outcomes are "0", we apply* H *on* q_1 *and* CNOT *on* $\overline{q}$ *again to recover the state; otherwise, we terminate the program and report that the state is* not *Bell state* $|\Phi\rangle$*.*

Unitary generation: The generated unitary may not be the exact inverse of the preceding operations. Suppose the desired state is $|\psi\rangle$, the only requirement for the unitary is $U : |\psi\rangle \mapsto |0\rangle$, and the output of U under input states other than $|\psi\rangle$ does not matter. This may allow simpler implementations of U. In general, it is not clear whether the generated U will be simpler or more complex when decomposing the U into basic single- and two-qubit gates. We demonstrate an in-principle unitary generation process but the actual implementation can be further optimized with techniques like tensor network, decision diagram, symbolic execution, etc. The scalability will be determined by the optimization on the matrix calculation and storage. This is left as future work.

7.4.2 Combining Assertions

The first transformation technique solves the measurement basis issue but does not consider the dimension mismatch issue, which will be addressed by the next two techniques. We first consider an assertion $\textbf{assert}(\overline{q}; P)$ in which the projection P has $\operatorname{rank} P \leq 2^{n-1}$ and $\operatorname{rank} P \neq 2^m$ with some integer m. We have the following proposition to decompose this assertion into multiple sub-assertions that do not have dimension mismatch issues.

Proposition 7.4.2 *For projection P with* $\operatorname{rank} P \leq 2^{n-1}$*, there exist projections $P_1, P_2, \cdots, P_l$ satisfying* $\operatorname{rank} P_i = 2^{n_i}$ *for all $1 \leq i \leq l$ where $n_i \in \mathbb{N}$, such that $P = P_1 \cap P_2 \cap \cdots \cap P_l$.*

Essentially, this way works for our scheme because conjunction can be defined in Birkhoff-von Neumann quantum logic. Theoretically, $l = 2$ is sufficient; but in practice, a larger l may allow us to choose simpler P_i for each $i \leq l$.

Using the above proposition, to implement $\mathbf{assert}(\overline{q}; P)$, we may sequentially apply $\mathbf{assert}(\overline{q}; P_1)$, $\mathbf{assert}(\overline{q}; P_2)$, $\cdots$, $\mathbf{assert}(\overline{q}; P_l)$. Suppose (U_{P_i}, Q_{P_i}) is an ICB of $\mathbf{assert}(\overline{q}; P_i)$ for $1 \leq i \leq l$, we have the following scheme to implement $\mathbf{assert}(\overline{q}; P)$:

1. Set counter $i = 1$;
2. If $i = 1$, apply U_{P_1}; else if $i = l$, apply $U_{P_l}^\dagger$ and return; otherwise, apply $U_{P_{i-1}}^\dagger U_{P_i}$;
3. Check Q_{P_i}; $i := i + 1$; go to step (2).

The transformation for $\mathbf{assert}(\overline{q}; P)$ when $\operatorname{rank} P \leq 2^{n-1}$ is:

$$\mathbf{assert}(\overline{q}; P) \equiv \mathbf{assert}(\overline{q}; P_1);\ \mathbf{assert}(\overline{q}; P_2);\ \ldots\ldots;\ \mathbf{assert}(\overline{q}; P_l)$$

where $\operatorname{rank} P_i = 2^{n_i}$ and $P = P_1 \cap P_2 \cap \cdots \cap P_l$. There are no dimension mismatch issues for these sub-assertions and they can be further transformed with Proposition 7.4.1.

Example 7.4.2 *Given register $\overline{q} = q_1, q_2, q_3$, how to implement* $\mathbf{assert}(\overline{q}; P)$ *where* $P = |00\rangle_{q_1q_2}\langle 00| \otimes I_{q_3} + |111\rangle_{q_1q_2q_3}\langle 111|$*? We first observe that $P = P_1 \cap P_2$ where*

$$P_1 = (|00\rangle_{q_1q_2}\langle 00| + |11\rangle_{q_1q_2}\langle 11|) \otimes I_{q_3},$$
$$P_2 = |00\rangle_{q_1q_2}\langle 00| \otimes I_{q_3} + |100\rangle_{q_1q_2q_3}\langle 100| + |111\rangle_{q_1q_2q_3}\langle 111|.$$

with following properties:

$$\text{CNOT}[q_1, q_2] \cdot P_1 \cdot \text{CNOT}[q_1, q_2] = I_{q_1} \otimes |0\rangle_{q_2}\langle 0| \otimes I_{q_3}$$
$$\text{Toffoli}[q_1, q_3, q_2] \cdot P_2 \cdot \text{Toffoli}[q_1, q_3, q_2] = I_{q_1} \otimes |0\rangle_{q_2}\langle 0| \otimes I_{q_3}.$$

Therefore, we can implement $\textbf{assert}(\overline{q}; P)$ *by:*

- *Apply* $\text{CNOT}[q_1, q_2]$*;*
- *Measure* q_2 *and check if the outcome is "0"; if not, terminate and report the error message;*
- *Apply* $\text{CNOT}[q_1, q_2]$ *and then* $\text{Toffoli}[q_1, q_3, q_2]$*;*
- *Measure* q_2 *and check if the outcome is "0"; if not, terminate and report the error message;*
- *Apply* $\text{Toffoli}[q_1, q_3, q_2]$.

7.4.3 Auxiliary Qubits

The previous two techniques can transform projections with $\text{rank}\, P \leq 2^{n-1}$ but those projections with $\text{rank}\, P > 2^{n-1}$ remain unresolved. This case cannot be handled with the conjunction of a group of sub-assertions directly because logic conjunction can only result in a subspace with fewer dimensions (compared with the original subspaces of the projections in the sub-assertions). The possible subspace of a projection in an n-qubit system has at most 2^{n-1} dimensions since we have to measure at least one qubit. As a result, we cannot use logic conjunction to construct a projection with $\text{rank}\, P > 2^{n-1}$. The logic disjunction of projections with small ranks can create a subspace of larger size but it is not suitable for assertion design. As discussed at the beginning of Section 7.3,

it is expected that a correct state is not changed during the assertion checking. But if a state ρ at the tested program location is in a space of a large size, applying a projective measurement with a small subspace may destroy the tested state when the tested state is not in the small subspace, leading to inefficient assertion checking.

We propose the third technique, introducing auxiliary qubits, to tackle this problem. Actually, one auxiliary qubit is already sufficient. Suppose we have an n-qubit program with a 2^n-dimensional state space. If we add one additional qubit into this system, the system now has $n+1$ qubits with a 2^{n+1}-dimensional state space. This new qubit is not in the original quantum program so it is not involved in any assertions for the program. A projection P with $2^{n-1} < \operatorname{rank} P \leq 2^n$ can thus be implemented in the new 2^{n+1}-dimensional space using the previous two transformation techniques. One auxiliary qubit is sufficient because the projection P is originally in a 2^n-dimensional space and we always have $\operatorname{rank} P \leq 2^n$.

The transformation for $\mathbf{assert}(\overline{q}; P)$ when $\operatorname{rank} P > 2^{n-1}$ is:

$$\mathbf{assert}(\overline{q}; P) \equiv a := |0\rangle;\ \mathbf{assert}(a, \overline{q}; |0\rangle_a\langle 0| \otimes P)$$

where a is the new auxiliary qubit. Noting that $\operatorname{rank}(|0\rangle_a\langle 0| \otimes P) = \operatorname{rank} P \leq 2^n$.

Example 7.4.3 *Given register $\overline{q} = q_1, q_2$, we aim to implement $\mathbf{assert}(\overline{q}; P)$ where $P = |0\rangle_{q_1}\langle 0| \otimes I_{q_2} + |11\rangle_{q_1 q_2}\langle 11|$.*

We may have the decomposition $|0\rangle_a\langle 0| \otimes P = P_0 \cap P_1$, where

$$P_0 = |0\rangle_a\langle 0| \otimes I_{\overline{q}},\ P_1 = |00\rangle_{aq_1}\langle 00| \otimes I_{q_2} + |011\rangle_{aq_1q_2}\langle 011| + |100\rangle_{aq_1q_2}\langle 100|,$$

and P_1 can be implemented with one additional unitary transformation:

$$\text{Fredkin}[q_2, a, q_1] \cdot P_1 \cdot \text{Fredkin}[q_2, a, q_1] = I_a \otimes |0\rangle_{q_1}\langle 0| \otimes I_{q_2}.$$

where the Fredkin *gate is defined in Chapter 2.*

Note that P_0 automatically holds since the auxiliary qubit a is already initialized to $|0\rangle$, we only need to execute:

- *Introduce auxiliary qubit a, initialize it to $|0\rangle$;*
- *Apply* Fredkin$[q_2, a, q_1]$*;*
- *Measure q_1 and check if the outcome is "0"; if not, terminate and report the error message;*
- *Apply* Fredkin$[q_2, a, q_1]$*; free the auxiliary qubit a.*

7.4.4 Local Projection: Trading Checking Accuracy for Implementation Efficiency

As shown in the three transformation techniques, we need to manipulate the projection operators and some unitary transformations to implement an assertion. These transformations can be easily automated when n is small or the tested state is not fully entangled (which means we can deal with them part by part directly). For projections over multiple qubits, it is possible that the qubits are highly entangled. Asserting such entangled states accurately requires non-trivial efforts to find the unitary transformations and we need to manipulate operators of size 2^n for an n-qubit system in the worst case, which makes it hard to fully automate the transformations on a classical computer when n is large. Such scalability issue widely exists in quantum computing research that

requires automation on a classical computer, e.g., simulation [247], compiler optimization and its verification [248, 249], formal verification of quantum circuits [232, 250].

In our runtime projection-based assertion checking, we propose **local projection** technique to mitigate this scalability problem (not fully resolve it) by designing assertions that only manipulate and observe part of a large system without affecting a highly entangled state over multiple qubits. These assertions, which are only applied on a smaller number of qubits, could always be automated easily with simplified implementations but the assertion checking constraints are also relaxed. This approach is inspired by the quantum state tomography via local measurements [251, 252, 253], a common approach in quantum information science.

We first introduce the notion of partial trace to describe the state (operator) of a subsystem. Let $\overline{q}_1$ and $\overline{q}_2$ be two disjoint registers with corresponding state Hilbert space $\mathcal{H}_{\overline{q}_1}$ and $\mathcal{H}_{\overline{q}_2}$, respectively. The partial trace over $\mathcal{H}_{\overline{q}_1}$ is a mapping $\mathrm{tr}_{\overline{q}_1}(\cdot)$ from operators on $\mathcal{H}_{\overline{q}_1} \otimes \mathcal{H}_{\overline{q}_2}$ to operators in $\mathcal{H}_{\overline{q}_2}$ defined by: $\mathrm{tr}_{\overline{q}_1}(|\phi_1\rangle_{\overline{q}_1}\langle\psi_1| \otimes |\phi_2\rangle_{\overline{q}_2}\langle\psi_2|) = \langle\psi_1|\phi_1\rangle \cdot |\phi_2\rangle_{\overline{q}_2}\langle\psi_2|$ for all $|\phi_1\rangle, |\psi_1\rangle \in \mathcal{H}_{\overline{q}_1}$ and $|\phi_2\rangle, |\psi_2\rangle \in \mathcal{H}_{\overline{q}_2}$ together with linearity. The partial trace $\mathrm{tr}_{\overline{q}_2}(\cdot)$ over $\mathcal{H}_{\overline{q}_2}$ can be defined dually. Then, the local projection is defined as follows:

Definition 7.4.1 (Local projection) *Given* **assert**$(\overline{q}; P)$*, a local projection* $P_{\overline{q}'}$ *over* $\overline{q}' \subseteq \overline{q}$ *is defined as:*

$$P_{\overline{q}'} = \mathrm{supp}\left(\mathrm{tr}_{\overline{q}\setminus\overline{q}'}(P)\right).$$

Proposition 7.4.3 (Soundness of local projection) *For any* $\rho \models P$*, we have* $\rho \models P_{\overline{q}'} \otimes I_{\overline{q}\setminus\overline{q}'}$.

This simplified assertion with $P_{\overline{q}'}$ will lose some checking accuracy because some states not in P may be included in $P_{\overline{q}'}$, allowing false positives. However, by taking the partial trace, we are able to focus on the subsystem of $\overline{q}'$. The implementation of

$\mathbf{assert}(\overline{q}'; P_{\overline{q}'})$ can partially test whether the state satisfies P. Moreover, the number of qubits in $\overline{q}'$ is smaller, and we only need to manipulate small-size operators when implementing $\mathbf{assert}(\overline{q}'; P_{\overline{q}'})$. We have the following implementation strategy which is essentially a trade-off between assertion implementation efficiency and checking accuracy:

- Find a sequence of local projection $P_{\overline{q}_1}, P_{\overline{q}_2}, \cdots, P_{\overline{q}_l}$ of $\mathbf{assert}(\overline{q}; P)$;
- Instead of implementing the original $\mathbf{assert}(\overline{q}; P)$, we sequentially apply $\mathbf{assert}(\overline{q}_1; P_{\overline{q}_1})$, $\mathbf{assert}(\overline{q}_2; P_{\overline{q}_2})$, $\cdots$, $\mathbf{assert}(\overline{q}_l; P_{\overline{q}_l})$.

Example 7.4.4 *Given register $\overline{q} = q_1, q_2, q_3, q_4$, we want to check if the state is the superposition of the following states:*

$$|\psi_1\rangle = |+\rangle_{q_1}|111\rangle_{q_2q_3q_4}, \quad |\psi_2\rangle = |000\rangle_{q_1q_2q_3}|-\rangle_{q_4},$$
$$|\psi_3\rangle = \frac{1}{\sqrt{2}}|0\rangle_{q_1}\left(|00\rangle_{q_2q_3} + |11\rangle_{q_2q_3}\right)|1\rangle_{q_4}.$$

To accomplish this, we may apply $\mathbf{assert}(\overline{q}; P)$ with $P = \mathrm{supp}\left(\sum_{i=1}^{3}|\psi_i\rangle\langle\psi_i|\right)$. However, projection P is highly entangled which prevents efficient implementation. But if we only observe part of the system, we will the following local projections:

$$P_{q_1q_2} = \mathrm{tr}_{q_3q_4}(P) = |0\rangle_{q_1}\langle 0| \otimes I_{q_2} + |11\rangle_{q_1q_2}\langle 11|,$$
$$P_{q_2q_3} = \mathrm{tr}_{q_1q_4}(P) = |00\rangle_{q_2q_3}\langle 00| + |11\rangle_{q_2q_3}\langle 11|,$$
$$P_{q_3q_4} = \mathrm{tr}_{q_1q_2}(P) = |00\rangle_{q_3q_4}\langle 00| + |11\rangle_{q_3q_4}\langle 11|.$$

To avoid implementing $\mathbf{assert}(\overline{q}, P)$ directly, we may use $\mathbf{assert}(q_1, q_2;\ P_{q_1q_2})$, $\mathbf{assert}(q_2, q_3;\ P_{q_2q_3})$, and $\mathbf{assert}(q_3, q_4; P_{q_3q_4})$ instead. Though these assertions do not fully characterize the required property, their implementation requires only relatively low cost, i.e., each of them only acts on two qubits.

In the next example, we show that a local projection may detect some bugs but not all of them.

Example 7.4.5 *Consider the following program with three qubits p, q, r:*

$$S \equiv p := H[p]; q := H[q]; r := H[r]; p, q := \mathrm{CZ}[p, q]; p, r := \mathrm{CZ}[p, r]; q := H[q]; r := H[r]$$

The program S produces a GHZ state $\frac{|000\rangle_{pqr}+|111\rangle_{pqr}}{\sqrt{2}}$ if the input state is $|000\rangle_{pqr}$. Suppose a full description of GHZ state involving three qubits is somewhat difficult to implement due to complexity; instead, we choose the assertion $\mathbf{assert}(p, q; P)$ inserted at the end of the program where projection P is the local projection of GHZ state, i.e.,

$$P = |00\rangle_{pq}\langle 00| + |11\rangle_{pq}\langle 11| = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

*which can be implemented by 1). Apply CNOT on p, q; 2). Measure qubit q and check if the measurement output is 0; 3). Apply CNOT on p, q. However, it is not difficult to realize that $\mathbf{assert}(p, q; P)$ is not a perfect description of GHZ state; for example, the following *bug* program S_{bug} where the final unitary transformation $r := H[r]$ is missing:*

$$S_{\mathrm{bug}} \equiv p := H[p]; q := H[q]; r := H[r]; p, q := \mathrm{CZ}[p, q]; p, r := \mathrm{CZ}[p, r]; q := H[q]$$

also passes $\mathbf{assert}(p, q; P)$ with input $|000\rangle_{pqr}$. However, the program S_{bug} in fact produces:

$$\frac{|00\rangle_{pq}|+\rangle_r + |11\rangle_{pq}|-\rangle_r}{\sqrt{2}}$$

if the input is $|000\rangle_{pqr}$*. On the other hand, consider another *bug* program* S'_{bug} *where the unitary transformation* $q := H[q]$ *is missing:*

$$S'_{\text{bug}} \equiv p := H[p]; q := H[q]; r := H[r]; p, q := \text{CZ}[p, q]; p, r := \text{CZ}[p, r]; r := H[r]$$

Then it can be shown that if the input state is $|000\rangle_{pqr}$*,* $\mathbf{assert}(p, q; P)$ *is not passed and we are able to conclude that* S'_{bug} *is not the desired program using Theorem 7.3.1.*

7.4.5 Summary

To the best of our knowledge, the three transformations constitute the first working flow to implement an arbitrary projective measurement on measurement-restricted quantum computers. A complete flow to make an assertion $\mathbf{assert}(\overline{q}; P)$ (on n qubits) executable is summarized as follows:

1. If $\text{rank}\, P > 2^{n-1}$, initialize one auxiliary qubit a, let $n := n+1$ and $P := |0\rangle_a\langle 0| \otimes P$ (Section 7.4.3);

2. If $\text{rank}\, P \notin \{2^{n-1}, 2^{n-2}, \cdots, 1\}$, find a group of sub-assertions (Section 7.4.2);

3. Apply unitary transformations to implement the assertion or sub-assertions (Section7.4.1).

The three transformations cover all possible cases for projections with different ranks and basis. Therefore, all projection-based assertions can finally be executed on a quantum computer. The local projection technique can be applied when an assertion is hard to be implemented (automatically). Whether to use local projection is optional.

7.5 Overall Comparison

In this section, we will have an overall comparison among Proq and two other quantum program assertions in terms of assertion coverage (i.e., the expressive power of the predicates, the assertion locations) and debugging overhead (i.e., the number of executions, additional gates, measurements).

Baseline: We use the statistical assertions (Stat) [237] and the QEC-inspired assertions (QECA) [240] as the baseline assertion schemes. To the best of our knowledge, they are the only published quantum program assertions till now. Stat employs a classical statistical test on the measurement results to check if a state satisfies a predicate. QECA introduces auxiliary qubits to indirectly measure the tested state.

7.5.1 Coverage Analysis

Assertion predicates: Proq employs projections which are able to represent a wide variety of predicates. However, both Stat and QECA only support three types of assertions: classical assertion, superposition assertion, and entanglement assertion. The expressive power difference has been summarized in Figure 7.1. For Stat, all these three types of assertions can be considered as $\operatorname{rank} P = 1$ special cases in Proq. The corresponding projections are

$P = |t\rangle \langle t|$, t ranges over all n-bit strings for classical assertion

(suppose n qubits are asserted)

$P = |{+}{+}{+}\ldots\rangle \langle {+}{+}{+}\ldots|$ for superposition assertion

$P = (|00\ldots0\rangle + |11\ldots1\rangle)(\langle 00\ldots0| + \langle 11\ldots1|)$ for entanglement assertion

Stat's language does not support other types of states. QECA supports arbitrary 1-qubit states (these states can naturally cover the classical assertion and superposition assertion in Stat), some special 2-qubit entanglement states, and some special 3-qubit entangle states. These states can be considered as some rank $P = 1, 2, 4$ special cases in Proq, respectively. So all QECA assertions are covered in Proq. Moreover, the implementations of QECA assertions are all designed manually without a systematic assertion implementation generation so they cannot be extended to more cases directly. The expressive power of the assertions in Proq, which can support many more complicated cases as introduced in Section 7.3 and 7.4, is much more than that of the baseline schemes.

Assertion locations: Thanks to the expressive power of the predicates in Proq, projection-based assertions can be injected at more locations with complex intermediate states in a program. The baseline schemes can only inject assertions at those locations with states that can be checked with the very limited types of assertions. If the baseline schemes insert assertions at locations with other types of states, their assertions will always return negative results since the predicates in their assertions are not correct. Therefore, the number of potential assertion injection locations of Proq is much larger than that of the baseline schemes.

7.5.2 Overhead Analysis

It is not easy to directly perform a fair overhead comparison between Proq and the baseline because Proq supports many more types of predicates as explained above. We first discuss the impact of this difference in assertion coverage in practical debugging.

Assertion coverage impact: Proq support assertions that cannot be implemented in Stat and QECA. These assertions will help locate the bug more quickly. When inserting assertions in a tested program, Proq assertions can always be injected closer to a potential

bug because Proq allows more assertion injection locations. The potential bug location can then be narrowed down to a smaller program segment, which makes it easier for the programmers to manually search for the bug after an error message is reported.

Then we remove the assertion coverage difference by assuming all the assertions are within the three types of assertions supported in all assertion schemes.

Assertion checking overhead: We mainly discuss two aspects of the assertion checking overhead, 1) the number of assertion checking program executions and 2) the numbers of additional unitary transformations (quantum gates) and measurements to implement each of the assertions.

1. **Compare with Stat:** Stat's approach is quite different from Proq. It only injects measurements to directly measure the tested states without any additional transformations.

   (a) **number of executions:** The classical assertion, the first supported assertion type in Stat, is equivalent to the corresponding one in Proq. The tested state remains unchanged if it is the expected state. However, when checking for superposition states and entanglement states, the number of assertion checking program executions will be large because 1) Stat requires a large number of samples for each assertion to reconstruct an amplitude distribution over multiple basis states, and 2) the measurements will always affect the tested states so that only one assertion can be checked per execution. It is not yet clear how many executions are required since the statistical properties of checking Stat assertions are not well studied. The original Stat paper [237] claims to apply chi-square test and contingency table analysis (with no details about the testing process) on the measurement results collection of each assertion but it does not provide the numbers of required executions to achieve an acceptable confidence level for different assertions over different

numbers of qubits, which makes it hard to directly compare the checking overhead (no publicly available code). We believe the number of executions will be large at least when the tested state is in a superposition state over multiple computational basis states. For example, the superposition assertion, which checks for the state $|{+}{+}{+}\ldots\rangle$ in an n-qubit system, requires $k \gg 2^n$ testing executions to observe a uniform distribution over all 2^n basis states.

(b) **number of gates and measurements:** For an assertion (any type) in Stat, it only requires n measurements on n qubits in assertion checking but it may need to be executed many times as explained above. For the corresponding assertions in Proq, a classical assertion requires n measurements (the same with Stat, e.g., Assertion A_0 in Figure 7.3). A superposition assertion requires additionally $2n$ H gates (e.g., Assertion A_1 in Figure 7.3). An entanglement assertion requires additionally $2(n-1)$ CNOT gates and 2 H gates (e.g., Assertion A_2 in Figure 7.3). Proq only needs few additional gates (linear to the number of qubits) for the commonly supported assertions.

2. **Compare with QECA:** All QECA assertions are equivalent to their corresponding Proq assertions. Therefore, QECA has the same checking efficiency and supports multi-assertion per execution if we only consider those QECA-supported assertions. The statistical properties (Theorem 7.3.1 and 7.3.2) we prove can also be directly applied to QECA. So **the number of the assertion checking executions is the same** for QECA and Proq. The difference between QECA and Proq is that the actual assertion implementation in terms of quantum gates and measurements. The **implementation cost of Proq is lower** than that of QECA because QECA always need to couple the auxiliary qubits with existing qubits. We will have concrete data of the assertion implementation cost comparison between

Proq and QECA later in a case study in Section 7.6.1.

7.6 Case Studies: Runtime Assertions for Realistic Quantum Algorithms

In this section, we perform case studies by applying projection-based assertions on two famous sophisticated quantum algorithms, the Shor's algorithm [3] and the HHL algorithm [191]. For Shor's algorithm, we focus on a concrete example of its quantum order finding subroutine. The assertions are simple and can be supported by the baselines, which allows us to compare the resource consumption between Proq and the baseline and show that Proq could generate low overhead runtime assertions. For HHL algorithm, instead of just asserting a concrete circuit implementation, we will show that Proq could have non-trivial assertions that cannot be supported by the baselines. In these non-trivial assertions, we will illustrate how the proposed techniques, i.e., combining assertions, auxiliary qubits, local projection, can be applied in implementing the projections. Numerical simulation confirms that Proq assertions can work correctly.

7.6.1 Shor's Algorithm

Shor's algorithm was proposed to factor a large integer [3]. Given an integer N, Shor's algorithm can find its non-trivial factors within $O(poly(log(N)))$ time. In this chapter, we focus on its quantum order finding subroutine and omit the classical part which is assumed to be correct.

$p := |0\rangle^{\otimes n}$;
while $M[p] = 1$ **do**
$p := |0\rangle^{\otimes n}$; $q := |0\rangle^{\otimes n}$; **assert**$(p, q; A_0)$; $p := H^{\otimes n}[p]$; **assert**$(p, q; A_1)$;
$p, q := U_f[p, q]$; **assert**$(p, q; A_2)$; $p := \mathrm{QFT}^{-1}[p]$; **assert**$(p, q; A_3)$;
od

Figure 7.2: Shor's algorithm program with assertions. The projections A_0, A_1, A_2, A_3 are defined in Section 7.6.1.

Shor's Algorithm Program

Figure 7.2 shows the program of the quantum subroutine in Shor's algorithm with the injected assertions in the quantum **while**-language. Briefly, it leverages Quantum Fourier Transform (QFT) to find the period of the function $f(x) = a^x \bmod N$ where a is a random number selected by a preceding classical subroutine. The transformation U_f, the measurement M, and the result set R are defined as follows:

$$U_f : \ |x\rangle_p |0\rangle_q \mapsto |x\rangle_p |a^x \bmod N\rangle_q, M = \Big\{ M_0 = \sum_{r \in R} |r\rangle \langle r|, M_1 = I - M_0 \Big\},$$

$$R = \{r \mid \gcd(a^{\frac{r}{2}} + 1, N) \text{ or } \gcd(a^{\frac{r}{2}} - 1, N) \text{ is a nontrivial factor of N}\}$$

For the measurement, the set R consists of the expected values that can be accepted by the follow-up classical subroutine. For a comprehensive introduction, please refer to [25].

Assertions for a Concrete Example

The circuit implementation we select for the subroutine is for factoring $N = 15$ with the random number $a = 11$ [254]. Based on our understanding of Shor's algorithm, we have four assertions, A_0, A_1, A_2, and A_3, as shown in Figure 7.2. Figure 7.3 shows the final assertion-injected circuit with 5 qubits. The circuit blocks labeled with **assert** are

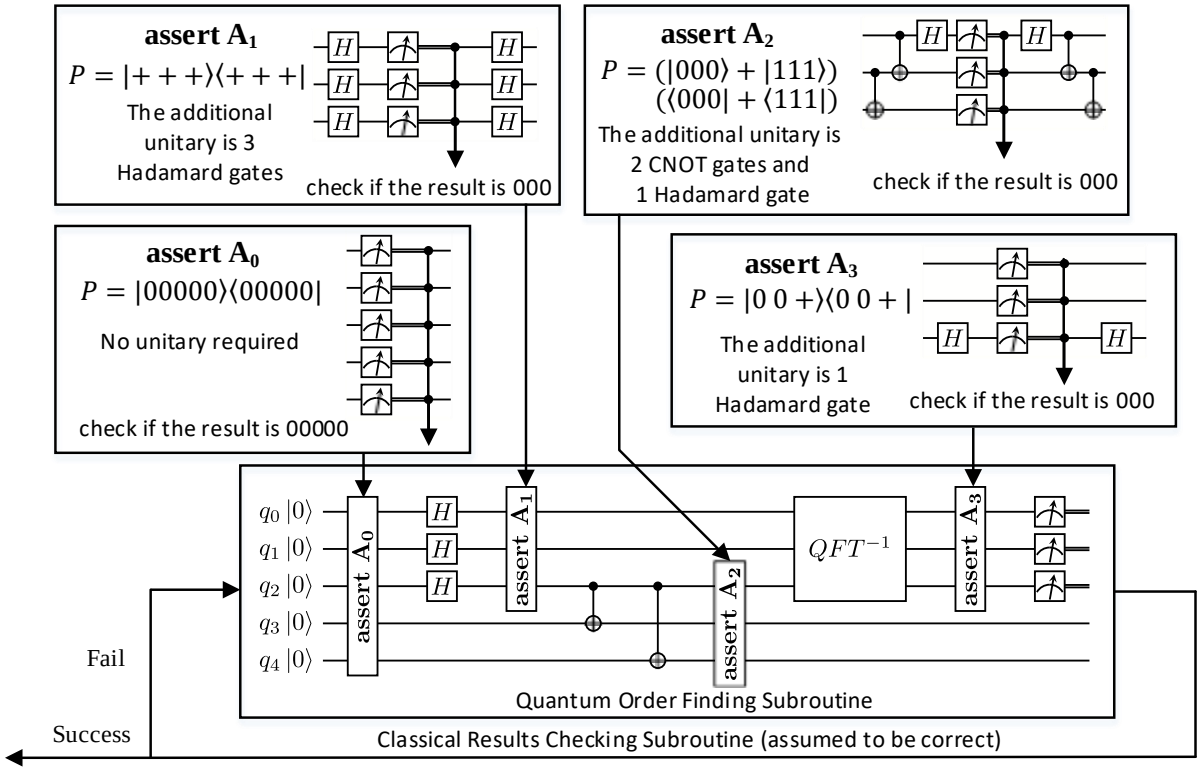


Figure 7.3: Assertion-injected circuit implementation for Shor's algorithm with $N = 15$ and $a = 11$

for the four assertions with four projections defined as follows:

$$A_0 = |00000\rangle_{0,1,2,3,4}\langle 00000|;\ A_1 = |{+}{+}{+}\rangle_{0,1,2}\langle {+}{+}{+}| \otimes |00\rangle_{3,4}\langle 00|;$$

$$A_2 = |{+}{+}\rangle_{0,1}\langle {+}{+}| \otimes (|000\rangle + |111\rangle)_{2,3,4}(\langle 000| + \langle 111|);$$

$$A_3 = (|000\rangle + |001\rangle)_{0,1,2}(\langle 000| + \langle 001|) \otimes (|00\rangle + |11\rangle)_{3,4}(\langle 00| + \langle 11|).$$

We detail the implementation of the assertion circuit blocks in the upper half of Figure 7.3. For each assertion, we list its projection, the additional unitary transformations, with the complete implementation circuit diagram. For A_1, A_2, and A_3, since the qubits not fully entangled, we only assert part of the qubits without affecting the results. The unitary transformations are decomposed into CNOT gates and single-qubit gates, which

is the same with QECA for a fair comparison.

Assertion Comparison

Similar to Section 7.5, we first compare the coverage of assertions for this realistic algorithm and then detail the implementation cost in terms of the number of additional gates, measurements, and auxiliary qubits.

Assertion coverage: All four assertions are supported in Stat and Proq. For QECA, A_0, A_1, and A_3 are covered but A_2 is not yet supported even if it is an entanglement state. The reason is that the QECA assertion only supports 3-qubit entanglement states with $rankP = 4$ but A_2 is a 3-qubit entanglement state with $rankA_2 = 1$.

We compare the circuit cost when implementing the assertions between Proq and QECA. Stat is not included because we have already discussed the implementation difference in Section 7.5.2 and it is not clear how many executions are required for Stat.

Table 7.1 shows the implementation cost of the three assertions supported by both Proq and QECA. In particular, we compare the number of H gates, CNOT gates, measurements, and auxiliary qubits. It can be observed that Proq uses no CNOT gates and auxiliary qubits for the three considered assertions, while QECA always needs to use additional CNOT gates and auxiliary qubits. This reason is that QECA always measures auxiliary qubits to indirectly probe the qubit information. So that additional CNOT gates are always required to couple the auxiliary qubits with existing qubits. This design

Table 7.1: Detailed assertion implementation cost comparison between Proq and QECA [240]

	A_0		A_1		A_3	
# of	Proq	QECA	Proq	QECA	Proq	QECA
H	0	0	6	6	2	2
CNOT	**0**	**5**	**0**	**6**	**0**	**4**
Measure	5	5	3	3	3	3
Aux. Qbit	**0**	**1**	**0**	**1**	**0**	**1**

$p := |0\rangle^{\otimes n};\ q := |0\rangle^{\otimes m};\ r := |0\rangle;$
while $M[r] = 1$ **do**
    assert$(p, r; P)$;
    $q := |0\rangle^{\otimes m};\ q := U_b[q];\ p := H^{\otimes n}[p]; p, q := U_f[p, q];\ p := \mathrm{QFT}^{-1}[p];$ **assert**$(p; S)$;
    $p, r := U_c[p, r];\ p := \mathrm{QFT}[p]; p, q := U_f^{\dagger}[p, q];\ p := H^{\otimes n}[p];$ **assert**$(p, q, r; R)$;
od
assert$(q; Q)$;

Figure 7.4: HHL algorithm program with assertions

significantly increases the implementation cost when comparing with Proq.

To summarize, we demonstrate the complete assertion-injected circuit for a quantum program of Shor's algorithm and the implementation details of the assertions. We compare the implementation cost between Proq and QECA to show that Proq has lower cost for the limited assertions that are supported by both assertion schemes.

7.6.2 HHL Algorithm

In the first example of Shor's algorithm, we focus the assertion implementation on a concrete circuit example and compare against other assertions due to the simplicity of the intermediate states. In the next HHL algorithm example, we will have non-trivial assertions that are not supported in the baselines and demonstrate how to apply the techniques introduced in Section 7.4.

The HHL algorithm was proposed for solving linear systems of equations [191]. Given a matrix A and a vector $\vec{b}$, the algorithm produces a quantum state $|x\rangle$ which is corresponding to the solution $\vec{x}$ such that $A\vec{x} = \vec{b}$. It is well-known that the algorithm offers up to an exponential speedup over the fastest classical algorithm if A is sparse and has a low condition number κ.

HHL Program

The HHL algorithm has been formulated with the quantum **while**-language in [255] and we adopt the assumptions and symbols there. Briefly speaking, A is a Hermitian and full-rank matrix with dimension $N = 2^m$, which has the diagonal decomposition $A = \sum_{j=1}^{N} \lambda_j |u_j\rangle\langle u_j|$ with corresponding eigenvalues λ_j and eigenvectors $|u_j\rangle$. We assume for all j, $\delta_j = \frac{\lambda_j t_0}{2\pi} \in \mathbb{N}^+$ and set $T = 2^n = \lceil \max_j \delta_j \rceil$, where t_0 is a time parameter to perform unitary transformation U_f. Moreover, the input vector $\vec{b}$ is presumed to be unit and corresponding to state $|b\rangle$ with the linear combination $|b\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle$. It is straightforward to find the solution state $|x\rangle = c \sum_{j=1}^{N} \frac{\beta_j}{\lambda_j} |u_j\rangle$ where c is for normalization.

The HHL program has three registers p, q, r which are n, m, 1-qubit systems and used as the control system, state system, and indicator of while loop, respectively. For detailed definitions of U_b, U_f, QFT, and the measurement M, please refer to [255, 191].

Debugging Scheme for HHL Program

We introduce the debugging scheme for the HHL program shown in Figure 7.4. The projections P, Q, S, R are defined as follows:

$$P = |0\rangle_p\langle 0| \otimes |0\rangle_r\langle 0|;\ Q = |x\rangle_q\langle x|;\ S = \text{supp}\left(\sum_{j=1}^{N} |\delta_j\rangle_p\langle \delta_j|\right)$$

$$R = |0\rangle_p\langle 0| \otimes (|x\rangle_q\langle x| \otimes |1\rangle_r\langle 1| + I_q \otimes |0\rangle_r\langle 0|).$$

Projection R is across all qubits while P is focused on register p, r and Q is focused on the output register q. These projections can be implemented using the techniques introduced in Section 7.4; more precisely:

1. Implementation of **assert**$(p, r; P)$:

   measure register p and r directly to see if the outcomes are all "0";

2. Implementation of **assert**$(q; Q)$:

   apply U_x on q; (additional unitary transformation in Section 7.4.1)

   measure register q and check if the outcome is "0";

   apply $U_x^\dagger$ on q;

3. Implementation of **assert**$(p, q, r; R)$:

   measure register p directly to see if the outcome is "0";

   introduce an auxiliary qubit a, initialize it to $|0\rangle$; (auxiliary qubit in Section 7.4.3)

   apply U_x on q and U_R on r, q, a;

   measure register a and check if the outcome is "0"; (combining assertions in Section 7.4.2)

   apply $U_R^\dagger$ on r, q, a and $U_x^\dagger$ on q;

where U_x is defined by $U_x|x\rangle = |0\rangle$ and U_R is defined by

$$U_R|1\rangle_r\langle 1| \otimes |i\rangle_q\langle i| \otimes |k\rangle_a\langle k| = |1\rangle_r\langle 1| \otimes |i\rangle_q\langle i| \otimes |k \oplus 1\rangle_a\langle k \oplus 1|$$

for $i \geq 1$ and $k = 0, 1$ and unchanged otherwise.

We need to pay more attention to **assert**$(p; S)$. The most accurate predicate here is

$$S' = \sum_{j,j'=1}^{N} \beta_j \overline{\beta}_{j'} |\delta_j\rangle_p\langle\delta_{j'}| \otimes |u_j\rangle_q\langle u_{j'}| \otimes |0\rangle_r\langle 0|$$

which is a highly entangled projection over register p and q. As discussed in Section 7.4.4, in order to avoid the hardness of implementing S', we introduce $S = \mathrm{supp}(\mathrm{tr}_{q,r}(S'))$ which is the local projection of S' over p. Though **assert**$(p; S)$ is strictly weaker than original **assert**$(p, q, r; S')$, it can be efficiently implemented and partially test the state.

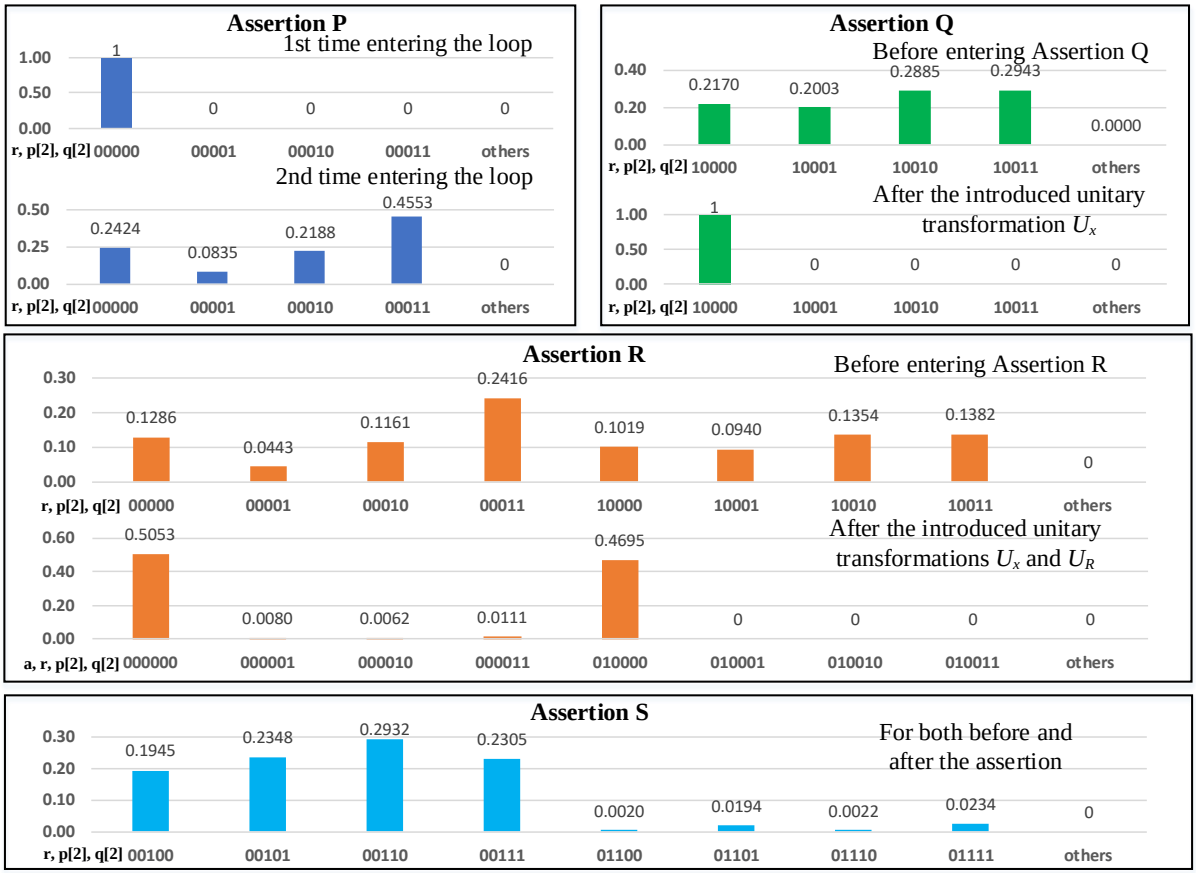


Figure 7.5: Numerical simulation results for the states around the assertions in HHL algorithm

Numerical Simulation Results

For illustration, we choose $m = n = 2$ as an example. Then the matrix A is 4×4 matrix and b is 4×1 vector. We first randomly generate four orthonormal vectors for $|u_j\rangle$ and then select δ_j to be either 1 or 3. Such configuration will demonstrate the applicability of all four techniques in Section 7.4. Finally, A and b are generated as

follows.

$$A = \begin{bmatrix} 1.951 & -0.863 & 0.332 & -0.377 \\ -0.863 & 2.239 & -0.011 & -0.444 \\ 0.332 & -0.011 & 1.301 & -0.634 \\ -0.377 & -0.444 & -0.634 & 2.509 \end{bmatrix}, b = \begin{bmatrix} -0.486 \\ -0.345 \\ -0.494 \\ -0.633 \end{bmatrix}$$

Assertion coverage: We have four assertions, labeled P, Q, R, and S, for the HHL program. Only P is for a classical state and supported by the Stat and QECA. Q, R, and S are more complex and not supported by the baseline assertions.

Figure 7.5 shows the amplitude distribution of the states during the execution of the four assertions and each block corresponds to one assertion. Since our experiments are performed in simulation, we can directly obtain the state vector $|\psi\rangle$. The X-axis represents those basis states of which the amplitudes are not zero. The Y-axis is the probability of the measurement outcome. Each histogram represents the probability distribution across different computational basis states. This probability is be calculated by $\|\langle\psi|x\rangle\|^2$, where $|x\rangle$ is the corresponding basis state. The texts over the histograms represent the program locations where we record each of the states. For example, in '**Assertion Q**' block, we show that the state vector has non-zero amplitudes on multiple basis states. But after applying the unitary transformation Assertion Q, the state vector only has non-zero amplitudes on one basis state.

Assertion P is at the beginning of the loop body. The predicate is $P = |000\rangle_{r,p}\langle 000|$, which means that the quantum registers r and p should always be in state $|0\rangle$ and $|00\rangle$, respectively, at the beginning of the loop body. Figure 7.5 shows that when the program enters the loop D at the first and second time, the assertion is satisfied and the quantum registers r and p are 0.

Assertion Q is at the end of the program. Figure 7.5 shows that there are non-zero amplitudes at 4 possible measurement outcomes at the assertion location. But after the

applied unitary transformation, the only possible outcome is 10000. Such an assertion is hard for Stat and QECA to describe but it is easy to define this assertion using projection in Proq.

Assertion R is at the end of the loop body. Figure 7.5 confirms that the basis states with non-zero amplitudes are in the subspace defined by the projection in assertion R. Its projection implementation involves the techniques of combining assertions and using auxiliary qubits. Such complex predicates cannot be defined in Stat and QECA while Proq can implement and check it.

Assertion S is in the middle of the loop body. At this place the state is highly entangled as mentioned above and directly implementing this projection will be expensive. We employ the local projection technique in Section 7.4.4. Since δ_js are selected to be either 1 or 3, the projection S becomes $|01\rangle_p\langle 01| + |11\rangle_p\langle 11|$. This simple form of local projection that can be easily implemented. Figure 7.5 confirms that the tested highly entangled state is not affected in this local projective measurement.

To summarize, we design four assertions for the program of HHL algorithm. Among them, only P can be defined in Stat and QECA. The remaining three assertions, which cannot be defined in Stat or QECA, demonstrate that Proq assertions can better test and debug realistic quantum algorithms.

7.7 Discussion

Program testing and debugging have been investigated for a long time because it reflects the practical application requirements for reliable software. Compared with its counterpart in classical computing, quantum program testing and debugging are still at a very early stage. Even the basic testing and debugging approaches (e.g., assertions) are not yet available or well-developed for quantum programs. This work made efforts

towards practical quantum program runtime testing and debugging through studying how to design and implement effective and efficient quantum program assertions. Specifically, we select projections as predicates in our assertions because of the logical expressive power and efficient runtime checking property. We prove that quantum program testing with projection-based assertion is statistically effective. Several techniques are proposed to implement the projection under machine constraints. To the best of our knowledge, this is the first runtime assertion scheme for quantum program testing and debugging with such flexible predicates, efficient checking, and formal effectiveness guarantees. The proposed assertion technique would benefit future quantum program development, testing, and debugging.

Although we have demonstrated the feasibility and advantages of the proposed assertion scheme, several future research directions can be explored as with any initial research.

Projection implementation optimization: We have shown that our assertion-based debugging scheme can be implemented with several techniques in Section 7.3 and demonstrated concrete examples in Section 7.6. However, further optimization of the projection implementation is not yet well studied. One assertion can be split into several sub-assertions, but different sub-assertion selections would have different implementation overhead. We showed that one auxiliary qubit is enough but employing more auxiliary qubits may yield fewer sub-assertions. For the circuit implementation of an assertion, the decomposition of the assertion-introduced unitary transformations can be optimized for several possible objectives, e.g., gate count, circuit depth. A systematic approach to generate optimized assertion implementations is thus important for more efficient assertion-based quantum program debugging in the future.

More efficient checking: Assertions for a complicated highly entangled state may require significant effort for its precise implementation. However, the goal of assertions is

to check if a tested state satisfies the predicates rather than to prove the correctness of a program. It is possible to trade-in checking accuracy for simplified assertion implementation by relaxing the constraints in the predicates. Local projection can be a solution to approximate a complex projective measurement as we discussed in Section 7.4.4 and demonstrated in one of the assertions for the HHL algorithm in Section 7.6. However, the degree of predicate relaxation and its effect on the robustness of the assertions in realistic erroneous program debugging need to be studied. Other possible directions, like non-demolition measurement [256], are also worth exploring.

7.8 Related Work

This chapter explores runtime assertion schemes for testing and debugging a quantum program on a quantum computer. In particular, the efficiency and effectiveness of our assertions come from the application of projection operators. In this section, we first introduce other existing runtime quantum program testing schemes, which are the closest related work, and then briefly discuss other quantum programming research involving projection operators.

7.8.1 Quantum Program Assertions

Recently, two types of assertions have been proposed for debugging on quantum computers. Huang and Martonosi proposed quantum program assertions based on statistical tests on classical observations [237]. For each assertion, the program executes from the beginning to the place of the injected assertion followed by measurements. This process is repeated many times to extract the statistical information about the state. The advantage of this work is that, for the first time, assertion is used to reveal bugs in realistic quantum programs and help discover several bug patterns. But in this debugging

scheme, each time only one assertion can be tested due to the destructive measurements. Therefore, the statistical assertion scheme is very time consuming. Proq circumvents this issue by choosing to use projective assertions.

Liu et al. further improved the assertion scheme by proposing dynamic assertion circuits inspired by quantum error correction [240]. They introduce ancilla qubits and indirectly collect the information of the qubits of interest. The success rate can also be improved since some unexpected states can be detected and corrected in the noisy scenarios. However, their approach requires manually designed transformation circuits and cannot be directly extended to more general cases. Their transformation circuits rely on ancilla qubits, which will increase the implementation overhead as discussed in Section 7.6.1.

Moreover, both of these assertion schemes can only inspect very few types of states that can be considered as some special cases of our proposed projection-based assertions, leading to limited applicability. In summary, our assertion and debugging schemes outperform these two existing assertion schemes mentioned above in terms of expressive power, flexibility, and efficiency.

7.8.2 Quantum Programming Language Research with Projections

Projection operators have been used in logic systems and static analysis for quantum programs. All projections in (the closed subspaces of) a Hilbert space form an orthomodular lattice [257], which is the foundation of the first Birkhoff-von Neumann quantum logic [242]. After that, projections were employed to reason about [258] or develop a predicate transformer semantics [259] of quantum programs. Recently, projections were also used in other quantum logics for verification purposes [260, 255, 261]. Orthogonal to

these prior works, this chapter proposes to use projection-based predicates in assertion, targeting runtime testing and debugging rather than logic or static analysis.

7.9 Conclusion

The demand for bug-free quantum programs calls for efficient and effective debugging scheme on quantum computers. This work enables assertion-based quantum program debugging by proposing Proq, a projection-based runtime assertion scheme. In Proq, predicates in the **assert** primitives are projection operators, which can significantly increase the expressive power and lower the assertion checking overhead compared with existing quantum assertion schemes. We study the theoretical foundations of quantum program testing with projection-based assertions to rigorously prove its effectiveness and efficiency. We also propose several transformations to make the projection-based assertions executable on measurement-restricted quantum computers. The superiority of Proq is demonstrated by its applications to inject and implement assertions for two well-known sophisticated quantum algorithms.

Chapter 8

SANQ: A Simulation Framework for NISQ Computing System

8.1 Introduction

Quantum computing has attracted great interest from both academia and industry in the last few decades due to its strong potential in accelerating various important applications, e.g., integer factorization [3], database search [7], molecule simulation [48]. The second quantum revolution, *transition from quantum theory to quantum engineering* [4], is leading us towards Noisy Intermediate-Scale Quantum (NISQ) era [53], when quantum computing devices have fewer than 1000 qubits and are not large enough to support Quantum Error Correction (QEC). To make good use of such NISQ devices which suffer from limited qubit lifetime and imperfect operations, more attention is given to NISQ system design and optimization in recent years, ranging across NISQ compiler [77, 85, 56], quantum control hardware architecture [262, 28, 263], NISQ device [51, 50, 49, 264], etc.

Ideally, all these innovations should be evaluated on realistic devices. However, NISQ systems require extreme execution environment and most of them still remain in physics

laboratories. Existing quantum computing cloud services, e.g. IBM Q Experience [37], Rigetti's QPU [52], only provide limited access which can not satisfy the ever-increasing demand for experiments for evaluating new NISQ system designs. These restrictions are blocking more researchers from getting into this area.

Simulation can be a potential solution to this problem as NISQ system innovations can be proposed and evaluated without accessing realistic hardware. Since a complete NISQ system consists of two major components, the quantum processor and its classical control system, a simulator for NISQ systems needs to meet the following requirements:

1. **Noisy Quantum Computing Simulation.** Quantum processors in NISQ era suffer from various noise effects. A simulator needs to be able to model the noises on realistic NISQ devices. The simulated output fidelity can help guide future NISQ system design.

2. **Classical Control System Simulation.** The design of a control system can significantly affect the overall NISQ system performance, especially the timing behavior. Such effects also influence the performance of the quantum processor. For example, longer execution time will bring more decoherence error.

Unfortunately, such a simulator that can satisfy these requirements is still missing. Traditional architectural simulators, e.g., GEM5 [265], GPGPU-Sim [266], are designed for classical digital computing without the ability to simulate quantum computing. Previous quantum computing simulation optimizations, no matter from algorithm level [267, 247, 268, 269, 270, 271, 272] or system level [273, 274, 275, 276, 277, 278], focus on a single execution and do not consider the computation redundancy among different noisy simulation traces. Moreover, a simulator that could capture the timing-sensitive components in the control system [279, 280] and be adaptive to different underlying quantum computing implementation technologies [28, 281] is still missing.

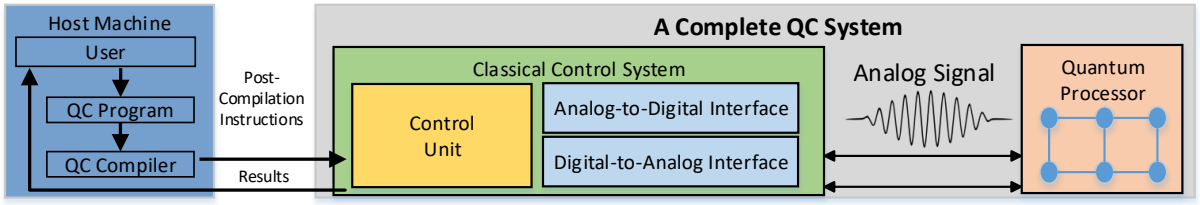


Figure 8.1: Schematic Overview of NISQ Computing System Architecture

In this chapter, we propose a simulation framework, namely SANQ, for NISQ system design and evaluation. SANQ consists of two major components. **First**, SANQ offers an optimized noisy quantum computing simulator with flexible error modeling accelerated by eliminating redundant computation. With the input circuit and the error model, SANQ will automatically generate a series of simulation traces. The traces will be reordered to maximize the overlapped computation between two consecutive traces and reduce the memory requirement for temporary intermediate state storage. Such inter-trace optimization can significantly improve the simulation performance, as a huge number of error injection traces have to be executed and averaged in the noisy simulation. **Second**, SANQ contains a reconfigurable control system architecture simulator to support various timing bottleneck analysis and related design space exploration. A set of abstracted timing-sensitive microarchitecture components (e.g., Digital-to-Analog Interface, Instruction Scheduler) is constructed to capture the timing behavior of an NISQ system. These components can then be connected to simulate the execution of an entire quantum program, providing not only the overall execution time but also the utilization of each component. In particular, a mini control system architecture configuration is provided by default.

In summary, the main contributions of this chapter are:

- We present the first comprehensive simulation infrastructure, which consists of an accelerated noisy quantum computing simulator and a control system architecture

simulator, for NISQ system modeling and evaluation.

- We propose a novel trace reorder technology to accelerate the noisy quantum computing simulation by eliminating redundant computation among error injection traces. To the best of our knowledge, this is the first trace-level optimization and can cooperate with existing full-state quantum computing simulators.

- We provide a reconfigurable control system architecture simulator with a set of abstracted timing-sensitive component models to support various timing bottleneck analysis and related design space exploration.

- Experimental results show that 1) Our accelerated noisy simulator can reduce about 79% computation amount on average (up to 95%) compared with existing full-state simulators. 2) Our control system simulator can capture the timing behavior of two real control systems.

- Several examples are provided as a preliminary study to show that SANQ could benefit compiler optimization, control system design, etc.

8.2 Background

In this section, we will present a brief review of relevant background knowledge to help understand the NISQ computing system.

8.2.1 NISQ System

Figure 8.1 shows a schematic NISQ computing system. On the left is a host machine, a classical computer which will interact with users and control the quantum computing system. Users provide quantum programs and the quantum computing compilers will

convert these programs to the basic instructions which can be executed by the control system. The control system will further convert the instructions to control signals and send them to the quantum processor to implement different operations.

Quantum Processor. The quantum processor is the core of the NISQ computing system, which can be implemented by different underlying technologies, e.g., superconducting quantum circuit [78], ion trap [282], and quantum dots [283]. The state of the qubits on the quantum processor is changed by external physical operations, e.g. microfrequency electronic signals [284], lasers [285]. For the lack of QEC, the qubits are also affected by various noise effects [25]. Unlike classical processors which work on digital signals, quantum processors are manipulated by analog signals.

Classical Control System. A classical control system lies between the host machine and the quantum processor [105]. It converts post-compilation instructions into control pulse signals to control the quantum processor. The measurement results in analog form are also received from the quantum processor and converted to a digital form. Such a classical control system provides a digital interface for the quantum processor and makes the NISQ system a co-processor of the host machine.

8.3 Simulator Overview

In this section, we will provide an overview of SANQ, a simulation framework that contains a noisy quantum computing simulator and a classical control system simulation infrastructure to cover the entire NISQ computing system. The workflow of SANQ is illustrated in Figure 8.2.

Input. The input required by SANQ has three components, a post-compilation quantum program, an error model, and a control system design. The instructions in the post-compilation quantum program must be executable on the simulated hardware,

which means all the quantum operations have been decomposed into hardware supported operations via compilation. The rest two components are about the simulated NISQ system. An error model should be provided to describe error operator, error position, and error probability on the simulated noisy quantum processor. Users can define customized error model via the provided interface. More accurate error model can come from the vendor or be characterized by physical experiments. The hardware design of the control system is about the model of each hardware module in the simulated control system. Users need to specify the output of each module under all possible input and how these modules are connected. By default, SANQ is pre-configured to be the baseline system model in the rest of this chapter. The input used in this chapter for the baseline error model and control system design is provided for user reference.

Simulation. With the required input information, the two simulation components in SANQ can provide comprehensive modeling of an entire NISQ system. The noisy quantum computing simulator uses the error information to construct an error model and generate error injection traces for the follow-up Monte Carlo (MC) simulation. The generated error injection traces will first be analyzed and reordered to eliminate redundant computation. Then SANQ will perform functional quantum computing simulation for all the error injection traces and average the results, to obtain an output distribution and evaluate the fidelity. On the other hand, the control system simulation infrastructure in SANQ will use the provided hardware design to generate a behavior model for the simulated control system. Traditional architectural simulation is then performed to model how the control system will execute each instruction of the input quantum program and control the quantum processor. Important information like the total execution time for a quantum program and the control hardware resource utilization rate can be simulated to evaluate the overall system performance.

Output. The output from SANQ will demonstrate key execution information of

the simulated NISQ system. The noisy quantum computing simulator will provide the final output distribution in the MC simulation. By comparing this result with error-free execution, SANQ can evaluate the fidelity for one quantum program execution on the simulated quantum processor. The control system simulation will then provide detailed timing information for one execution. More information like the occupation for each hardware component can also be collected to help locate the bottleneck in the simulated control system.

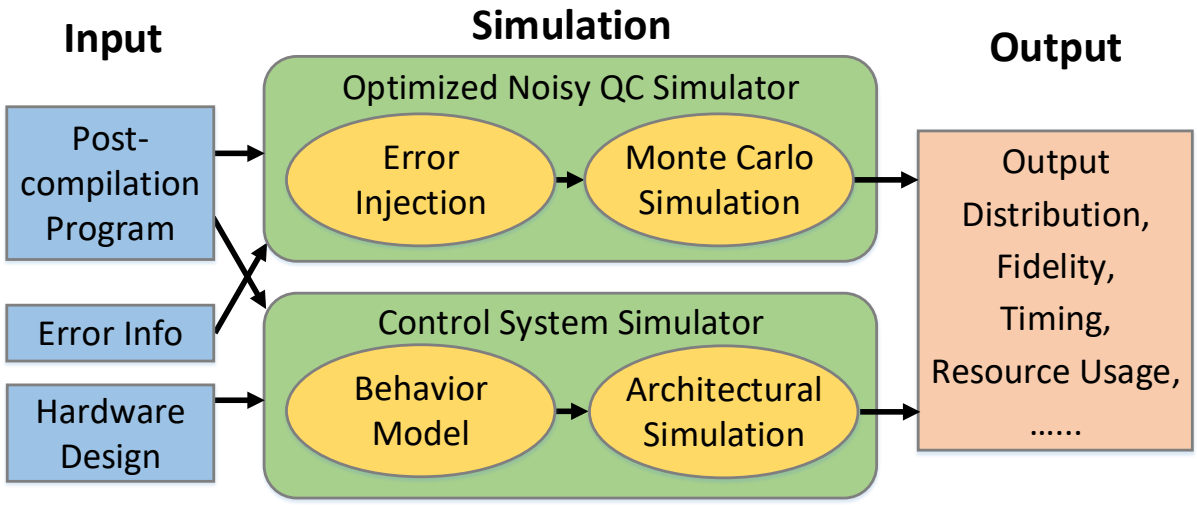


Figure 8.2: SANQ Workflow

This section provides an overview of SANQ. In the next two sections, we will introduce the two simulation components in detail with examples of how SANQ could simulate an existing NISQ system. In Section 8.4, we illustrate how to configure the error model based on IBM's public quantum processor information and how to accelerate the noisy quantum computing simulation by eliminating redundant computations. In Section 8.5, we will construct a mini control system in SANQ based on real control systems.

8.4 Noisy Simulation & Optimization

In general, simulating quantum computation on a classical machine is a hard problem. Noisy full-state quantum computation simulation is even more time-consuming since it requires simulating error-injected circuit many times to obtain an averaged result distribution. In this section, we will demonstrate how users can define an error model with *error operator*, *position*, and *probability*. SANQ will generate error injection traces based on the error model before running the actual simulation calculation. Then, we will introduce how we reorder these error injection traces to leverage the redundant computation among them without too much memory usage.

8.4.1 Error Modeling in Noisy Simulation

An error model indicates how error happens during the computation process. In SANQ, the error model has three parts, error operator, error position, and error probability.

Error Operator

Error operators are some special operators that will be randomly injected in the quantum circuit in order to model the noise effect in the quantum program execution on noisy quantum hardware. The three Pauli matrices, X, Y, and Z (given in Equation 8.1), are commonly used error operators to describe coherent errors. When a error happens, an error operator will be applied on the target qubit(s).

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{8.1}$$

Error Position

Error positions are the places where an error could possibly be injected in the simulated quantum circuit. For gate errors, error operators can be injected after a gate. For Some other errors like decaying from high-energy state $|1\rangle$ to low-energy state $|0\rangle$ or interacting with the environment can happen without an operation. Such an error could appear at any place across the quantum circuit.

Error Probability

After the error operators and positions are determined, we still need to know the probability for each error position with each error operator. Each time when we meet an error position during the simulation, we will randomly inject one error operator based on the error probability for each operator at this position.

Trace Generation

The error operator, position, and probability can construct an error model which can be used in the noisy quantum computing simulation. The error injection simulation traces will then be generated under the given error model. We use the symmetric depolarization error channel, a standard model employed in most noisy simulators [273], as an example to illustrate this procedure. Under this error model, the three error operators are X, Y, Z. There error probability for these three errors are equal, $p = P(X) = P(Y) = P(Z)$. The error probability and the simulated circuit are shown in Figure 8.3. Since the error is triggered by operations, we inject an error operator E after each gate. On the right of Figure 8.3 is the final error injected circuit. We will traverse this circuit to generate one simulation trace. Every error operator E is replace by X, Y, and Z with the same probability p, or by the identity operator I with the probability $1-3p$. We then record the

final circuit which will be used for one simulation trace. This traversal will be repeated for a large number of times to generate enough traces for the noisy simulation.

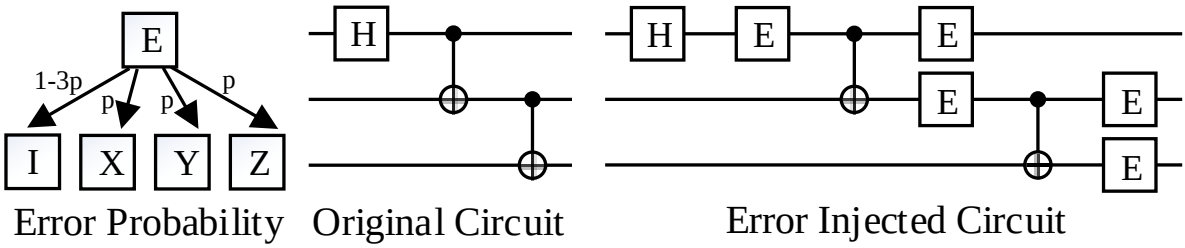


Figure 8.3: Depolarization Error Channel and Injection

Other Types of Errors

Except for the examples above, other types of coherent errors can also be defined in SANQ by reconfiguring the error operator, position, and probability. For the measurement operation errors which are very common but incoherent, since the error operator can only be applied to quantum states while the result after the measurement is a classical bit, we inject an error that flips the measurement result bit with the specified probability right after the measurement operation.

8.4.2 Noisy Quantum Computation Simulator Optimization

The redundancy among the error-injected simulations can be leveraged to reduce amount of computation. If two error-injection simulation traces share the same state in the middle, we can save this intermediate state in one simulation trace and then reuse it in the other simulation trace to eliminate the computation before this state. However, the size of a state grows exponentially as the number of qubits increases and it takes significant memory space to store a state vector. Thus, how to identify and store these states efficiently must be addressed to enable this inter-trace quantum computation

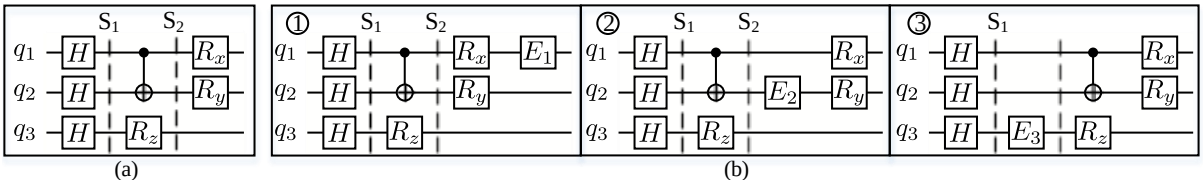


Figure 8.4: Example for Computation Redundancy and Execution Reordering

simulation optimization.

After we obtain the MC simulation traces with the input circuit and the error model, SANQ will perform a static analyze on the generated traces to identify the redundant computation and optimize the simulation trace order. We will first start from a example to illustrate the computation redundancy and the discuss how to efficiently run all the simulation traces.

Computation Redundancy

Figure 8.4 shows an example to demonstrate the computation redundancy. There are totally four error injection executions in this example, represented by four quantum circuits. The first one in (a) is the original error-free execution. S_1 and S_2 are two intermediate states during the error-free execution. The other three in (b) (labeled with ①, ②, and ③) are error injected executions. Each of them has one error operator occurred, represented by gates $E_{\{1,2,3\}}$. To run the noisy quantum computing simulation, all these four quantum circuits will be simulated and then averaged to obtain a distribution of the final output. We can find that all the four quantum circuits are exactly the same before reaching S_1 state. The state vector of S_1 is the same for all four execution since no errors are injected before S_1. As a result, the computation from the initial state to S_1 can be shared by all four executions. The state vector at S_1 only needs to be calculated and stored in one execution. The rest three executions can start from the stored S_1 state instead of starting from the beginning. Such redundancy exists at multiple locations

across the error injection MC executions. For example, the state vector at S_2 can be also be shared by the error-free execution and the first two error injected executions ①②.

The motivating example above has shown computation redundancy among MC executions. We can store some state vectors when we first reach such states and the results will be reused in the following executions. However, the maximal number of state vectors we can store is limited since one state vector has 2^n amplitudes (n is the number of qubits). Although several techniques have been proposed to store the state vector in a compressed form [271, 270], the memory requirement will still grow exponentially as the number of qubits increases. To allow circuits with more intermediate states to be simulated efficiently, we introduce an execution reorder technique to reduce the maximal number of concurrently maintained state vectors without loss of the benefit from the computation redundancy elimination.

Execution Reorder

Different execution order can significantly affect the number of states that need to be stored. For the example in Figure 8.4 (b), ①②③ is an inefficient MC execution order. When running ①, both the states S_1 and S_2 need to be stored so that ② can start from S_2 and ③ can start from S_1. An optimized execution order for this example can be ③②①. When executing ③, we only need to store state S_1. The execution of ② can directly start from the stored S_1 and then S_1 can be dropped since it is no longer used in the follow-up executions. During the execution of ②, S_2 will be stored and finally used when executing ①. Consequently, only one state vector needs to be stored during the entire simulation process. An optimized execution order reduced 50% of memory requirement (from two state vectors to one state vector) compared with a straight-forward order in this example.

In our noisy quantum computation simulator, we first generate the MC execution traces without actually running the simulation. The simulated quantum circuit is divided

into layers, in which any two quantum operations are not applied on the same qubit. Error operators will only be injected at the end of each layer (shown in Figure 8.3). One execution trace will record the location and operator of each injected error. These traces will be ordered by the location of the first injected error. The traces with the first error injected in the first layer (e.g., ③ in Figure 8.4) will appear at the beginning of the execution order, followed by those traces with the first error injected in the second layer (e.g., ② in Figure 8.4), and so on.

After the ordering procedure above, we begin our simulation by executing the first layer of the circuit with no error injected and store the state as S_1. This part of computation can be shared by all MC traces. Then we will execute all the traces with errors first injected in the first layer. If two or more error traces share the same first error (injected on the same qubit with the same error operator), these traces will be grouped. The simulation for these traces can be optimized recurrently if we consider S_1 as the initial state and let the remaining circuit after the first layer be the simulated circuit. After finishing the traces with first error in the first layer, we can execute one more layer without error and store the new state as S_2. Now S_1 can be dropped as no executions remaining will rely on it. Additional memory space is only required when recurrent reordering happens because these traces sharing first error operator need to store the state vector after the shared error to help eliminate the computation redundancy among them. The maximal number of state vectors we need to store is the recursion depth, which is small because the probability for two independent randomly generated traces to have m shared error operators decreases exponentially as m increases.

This execution reorder technique leverages the inter-trace computation redundancy and can cooperate with existing quantum computing simulation optimizations which focus on the execution of one simulation trace. The final simulation result will not be changed since the output of all traces are calculated and averaged.

8.5 Control System Simulator

In this section, we illustrate how to simulate a control system in SANQ with the default design, a mini control system, as an example. We start from discussing the assumptions on the programming model, compiler, and quantum processor, because they will affect the interface of the control system. Then we will introduce how to compose a hardware design with key timing-sensitive components abstracted from an investigation on several existing control systems from major vendors, e.g., IBM [281], Google [286], Rigetti [287], and TU Delft [28]. The behavior of each instruction can then be specified and finally we can simulate an entire quantum program in the composed control system.

8.5.1 Assumptions

Although programming and compilation should be done on the host machine and are not simulated in SANQ, some assumptions need to be made for them before we can continue to construct the architecture of a classical control system. For the quantum processor, our assumption is only about the interface with the control system and does not affect the error models in the noisy simulation.

Programming Model and Compiler

This mini control system accepts OpenQASM [226], the interface language of IBM's quantum computing cloud service designed for small depth quantum circuits, as the ISA. OpenQASM is selected due to its rich benchmark resource and compiler support. Quantum programs can be developed in high-level languages like Scaffold [233], Quipper [92], or Q# [231], and then compiled to flattened OpenQASM format instructions. However, some OpenQASM instructions are not executable so that we add some constraints for the program used in our mini control system. There are only 5 types of instructions from

OpenQASM remaining after compilation (the first 5 types in Figure 8.5). In addition, we add one 'Wait' instruction, which is critical in realistic control systems [28, 281], to enable more flexible timing control. Our control system will support this 6 types of instructions. For simplicity, the conditional instruction in the original OpenQASM standard is slightly modified and we only support one instruction in the branch based on one bit comparison result. The quantum operations in the post-compilation instructions are in the Quantum ISA (QISA) of the target quantum processor, which means the control signals for these operations are prepared and available. A conditional instruction in OpenQASM is also included and will be managed inside the control system. All the hardware constraints, e.g. the limited physical two-qubit gate availability, have been addressed during compilation optimization and the generated quantum program is completely hardware compatible. All the post-compilation instructions have been pre-uploaded to an instruction memory in the control system. There is no communication between the host machine and the control system during the quantum program execution.

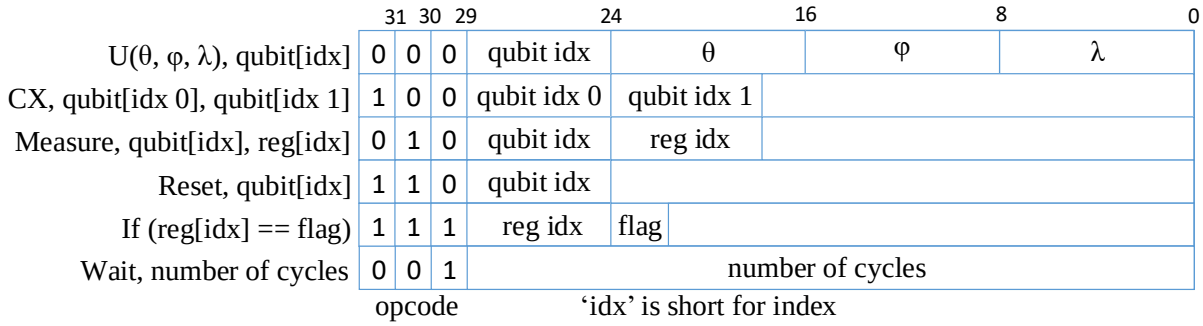


Figure 8.5: Instruction Encoding for Mini Control System

Quantum Processor

In this work, the quantum processor is assumed to be based on superconducting quantum circuit technology. The control signals for superconducting qubits are pre-calibrated micro-frequency electronic waveforms. Adapting IBM's configuration [288],

one single-qubit operation requires one control signal to be applied to that qubit. One two-qubit gate needs three control signals applied to the two qubits and the resonator between them. Other quantum computing technologies may have different interfaces. For example, ion trap devices can be manipulated by lasers. These different interfaces can be abstracted by analog signal channel between the quantum processor and the control system.

8.5.2 Hardware Design

With the assumptions above, users can specify the hardware design of the control system. For each hardware module, users need to determine what internal states the hardware module should maintain, and the output under all possible inputs. Moreover, users need to specify how the input and output ports of the hardware modules are connected in the hardware design.

As an example, a mini control system consisting of a control unit, a Digital-to-Analog (DA) interface, and an Analog-to-Digital (AD) interface, is shown in Figure 8.6. The hardware modules in the control unit are introduced as follows:

- *Instruction Memory.* This memory stores all the instructions. Since there is no existing binary encoding standard for OpenQASM [238]. we assume that each instruction consumes 32 bits (encoding shown in Figure 8.5). The input for this module is a memory address from Program Counter and the output is the instruction on that address which will be sent to a Decoder.

- *Program Counter.* The Program Counter (PC) records the address of the next instruction. It will automatically increase after one instruction is issued by the scheduler. It can also accept new address under conditional instructions.

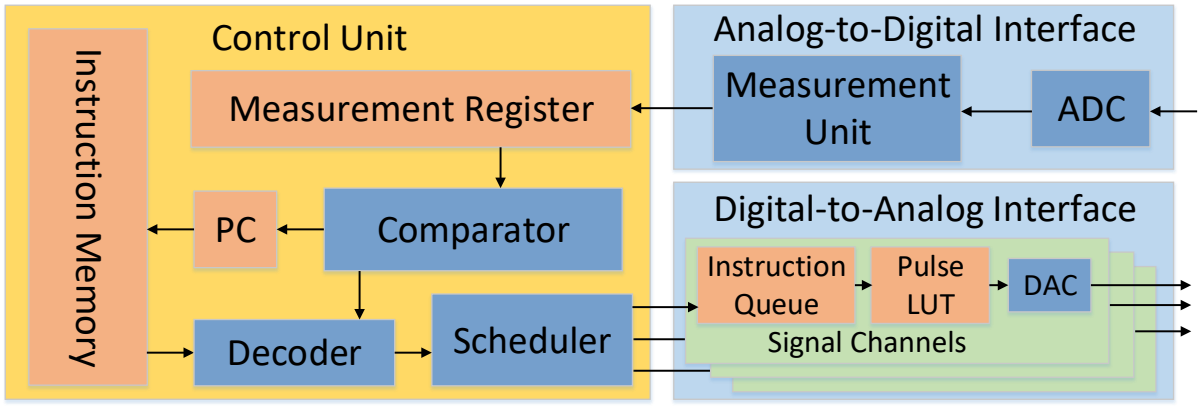


Figure 8.6: A Mini Control System

- *Measurement Register.* The measurement register stores the measurement results from the measurement unit. The comparator can read the measurement register.

- *Decoder & Comparator.* The decoder will decode those instructions in binary form fetched from the instruction memory. If it is a conditional instruction, the decoder will ask the comparator to read the measurement registers, do the comparison to determine the address of the next instruction. If an instruction needs to be applied on the quantum processor, the decoder will send the operation information to the scheduler.

- *Scheduler.* The scheduler will decide which signal channel(s) will be used to apply an operation and send the operation to the instructions queue(s) of the signal channel(s). The operation dispatch policy is to find the signal channel(s) that can finish all the jobs in the queue(s) at the earliest time. The instructions are dispatched in order.

Interface Design. The Mini Control System adopted the interface design from Quantum Control Box (QCB) [28], which is briefly introduced as follows. For the DA interface, we employ three DA signal channels, the minimum requirement to implement

two-qubit gates. Each channel has an instruction queue as a temporary buffer for the instructions. The waveform is implemented by a Pulse Look-Up-Table (LUT), which can directly fetch stored pulse data, and we assume that all the pulse waveform data are already in the LUT. A Digital-Analog-Converter (DAC) follows the Pulse LUT to generate analog signals. For the AD interface, one AD channel will receive an analog signal from the quantum processor and convert it to digital form by an Analog-Digital-Converter (ADC). The Measurement Unit will perform a weighted integration over the signal and then compare the results with a threshold value to determine whether the measurement result is 0 or 1. All the channels in the AD/DA interface can connect to different qubits via switches.

8.5.3 Behavior Model Generation

After the hardware design is specified, SANQ will generate a behavior model for the simulated control system. A behavior model is about how the hardware will execute the given instructions. In our mini control system example, only 5 types of basic instructions in OpenQASM standard [226] and the additional 'Wait' instruction in Figure 8.5 will appear after being compiled and flattened. The execution for these 6 types of instructions in the mini control system is listed here.

1. $U(\theta, \phi, \lambda)$. $U(\theta, \phi, \lambda)$ is a parameterized single-qubit gate. The decoder will send the instruction information to the scheduler and the scheduler will select one signal channel and put the instruction in the instruction queue. When this instruction is popped out, its control pulse will be fetched from the Pulse LUT, converted to an analog signal through DAC, and sent to the target qubit.

2. *CX*. CX is Control-NOT, the only supported two-qubit gate. Different from single-qubit gates, the scheduler needs to select three signal channels to complete this

operation.

3. *Measure.* Measure is the measurement operation. The scheduler needs to choose one DA channel to send a special pulse and one AD channel will receive a feedback pulse. The Measurement Unit will determine the output and write the result to the Measurement Register.

4. *Reset.* Reset is a single-qubit operation that resets the qubit to $|0\rangle$ state. In this mini control system, Reset is implemented by passive reset, which waits for $5 \times T_1$ coherence time to let the qubit decay to $|0\rangle$ state.

5. *If.* This is a conditional instruction. The decoder will ask the comparator to read the measurement register, do the comparison to determine the address of the next instruction. If the condition is not satisfied, the next instruction will be ignored.

6. *Wait.* The control system will wait for a specific number of cycles before executing the next instruction.

8.5.4 Architectural Simulation

After the behavior model is established, SANQ will simulate the control system by executing the provided post-compilation instructions. The post-compilation instructions are put into the *Instruction Memory* first and PC is set to be the address of the first instruction. Then, the configured NISQ control system will be simulated.

Besides simulating the execution time, SANQ can also actively collect and record the states of all the hardware modules, e.g., the number of instructions in each instruction queue, the number of instructions executed, etc. These statistical data can help locate the bottleneck in the system design. An example will be given in Section 8.7.

8.6 Evaluation

To demonstrate the effective and efficiency of our comprehensive NISQ simulator, we conduct a series experiments to evaluate the computation saving in the optimized noisy simulator and the timing simulation accuracy of our control system simulator.

8.6.1 Evaluating the Noisy Simulation Optimization

We conducted two groups of experiments to give a full test of our accelerated noisy simulator: 1) large-scale circuits with artificial error model. 2) small-scale circuits with realistic device error model.

Baseline. The baseline noise simulation strategy is from a full-state quantum computation simulator, Rigetti's QVM [289], which executes the error injection traces sequentially to generate an output distribution.

Metric. In order to perform a fair evaluation of our noisy simulator optimization, the metrics in this section are chosen to be independent of implementation and platform. For the computation time, we use the number of basic operations (matrix-vector multiplication) in the full-state quantum computation simulation to indicate the computation amount. For the memory consumption, we use the number of Maintained State Vectors (MSVs) during the noisy simulation since the memory space for the state vectors, which will grow exponentially as the number qubits increases, dominates the memory consumption.

Artificial Error Model

Benchmarks. Random circuit is widely-used in benchmarking quantum computation simulators [274, 290, 291, 247, 292]. We use Quantum Volume (qv) benchmark, one type of random circuit proposed by IBM [293]. Several qv programs are generated with various

numbers of qubits (from 10 to 40) and circuit depth (from 5 to 20) to test the computation saving and memory consumption as the input circuit scales. For example, 'n10,d10' means 10 qubits with circuit depth 10. The largest circuit used in this experiment with 40 qubits and depth of 20 is already close to the limit of existing full state quantum computation simulators [247].

Error Model. We use the symmetric depolarizing gate error model adopted by other simulators [76, 273]. The error rates of single-qubit gates ranges from 10^{-3} to 10^{-4}. 10^{-3} represents state-of-the-art superconducting quantum circuit technology and 10^4 reflects extrapolations of progress in hardware. The error rates of two-qubit gates and measurement operations are set to be $10\times$ of single-qubit gates. We generate 10^6 random error injection traces based on the error model for all quantum volume programs.

Results. Since the benchmark size is too large to be simulated on a standalone machine, we calculate the computation amount without actually performing the computation. Figure 8.7 shows the computation amount for all benchmarks with different error rates. On average, we can save about 79% computation. In the worst case, for a quantum volume circuit of the largest size and highest error rate, we can still save about 31% computation. The computation amount drops dramatically with lower error rates which can be expected in future devices. Figure 8.8 shows the number of MSVs, which grows slowly as the circuit depth increases. On average we need to store about 6 intermediate state vectors. When the number of qubits increases, the number of MSVs decreases because there are more potential error positions which reduce the probability for two traces to share the same injected error.

Realistic Error Model

Benchmarks. Table 8.1 shows the 12 quantum programs used in this experiment. They are collected from IBM OpenQASM benchmarks and prior work [238, 56]. These

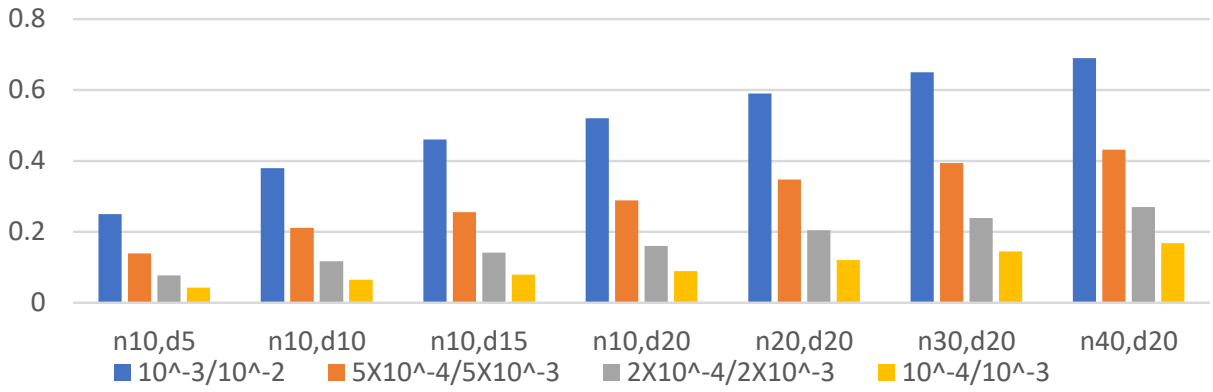


Figure 8.7: Normalized Computation for QV Experiments

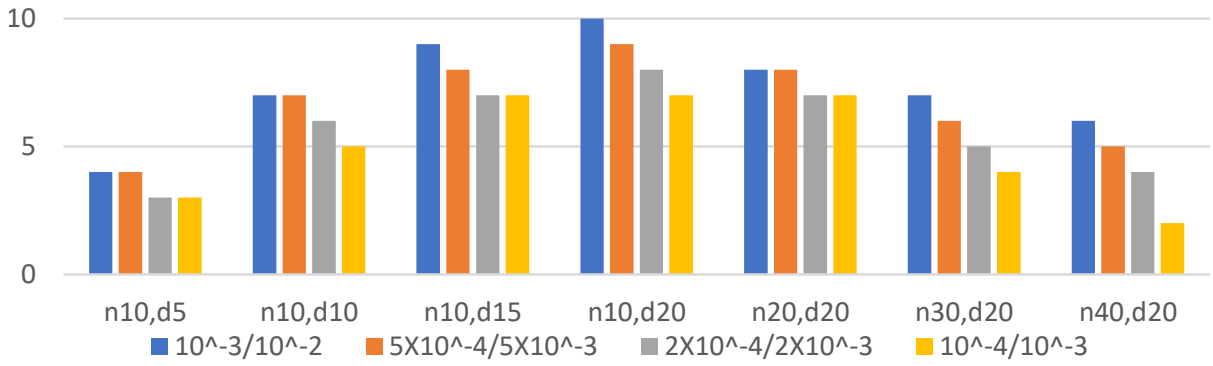


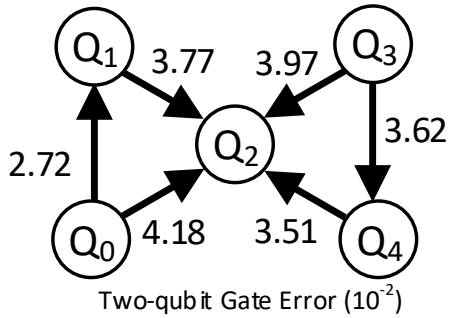
Figure 8.8: Memory Consumption for QV Experiments

benchmarks include Bernstein-Vazirani algorithm (bv) [294], Quantum Fourier Transform (qft) [25], Quantum Volume (qv) [293], Grover algorithm [7], Randomized Benchmarking (rb) [295], Modular Multiplication (7x1mod15) [76], and W-state [296]. The four columns on the right in Table 8.1 show the number of qubits and instructions in the post-compilation programs for each benchmark. The selected programs have 5 or fewer qubits to be simulated on the IBM 5-qubit chip model (illustrated by Figure 8.9) and do not contain *Reset* instructions. The measurement instructions only appear at the end of each program so that there are no conditional instructions. All the benchmarks only have $U(\theta, \phi, \lambda)$, CX, and $Measure$ instructions after compilation.

Error Model. We still use the symmetric depolarizing gate error model with the

Table 8.1: Benchmark Characteristics

Name	Qubit #	U #	CX #	Measure #
rb	2	9	2	2
grover	3	87	25	3
wstate	3	21	9	3
7x1mod15	4	17	9	4
bv4	4	8	3	3
bv5	5	10	4	4
qft4	4	42	15	4
qft5	5	83	26	5
qv_n5d2	5	44	12	5
qv_n5d3	5	74	21	5
qv_n5d4	5	100	30	5
qv_n5d5	5	130	36	5



	Single-qubit Gate Error (10^{-3})	Measurement Error (10^{-2})
Q_0	1.37	2.40
Q_1	1.37	2.60
Q_2	2.23	3.00
Q_3	1.72	2.20
Q_4	0.94	4.50

Figure 8.9: Error Rates on IBM Yorktown Chip [37]

error probability specified in Figure 8.9. All the benchmarks are compiled and mapped to this IBM's 5-qubit device [297] to the determine the actual physical qubits. We generate various numbers of traces (from 1024 to 8192) to test the computation saving under different simulation configurations.

Results Figure 8.10 shows the computation saving for all benchmarks and different numbers of traces. The proposed optimization can save about 75% $\sim$ 85% of computation on average with the number of traces increases from 1024 to 8192. In the worst case when the benchmark is large ('qv_n5d5'), the computation amount saving still achieves 57% with 8192 traces. We can also see that the more traces we execute, the more computation

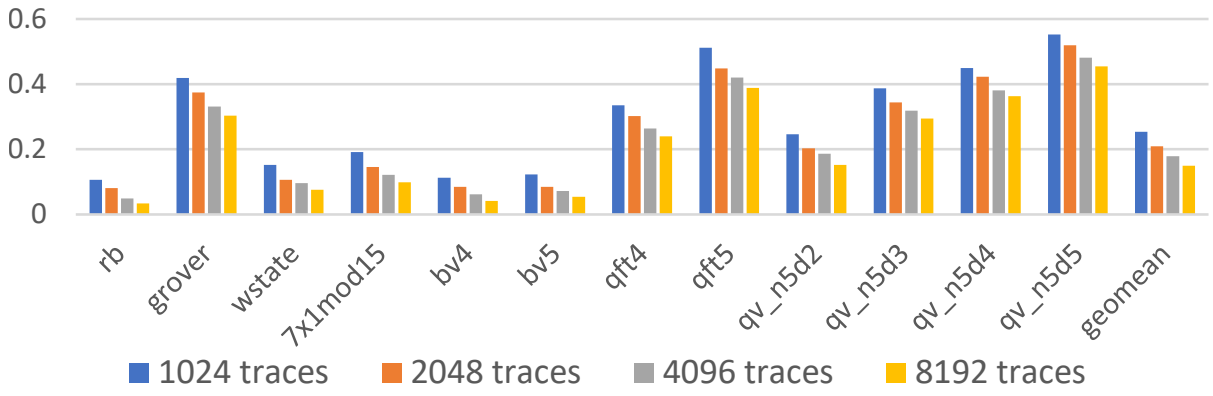


Figure 8.10: Normalized Computation for Realistic Error Model

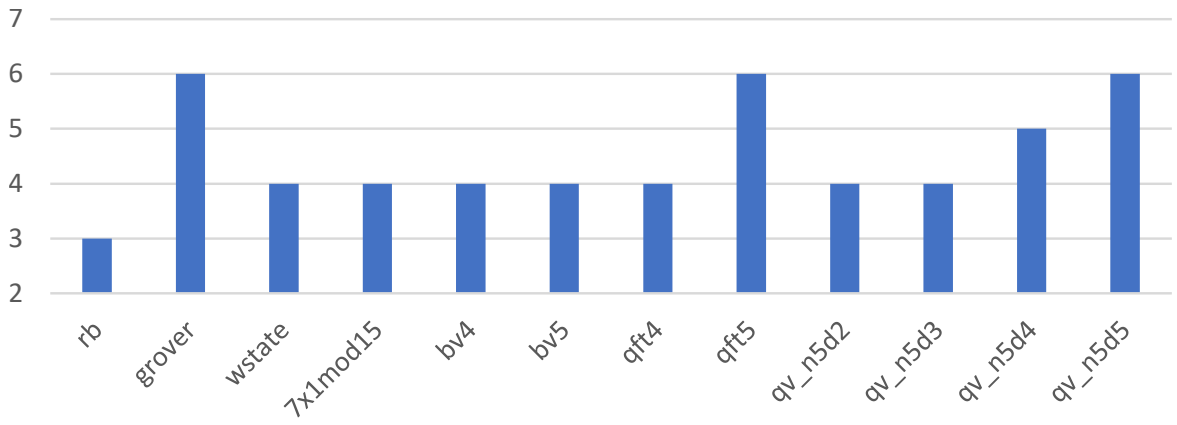


Figure 8.11: Memory Consumption for Realistic Errror Model

we will save because more overlapped computation can be identified. Figure 8.11 shows the number of MSVs in experiments with 1024 traces and this result does not significantly change when the number of traces increases from 1024 to 8192. The number of MSVs is 3 for the smallest benchmark 'rb' and only 6 in the largest benchmarks 'qft5' and 'qv_n5d5'. As discussed in Section 8.4, the number of MSVs will grow slowly since the probability for two traces to share the same m injected errors decays exponentially with m.

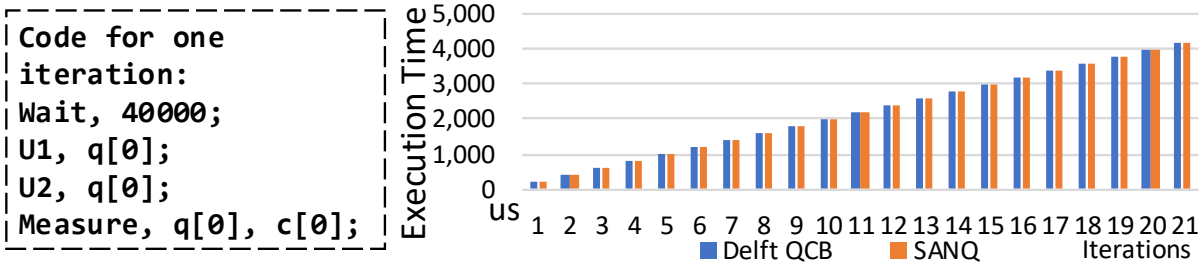


Figure 8.12: Timing Behavior Experiments with TU Delft's QCB

8.6.2 Evaluating the Control System Simulator

We evaluate the timing behavior of our control system simulator against two realistic control systems from IBM [281] and TU Delft [28].

TU Delft's Control System. TU Delft's Quantum Control Box (QCB) is a control system for a single-qubit quantum processor [28] The clock frequency is set to be $200MHz$. Other key parameters are shown in Table 8.2. The latency of single-qubit gates, two-qubit gates, and measurement operations are assumed to be $20ns$, $40ns$, and $300ns$, respectively.

To compare our simulation results our simulator against realistic execution of QCB, we run the AllXY program[1], the original testing experiment for QCB [28]. The AllXY test program has 21 iterations and in each iteration, two single-qubit gates are applied to one qubit followed by a measurement operation. Figure 8.12 shows the code for one iteration ($U1$ and $U2$ represent different single-qubit gates in different iterations) and the execution time on QCB and SANQ. Our simulator could imitate the timing behavior of QCB with very low error ($< 1\%$) and the small error becomes negligible as the number of iterations increases.

IBM's Control System. IBM's experimental control system model is different so that our simulator needs to be reconfigured. The latency for single-qubit and two-

[1]For details about AllXY program, please refer the QCB paper [28].

Table 8.2: Baseline Control System Model

	Single-qubit Gate	Two-qubit Gate
Latency	$20ns$	$40ns$
Channel	1	3
DA Channel #	3	
AD Channel #	1	
Measurement	Latency $300ns$, 1 AD Channel and 1 DA Channel	

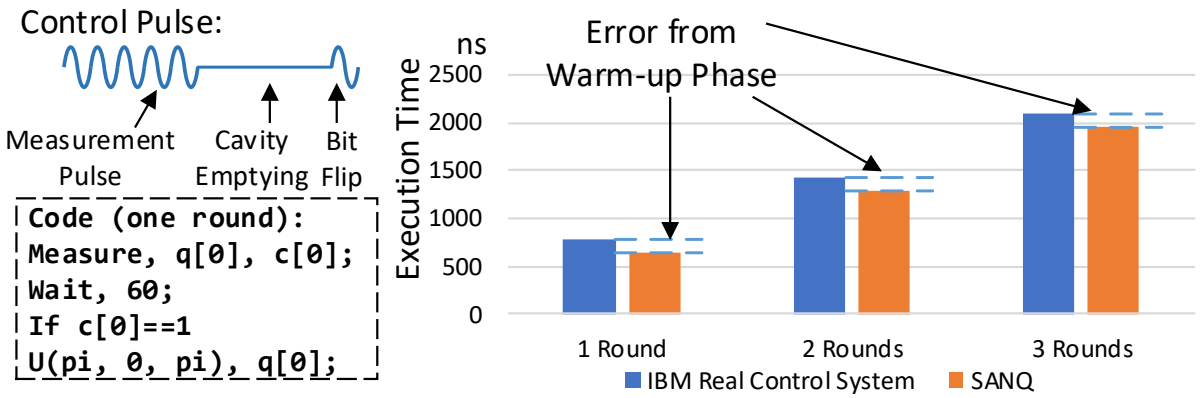


Figure 8.13: Timing Behavior Experiments with IBM

qubit gates are $50ns$ and $300ns$, respectively, with 2 DA channels and 2 AD channels. The test program is Active Reset as shown in Figure 8.13 on the left. We first send measurement pulse to a qubit and then wait for 60 cycles for cavity emptying (required by IBM's device). If the measurement result is $|1\rangle$, we apply a bit flip operation. This procedure is repeated for 3 times to guarantee a high reset fidelity. The execution time of IBM's real control system and the simulation results are in Figure 8.13 on the right. The simulated execution time is close to that of IBM's real system. There exists a constant error (about $130ns$) which comes from the warm-up phase of the control system because such procedure before issuing the first instruction is not yet simulated in SANQ. In summary, the average error ratio is 10% and such error can be mitigated if we take the communication between the host machine and the control system into consideration, which will be addressed in our future work.

8.7 Future Applications

In this section, we propose three future applications of SANQ that are not available on existing quantum computation simulators. First, SANQ can perform a comprehensive system performance evaluation by simulating both the quantum processor and the control system. Second, SANQ can perform design space exploration for the control system to guide future control hardware architecture design. Third, by monitoring the utilization of the hardware components, SANQ can help locate new optimization opportunities to improve NISQ system design. The rest of this section will provide three examples to illustrate the applications of SANQ in detail.

Baseline Configuration. The baseline quantum processor model in this section is from the IBM 5-qubit Chip [37] and generated in Section 8.4. The control system model is the TU Delft's QCB in Section 8.6 with key parameters shown in Table 8.2. The baseline compiler remains the same with Section 8.6 and the benchmarks used are in Table 8.1.

8.7.1 System Performance Evaluation

We demonstrate the ability to perform a comprehensive system performance evaluation by comparing two different QC compiler optimization approaches on the qubit mapping problem. One is the dynamic programming approach (DYN) in Enfield [56]. The other one is a heuristic approach for efficient qubit mapping (EFF) [77].

Experiment Design. We compile the 12 benchmarks with the two compilers mentioned above. Then we simulate the execution fidelity and time, from the noise quantum computation simulator and the architectural control system simulator, respectively. Since some benchmarks are large and the correct output will be hidden by the noise on IBM's 5-qubit device [107], the term 'execution fidelity' used in this section is the ratio of error-

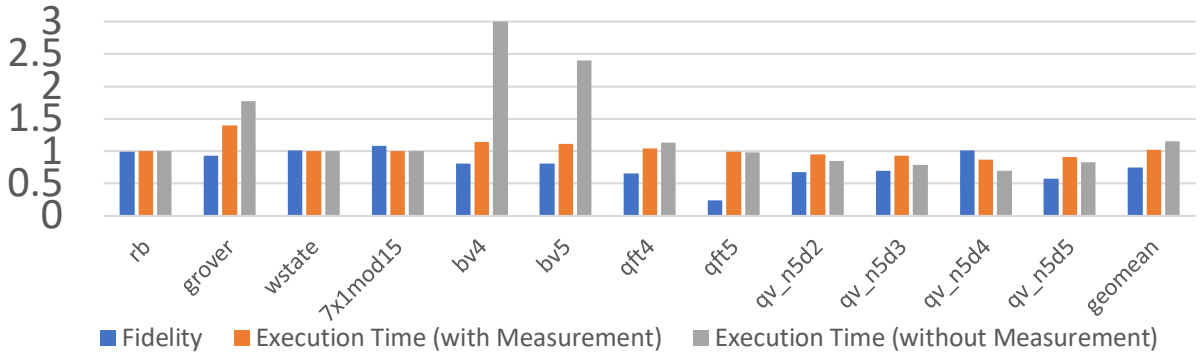


Figure 8.14: Normalized Simulation Result of EFF

free trace count over total trace count. Both the quantum processor model and control system model used in compilation and simulation are the baseline models.

Results. Figure 8.14 shows that the execution fidelity and time (with and without measurement operations included) of EFF normalized to the results of DYN. For two small benchmarks 'rb' and 'wstate', EFF and DYN generate the same code and the simulation results are the same for them. In general, DYN is well optimized for CX gates and the execution fidelity is about 35% better than that of EFF on average. However, EFF also considered parallelism optimization. For the 'qv' benchmarks, the execution time is shorter for EFF even when the execution fidelity is still worse than EFF. For the 'bv4' and 'bv5' benchmarks, they are small and the dominant factor in execution time is the CX gates so that EFF is much worse than DYN. The original evaluation in the EFF and DYN papers [56, 77] was based on the coarse-grained gate count and circuit depth metric in the generated program. SANQ generates consistent results to verify the optimality of DYN and the parallelism optimization in EFF. Moreover, SANQ could perform fine-grained fidelity and execution time evaluation, preparing for deeper compiler optimization.

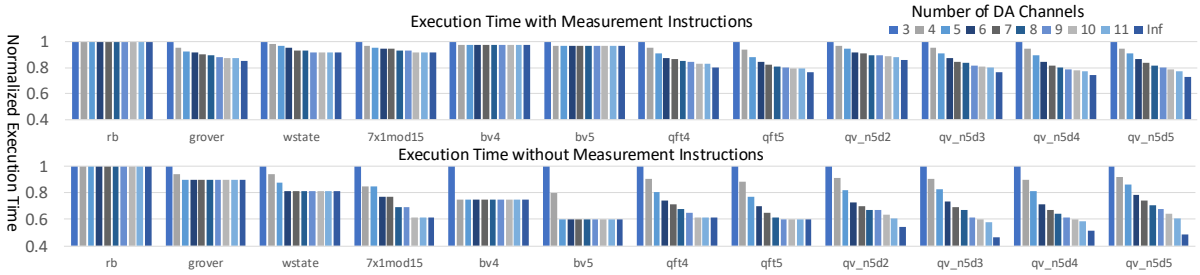


Figure 8.15: Execution Time Comparison with Various Numbers of DA Channels

8.7.2 Design Space Exploration

By simulating the classical control system, SANQ is able to perform design space exploration to help guide the control system design. This example focuses on the number of DA channels, which places an upper bound on the instruction parallelism. For a quantum processor, instructions applied on different qubits can be executed in parallel theoretically. However, the number of DA channels to send the control pulses is limited in a realistic control system. The baseline employs three DA channels (the same with QCB configuration [28]), which can support at most three simultaneous single-qubit operations or one two-qubit operation. In this study, we investigate how the number of DA channels can affect the overall performance of a NISQ computing system.

Experiment Design. We vary the number of DA channels from three to eleven and simulate the execution time. All other configurations remain the same. In the end, we assume that there are infinite DA channels to remove this constraint. This will show the ultimate limit if we continue to increase the number of DA channels.

Results. Figure 8.15 shows the execution time with various numbers of DA channels. The results shown in the upper half include the measurement instructions. All the benchmarks can benefit from more DA channels, except 'rb', which only has two qubits and is not constrained by the number of DA channels. Larger size benchmarks can save more execution time than small size benchmarks. When there are eleven DA channels,

most benchmarks have been close to the upper bound with infinite DA channels, which is about 15% on average since the execution is also limited by other effects, such as instruction dependencies.

Our simulation shows that the execution time of measurement instructions is the major limitation in this case study. For all the experiments, the number of AD channels is always one which means that all the measurement instructions must be executed sequentially. Moreover, the size of the selected benchmarks is small but the latency of measurement instruction is much longer than other operations in our quantum processor model ($300ns$ vs. $20 \sim 40ns$). Fortunately, all the measurement operations are at the end of each benchmark and we can calculate the execution time before the measurement. The execution comparison without the measurement instructions is provided in the lower half of Figure 8.15 and the average execution time-saving limit can achieve about 36%.

8.7.3 Finding New Optimization Opportunity

The third example will show that SANQ can suggest new optimization opportunities in NISQ system design by analyzing the execution status and locating the bottlenecks. For this example, we monitor the utilization rate of the DA channels in the system performance evaluation experiments (in Section 8.7.1). Figure 8.16 shows one bottleneck found in our experiments. On the left are the first five instructions in 'bv4' benchmark. In this case, SANQ finds that from $0ns$ to $20ns$, the number of instructions that is being executed is three and the DA channel utilization rate is 100%. But starting from $20ns$ to $60ns$, only one instruction is being executed and the utilization rate is just 33.3%. The reason for this situation is discovered after looking into the execution details (shown in the middle of Figure 8.16). From $0ns$ to $20ns$, the first three instructions are executed in parallel. The fourth instruction cannot be executed due to the DA channel constraint.

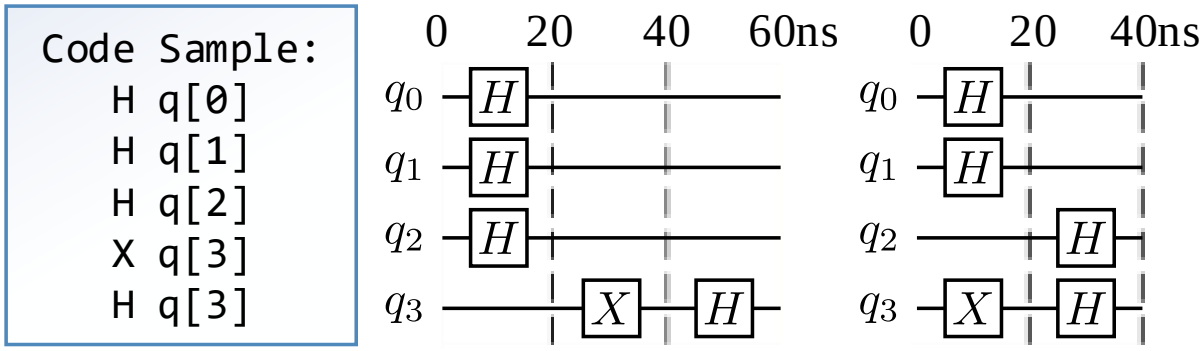


Figure 8.16: Example of Bottleneck

But from $20ns$ to $60ns$, only two instructions are executed because all the following instructions involve q_3 and cannot be executed before the fifth instruction. As a result, the utilization rate of DA channels is only 33% from $20ns$ to $60ns$.

It is hard to locate such bottleneck through traditional program profiling or quantum computing simulation without considering the actual control system. SANQ gives such opportunity to identify such hidden bottleneck, preparing for future system optimization. For example, the bottleneck mentioned above can potentially be solved in two ways. One approach could be compiler optimization. If the compiler knows that there are only three DA channels in the control system and hopes to reduce the execution time, a simple instruction reschedule can resolve this problem. For example, Figure 8.16 shows an example on the right. The compiler can exchange the third and the fourth instruction without changing the circuit function. The baseline control system can execute the X gate on q_3 first. Then the remaining two H gates can be executed in parallel. Totally, the first five instructions now only consume $40ns$, saving 33% of execution time compared with the original execution. Another approach is to employ a more intelligent scheduling policy for the control system. The baseline considers one instruction at a time and only dispatch instructions in order. This example suggests that a more powerful scheduler can consider more instructions ahead and issue instructions out-of-order to achieve a higher

utilization of hardware resources.

8.8 Limitations and Future Work

This chapter provides a simulation framework for a whole NISQ system. However, as an initial work in this area, SANQ comes with some limitations. In this section, we briefly discuss these limitations and our future plan.

More Precise Noise Modeling. The proposed noisy quantum computation simulator is equipped with widely used noise models. However, errors in realistic hardware can be even more complex. For example, all errors are generated independently in our Monte Carlo simulation while error correlation actually exists and is being studied by physicists [298, 299]. Deeper understanding of the mechanisms on the target quantum computing platform will lead to more precise noise models.

Advanced Quantum Control Architecture. The baseline control system is implemented with OpenQASM [226], a widely used intermediate representation for NISQ computing process. This interface language is designed for small depth quantum circuit experiments on IBM's quantum computing cloud service and lacks several important features for a control system ISA, e.g., efficient encoding, flexibility for quantum optimal control [300, 301]. For further research, SANQ will adopt more advanced quantum control architectures, such as eQASM [263].

Cooperating with Host Machine. SANQ assumes that all the post-compilation instructions have been transferred to the control unit and does not consider the host machine. The assumption brings error in the simulation as discussed in Section 8.6. In the future, SANQ can be integrated as a sub-module into an existing computer system simulator, e.g., GEM5 [265], to include communication between the host machine and the quantum computing subsystem.

8.9 Related Work

Quantum Computing Simulator and Optimization. Previous optimizations for quantum computation simulators can be summarized into two categories. Some simulators increase the simulation capability from algorithm-level [267, 247, 268, 269, 270, 271, 272]. , e.g., tensor networks [247, 268], stabilizers [269, 270], decision diagrams [271, 272]. These works exploited sparsity or redundancy inside a single quantum computing simulation process while the proposed optimization leverages the redundancy among multiple MC simulation executions. The other type of optimizations is from computer system level, including vector instructions[273, 274], specialized linear algebra library [275], multi-thread [273, 274, 276], distributed system [274, 275, 276], GPU [277, 278]. Our acceleration is from algorithm-level and is compatible with previous approaches focusing on single trace simulation optimization.

Control Systems. The electronic interface for quantum processors has been studied for small size cases [302, 303, 304, 305, 105]. Fu *et al.* proposed QuMA, a microarchitecture with accurate timing control, fast feedback control, etc. for a superconducting quantum processor [28]. A cycle accurate microarchitectural-level simulator called QuMASim is developed for this specific architecture [279, 280]. Leon *et al.* also proposed a cycle accurate simulator for each hardware module in the control system without a complete microarchitecture [306]. Dijk *et al.* proposed SPINE, a toolset with a circuit simulator, for co-simulation of the electrical circuit and a spin-qubit-based quantum processor [307]. The control system simulator in this work provides abstracted timing-sensitive components and can be easily reconfigured to various systems.

8.10 Conclusion

This chapter introduces SANQ, a simulation framework for architecting NISQ computing systems. SANQ consists of two components, an optimized noisy quantum computation simulator and an architectural simulation infrastructure for the classical control system. The noisy quantum computation simulator is equipped with flexible error modeling and optimized by computation redundancy elimination. The architectural simulation infrastructure can construct behaviour models and evaluate control systems design decisions. The usage of SANQ is illustrated by adopting realistic error model and published control system design. Three examples are given to show that SANQ could benefit NISQ system design through comprehensive system evaluation and execution status analysis. In conclusion, this chapter proposes the first NISQ system simulator, allowing more researchers to participate in quantum computing research and perform the early-stage evaluations for future innovations.

Chapter 9

Conclusion and Discussion

Quantum computing is still in its very early stage and many challenges remain ahead. This dissertation explores how to improve quantum computer systems from several different aspects, ranging across programming language to compiler and hardware architecture with an emphasis on the software and compiler side. Several research outputs in this dissertation have been adopted by industry quantum software frameworks. We believe that the high-level principles in this dissertation, the cross-layer co-design and high-level optimization, can be long-living and hasten the onset of practical quantum advantage.

In the rest of this chapter, we will elaborate on why we should continue pursuing quantum computing and discuss some future research directions.

9.1 Pursuit of Quantum Computing

The works covered in this dissertation started from 2017 when quantum computing was less popular compared with that in 2022 when the dissertation was finished. Many leading industry companies and startups joined the pursuit of quantum computing. Along with the increasing popularity, criticism also arises. It has been around 40 years since the

beginning of the pursuit of quantum computing but we have not yet succeeded. Google claimed that they achieved 'quantum supremacy' [12] while the claim was retracted as the classical simulation algorithm is also improved [308]. There are still many challenges to be solved before we can achieve practical quantum computing. It is highly possible that we are still in the hype cycle [309] similar to many other technology innovation areas.

We argue that we should still pursue quantum computing, no matter whether quantum computing will finally be successful or not. Suppose that, in the worst case, we finally cannot make a large-scale fault-tolerant quantum computer and we cannot run Shor's algorithm or other promising quantum algorithms at a large scale. In this case, our pursuit of a quantum computer is still very valuable. Such pursuit is similar to the "race to the moon" which is much more than just sending people to the moon. During this process, many related studies (e.g., rocket technology, remote sensing, how people can survive in outer space) were triggered and many companies are developed. Building a quantum computer is not just about quantum computing. Actually, it has already triggered a lot of technological innovations. For example, to build a quantum computer, we need to develop cryogenic devices and electronics, new materials, fabrication technologies, etc. On the algorithm side, we have developed new theories to understand quantum and new algorithms, It also triggers interdisciplinary collaborations, for example, finding the analog control pulse of a quantum system is an important problem and people have already tried to use machine learning/reinforcement learning techniques to tackle this problem. Overall, this pursuit of quantum computing will definitely benefit the overall society.

It is also frequently questioned that the near-term works in quantum computing may be obsolete. We agree that the final large-scale quantum computer may be very different from those small prototypes we have today. But the technologies at different stacks may

impact quantum computer development in different ways. For higher-level technologies like theories and algorithms, we believe they will have a long lifetime because mathematics theory usually does not change too much. For low-level technologies like the device and hardware, they are in rapid development and evolving. However, to understand the true potential of these different technology routes, we need to carefully design and optimize the entire system. This is a road that we must walk through before we can reach the large-scale fault-tolerant quantum computer.

9.2 Future Research Directions

Looking forward, we will deepen and strengthen the quantum computer system research in several dimensions. Specifically, I plan to develop photonics-based quantum systems that execute non-circuit quantum computation models. I will also work on compiler-based autotuning for quantum algorithm design and projection-based quantum program analysis/debugging for quantum software engineering. My long-term goal is to hasten the onset of practical quantum advantage with our quantum computing system design and optimizations.

9.2.1 Photonics-Based Quantum Computing with Non-Circuit Computation Model

The recent development of integrated silicon photonics allows creating and manipulating many photons and makes the photonics a new promising quantum computing technology candidate. Compared with other mainstream candidates like superconducting quantum circuit or ion trap, photonics-based quantum computing is appealing because photons have low noise (decoherence free, crosstalk free), light-speed transmission,

room temperature operation, and fast measurement. Despite the advancement on the photonics devices, the software infrastructure and hardware architecture for photonics-based quantum computing system are much less developed. Most existing quantum computing systems are constructed for running the quantum circuit computation model with gates and qubits. However, the computing paradigms of photonics-based quantum computing, including measurement-based quantum computing, fusion-based quantum computing, continuous-variable quantum computing, are of very different characteristics even if they are theoretically equivalent. Therefore, existing quantum computing systems cannot be directly migrated to photonics-based quantum computing. I plan to develop a set of software tools to support and optimize the inter-paradigm compilation, including 1) efficient data conversion between conventional two-dimensional qubits and infinite-dimensional qumodes on photonics devices, and 2) operation translation with an emphasize on classical-quantum interaction due to the frequent measurement on photonics-based quantum computing. I will also explore the hardware architecture design for photonics-based quantum computing system, including the on-chip photon generator, router, detector, and the classical post-processing units.

9.2.2 Advanced Quantum Compilation with Algorithmic Auto-tuning

Traditional algorithms that demonstrate quantum speedup cannot be implemented on state-of-the-art devices due to the huge resource requirement. For example, factoring a number that is too large to tackle using known classical algorithms, Shor's algorithm requires thousands of logical qubits and tens of millions of physical qubits when full quantum fault tolerance is applied. New quantum algorithms like VQE for quantum chemistry simulation and QAOA for combinatorial optimization, which seem to be less

resource-demanding and more noise resilient, have been devised recently. However, executing these algorithms on today's quantum hardware platforms is still challenging. As such, there is active research trying to develop resource-efficient algorithmic alternatives tailored to various quantum devices. Yet, manual algorithmic optimization is slow and challenging given the large optimization space and complex design trade-offs. This motivates us to investigate compiler toolchains that could enable automatic algorithmic optimizations for the quantum computing domain, *e.g.,* an advanced compilation infrastructure with the capability of algorithmic autotuning. Specifically, I will abstract the algorithmic designs in the quantum domain into a compiler optimization problem. The key here is to crystallize the development of resource-efficient quantum algorithms into a set of optimization tuning configurations, figure out their dependencies and applicability, and explore the optimization space efficiently. I will develop the first language and compiler support for quantum algorithmic optimization. The algorithmic design space will be explored systematically and efficiently to generate quantum algorithms that match or outperform those manually crafted by domain experts.

9.2.3 Projection-Based Quantum Program Debugging and Analysis

Program logics, model-checking, equivalence checking, termination analysis, reachability analysis, and invariant generation, have been extended to evaluate the correctness of the error-prone quantum programs. Unfortunately, these methods often require simulation on the entire state and thus are restricted to programs with only 25-40 qubits even on the best supercomputers today. As an alternative method, runtime assertion is recently employed in quantum computing for efficient testing and debugging. Existing quantum program assertions employ a classical language to describe the assertion predi-

cates and only three simple specific types of quantum states asserted. A lot of complex intermediate program states cannot be tested by these assertions due to their limited expressive power. To address these challenges, I plan to establish a general framework for quantum program analysis, testing, and debugging. The key is to leverage the power of *projection operators* in quantum logic and projective measurements. Our preliminary work [22] on projection-based runtime assertions has demonstrated great efficiency and I will extend it to a more realistic setting and build end-to-end toolsets for practical usage. I also plan to augment quantum program analysis through automatic projective invariant generation and quantum logic reasoning with projection-based predicates. I will rigorously formulate the theoretical foundations for the new analysis and debugging schemes, and propose efficient and effective practical implementations of the runtime assertions, invariant generation, and logical reasoning.

9.2.4 Electronic Design Automation for Quantum Device/System.

The demand on larger-scale quantum computing devices and systems naturally invokes electronic design automation (EDA), which can help with device modeling, simulation, and design space exploration while a top-down EDA toolset for quantum devices is still missing. This motivates me to develop new systematic quantum computing device modeling methods via simulation from the physics level, circuit level, and architecture level. At the *physics* level, the traditional first principle simulation methods, e.g., electromagnetic field simulation, are usually very time-consuming. I plan to accelerate the physics level simulation from both the algorithm side (e.g., customizing finite element analysis for quantum computing devices, applying machine learning techniques) and system side (e.g., hardware acceleration, compiler/programming system support).

The physics level simulation results will serve as lumped parameter models in the circuit level simulation. At the *circuit* level, I will leverage EDA methods from traditional CMOS technology. I plan to optimize the device fabrication, e.g., circuit place and routing for superconducting quantum processors when performing circuit quantum electrodynamics simulation on the lumped parameter model. The simulated circuit models will serve as the hardware building blocks in the architecture level simulation. At the *architecture* level, I plan to cover both the quantum processor itself and the peripheral digital-analog hybrid classical control system for a comprehensive evaluation. I have built a preliminary architecture design flow [19] for superconducting quantum processors based on IBM's device model and I will extend it to support more quantum computing technologies.

Appendix A

Appendix for Chapter 6

In this chapter we provide the artifact description of the Paulihedral compiler introduced in Chapter 6.

A.1 Artifact Abstract

The artifact contains the source code of the Paulihedral compiler and other necessary code scripts to reproduce the key results (Table 6.2, 6.3, and 6.4) and compare with the baselines in our evaluation. The hardware requirement is a regular X86 server/laptop but the memory size may limit the size of the benchmark that can be compiled. The IBM Melbourne device used in our evaluation has just retired and the related results cannot be reproduced. But we still keep the original script of that experiment for your reference. The software dependencies only contain common software packages. We also provide our benchmark generation script and have pre-generated all benchmarks used in our evaluation. Note that for Table 6.3 the results are averaged over 20 randomly generated graphs per benchmark. While in our artifact, we show the result of one random seed and a slight deviation is expected.

A.2 Artifact Checklist

- **Language:** Paulihedral has a new intermediate representation (IR), Pauli IR, which is implemented by a 2-dimensional Python list in this artifact. Examples can be found in 'Paulihedral.ipynb'.
- **Algorithm:** Paulihedral has four core algorithms.
  - Gate-count-oriented scheduling (Section 4.1) is the function 'gate_count_oriented_scheduling' in 'parallel_bl.py'.
  - Depth-oriented scheduling (Section 4.2) is the function 'depth_oriented_scheduling' in 'parallel_bl.py'.
  - Block-wise optimization on fault-tolerant backend (Section 5.1) is in function 'block_opt_FT' in 'synthesis_FT.py'.
  - Block-wise optimization on superconducting backend (Section 5.2) is in function 'block_opt_SC' in 'synthesis_SC.py'.
- **Benchmarks:** The benchmarks are the Pauli IR programs of the simulation kernels listed in Table 6.1.
- **Runtime environment:** Python, Jupyter Notebook.
- **Disk space required:** 10 GB is sufficient for the artifact and all software dependencies.
- **Hardware:** Intel CPU, Memory size depending on the benchmark size (the largest benchmarks can be processed with 1T RAM).
- **Experiments:** Compiling the Pauli IR programs using Paulihedral and follow-up generic quantum compilers.

- **Time to prepare workflow:** 10 minutes
- **Time to complete experiments:** The approximate execution time for each benchmark under different configurations can be found in Table 6.2. It will take hundreds of CPU hours to fully reproduce all results in Table 6.2, 6.3, and 6.4.
- **Output:** The output of the compilation is the quantum circuit containing CNOT gates and single-qubit gates only.
- **Metrics:** We consider the following metrics in the output
  - Number of single-qubit gates
  - Number of CNOT gates
  - Number of all gates
  - Circuit depth
  - Execution time

  All these metrics can be directly counted from the output quantum circuit.
- **Publicly available:** Yes
- **Code license:** Apache License 2.0
- **Workflow framework used:** Jupyter notebook, Qiskit, t|ket⟩
- **Archived repo:** `https://zenodo.org/record/5780204`
- **DOI:** 10.5281/zenodo.5748398

A.3 Description

A.3.1 How to Access

The artifact is available at the following Zenodo link `https://zenodo.org/record/5780204` with DOI 10.5281/zenodo.5780204. You can download the zip file and then decompress it.

A.3.2 Hardware Dependencies

A regular server with Intel CPUs can run our artifact while the amount of RAM may limit the size of benchmarks that can be executed. In our experiments, we use 1T RAM to execute all benchmarks. If you do not have enough RAM, it is possible that the large benchmarks like 'NaCl' and 'Rand-80' are not executable due to out of memory. Note that in Section 6.4, we have real system experiments on IBM's Melbourne device. This device has permanently retired and is not longer accessible. So we are not able to reproduce the results in Figure 6.11.

A.3.3 Software Dependencies

The artifact in implemented in Python 3.8.12. We require Qiskit and t|ket⟩. In our experiments, we use Qiskit 0.23.5 and t|ket⟩ version 0.11.0. while other versions may or may not work. These two frameworks requires numpy 1.20.0. We also need jupyter notebook, which can installed from Anaconda, since we prepare the file 'Paulihedral.ipynb' that contains scripts to automatically and interactively reproduce the results in Table 6.2, 6.3, and 6.4 for easy validation. See 'README.md' for installing the software dependencies. Note that the PySCF version must be 1.7.6. The QAOA compiler used in our evaluation (Section 6.2, Table 6.3) is downloaded from https://github.com/mahabubul-

alam/QAOA-Compiler and has already been integrated in this artifact (in the folder 'QAOA-Compiler'). The QAOA compiler requires networkx 2.5.0 and commentjson 0.9.0. The list of dependencies can be found in 'requirements.txt'.

A.3.4 Benchmarks

The benchmarks can be generated using the file 'gene_benchmark.py'. We have pre-generated all benchmarks used in our evaluation and they can be found in benchmark/-data. You can also generate Pauli IR programs from you own Hamiltonians/applications following the format in the example in the first code block in 'Paulihedral.ipynb'.

A.4 Installation

To use our artifact, you can first download the repo to your local machine. Then you can install the software dependencies by running the command:

```
pip install -r requirements.txt
```

A.5 Evaluation and Expected Results

After you download the artifact and install all software dependencies, you can open the jupyter notebook file 'Paulihedral.ipynb'. The first code block will demonstrate an example of a Pauli IR program. Note that we just set all the rotation angles in the center Rz gates to '1.0'. The Rz gates will not be affected in the entire compilation flow. The second, third, and fourth code blocks will automatically reproduce the results in Table 6.2, 6.3, and 6.4, respectively. The results are printed out directly. Note that the results in Table 6.3 are averaged over 20 randomly generated graphs per benchmark. While in our artifact, we show the result of one random seed. Therefore, a slight deviation

is expected. Note that the execution time cannot be perfectly reproduced because your local machine configurations can be different from the server we used in our evaluation while the trend should remain the same.

Since reproducing all the results are very time consuming (hundreds of CPU hours on a server), we add an option in 'config.py' so that small-size experiment results can be reproduced quickly. In 'config.py', if you set 'test_scale' to 'full', then the code will run all benchmarks; if you set 'test_scale' to 'small', then the code only run small benchmarks, which will take about a few CPU hours on a MacBook. By default, 'test_scale' is set to 'small'.

We also attach our code (in file 'real_system.py') for experiment on the IBM devices. This script can print out the compilation results when compiling the QAOA programs onto the IBM Melbourne chip. However, since the IBM Melbourne chip used in this dissertation is no longer available, the real system execution results in Figure 6.11 cannot be reproduced. You can change the device to other available IBM devices in the script.

Appendix B

Appendix for Chapter 7

B.1 Proof of the theorems, propositions, and lemmas

B.1.1 Proof of Theorem 7.3.1

Theorem: *Suppose we repeatedly execute S' (with l assertions) with input ρ and collect all the error messages.*

1. *(Posterior) If an error message occurs in* $\mathbf{assert}(\overline{q}_m; P_m)$*, we conclude that subprogram S_m is not correct, i.e., with the input satisfying precondition P_{m-1}, after executing S_m, the output can violate postcondition P_m.*

2. *(Posterior) If no error message is reported after executing S' for k times ($k \gg l^2$), we claim that program S is close to the bug-free standard program; more precisely, with confidence level* 95%*,*

   (a) the confidence interval of $\min_{S_{\text{std}}} D\left([\![S]\!](\rho), [\![S_{\text{std}}]\!](\rho)\right)$ *is* $\left[0, \frac{0.9l+\sqrt{l}}{\sqrt{k}}\right]$,

   (b) the confidence interval of $\max_{S_{\text{std}}} F\left([\![S]\!](\rho), [\![S_{\text{std}}]\!](\rho)\right)$ *is* $\left[\cos\frac{0.9l+\sqrt{l}}{\sqrt{k}}, 1\right]$,

where the minimum (maximum) is taken over all bug-free standard program S_{std} *that satisfies all assertions with input* ρ.

Moreover, within one testing execution, if the program s_m *is not correct but* $\mathbf{assert}(\overline{q}_m; P_m)$ *is passed, then follow-up assertion* $\mathbf{assert}(\overline{q}_{m+1}; P_{m+1})$ *is still effective in checking the program* S_{m+1}.

Proof: The proof has three parts.

• Error message occurred in $\mathbf{assert}(\overline{q}_m; P_m)$.

Obviously, no error message occurred in $\mathbf{assert}(\overline{q}_{m-1}; P_{m-1})$, which ensures that the current state ρ after the assertion $\mathbf{assert}(\overline{q}_{m-1}; P_{m-1})$ indeed satisfies $\rho \models P_{m-1}$.

After executing the subprogram S_m, the state becomes $[\![S_m]\!](\rho)$. The error message occurred in $\mathbf{assert}(\overline{q}_m; P_m)$ indicates that $[\![S_m]\!](\rho) \not\models P_m$, which implies subprogram S_m is not correct, i.e., with the input satisfying precondition P_{m-1}, after executing S_m, the output can violate postcondition P_m.

• No error message is reported.

We assume that for the original program S, the state before and after S_m is ρ_{m-1} and ρ_m for $1 \leq m \leq l$; and for the debugging scheme S', the state after $\mathbf{assert}(\overline{q}_m; P_m)$ is ρ'_m for $1 \leq m \leq l$ and set $\rho'_0 = \rho$.

We first show the trace distance D and angle A (distance defined by fidelity[1]) of $[\![S_m]\!](\rho'_{m-1})$ and ρ'_m. Realize that, the k executions of assertion $\mathbf{assert}(\overline{q}_m; P_m)$ are k independent Bernoulli trials with success (report error message) probability $\epsilon_m = 1 - \text{tr}\big(P_m[\![S_m]\!](\rho'_{m-1})\big)$. With the result that there is no success in k trials, we here use the commonly used methods of binomial proportion confidence interval, the Clopper-Pearson interval[2] [310] to estimate the actual value of probability ϵ_m. The confidence interval

[1]Formally, $A(\rho, \sigma) \triangleq \arccos(F(\rho, \sigma))$.

[2]It is also called the 'exact' confidence interval, as it is based on the cumulative probabilities of the binomial distribution.

(CI) of ϵ_m is $\left(0, 1-\left(\frac{\alpha}{2}\right)^{\frac{1}{k}}\right)$ with confidence level $1-\alpha$; in other words, based on the trial results, we may draw the distribution of possible actual value, which is expressed as:

$$\Pr(a \leq \epsilon_m \leq b) = \int_a^b f_X(x)\mathrm{d}x,$$
$$f_X(x) = \mathrm{Beta}(1,k) = k(1-x)^{k-1}.$$

According to Lemma 7.3.1, we know that:

$$D([\![S_m]\!](\rho'_{m-1}), \rho'_m) \leq \epsilon_m + \sqrt{\epsilon_m(1-\epsilon_m)} =: Y_m$$
$$A([\![S_m]\!](\rho'_{m-1}), \rho'_m) \leq \arccos(\sqrt{1-\epsilon_m}) =: Z_m$$

Some properties of Y_m and Z_m are listed below[3]:

	center estimate	CI
Y_m	$\frac{1}{k+1} + \sqrt{\frac{\pi}{4k+3}}$	$\left[0, \frac{\beta}{k} + \sqrt{\frac{\beta}{k}}\right]$
Z_m	$\sqrt{\frac{\pi}{4k+3}}$	$\left[0, \sqrt{\frac{\beta}{k}}\right]$

with $\beta = -\ln(\alpha/2)$.

[3]As we focused on the summation of values, we choose the mean of possible actual value as the center estimate, rather than the center of CI. As a consequence, the standard deviation is corrected to the distance of center estimate and right-bounded of CI.

Next, we derive the following inequalities:

$$\begin{aligned}
& D(\rho_l, \rho'_l) \\
\leq\ & D(\rho_l, [\![S_l]\!](\rho'_{l-1})) + D\left([\![S_l]\!](\rho'_{l-1}), \rho'_l\right) \\
=\ & D([\![S_l]\!](\rho_{l-1}), [\![S_l]\!](\rho'_{l-1})) + D\left([\![S_l]\!](\rho'_{l-1}), \rho'_l\right) \\
\leq\ & D(\rho_{l-1}, \rho'_{l-1}) + D\left([\![S_l]\!](\rho'_{l-1}), \rho'_l\right) \\
& \vdots \\
\leq\ & \sum_{m=1}^{l} D\left([\![S_m]\!](\rho'_{m-1}), \rho'_m\right) \\
\leq\ & \sum_{m=1}^{l} Y_m
\end{aligned}$$

and similarly,

$$A(\rho_l, \rho'_l) \leq \sum_{m=1}^{l} Z_m$$

using the fact that trace-preserving quantum operations (the semantic functions of terminating programs) are contractive for both D and A. Note that all Y_m are independent, so the estimate mean of $\sum_{m=1}^{l} Y_m$ is

$$\frac{l}{k+1} + l\sqrt{\frac{\pi}{4k+3}}$$

and the CI with confident level $1-\alpha$ is [4]

$$\left[0, \frac{l}{k+1} + l\sqrt{\frac{\pi}{4k+3}} + \sqrt{l}\left(\frac{\beta}{k} + \sqrt{\frac{\beta}{k}} - \frac{1}{k+1} - \sqrt{\frac{\pi}{4k+3}}\right)\right].$$

Similarly, we can construct the CI of $\sum_{m=1}^{l} Z_m$:

$$\left[0, l\sqrt{\frac{\pi}{4k+3}} + \sqrt{l}\left(\sqrt{\frac{\beta}{k}} - \sqrt{\frac{\pi}{4k+3}}\right)\right].$$

If k is large (e.g., greater than 100) and choose $\alpha = 0.05$ (the confidence level is 95%), we may simplify above formula and conclude:

1. The 95% CI of $D(\rho_l, \rho'_l)$ is
$$\left[0, \frac{0.9l + \sqrt{l}}{\sqrt{k}}\right],$$
2. The 95% CI of $F(\rho_l, \rho'_l)$ is
$$\left[\cos\frac{0.9l + \sqrt{l}}{\sqrt{k}}, 1\right].$$

Now, if we construct a sequence of subprograms S'_m which takes ρ'_{m-1} as input and output ρ'_m, obviously $S'_1; \cdots; S'_l$ is a bug-free standard program (that passes all assertions with input ρ). Therefore, we complete the proof.

• Even if some S_m is not correct, if the execution of S' does not terminate at **assert**$(\overline{q}_m; P_m)$, then the state after **assert**$(\overline{q}_m; P_m)$ is changed and satisfies P_m, which is actually the correct input for testing S_{m+1}. Therefore, the rest of the execution is still good enough for debugging other errors. ■

[4] The exact bound of CI is generally difficult to calculate. Given a set of X_i with estimate mean $\mathrm{E}X_i$ and CI $(\mathrm{E}X_i - w_i, \mathrm{E}X_i + w_i)$, a simpler way to estimate the CI of summation $\sum_i X_i$ is $\left(\sum_i \mathrm{E}X_i - \sqrt{\sum_i w_i^2}, \sum_i \mathrm{E}X_i + \sqrt{\sum_i w_i^2}\right)$, an interval centered at $\sum_i \mathrm{E}X_i$ with width $\sqrt{\sum_i w_i^2}$, similar to the behavior of standard deviation.

B.1.2 Proof of Lemma 7.3.1

Lemma: *For projection P and density operator ρ, if $\mathrm{tr}(P\rho) \geq 1-\epsilon$, then*

1. $D\Big(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\Big) \leq \epsilon + \sqrt{\epsilon(1-\epsilon)}$.
2. $F\Big(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\Big) \geq \sqrt{1-\epsilon}$.

Proof: 1. For pure state $|\psi\rangle$, we have:

$$\begin{aligned}
\mathrm{tr}\,|P|\psi\rangle\langle\psi|P^\perp| &= \mathrm{tr}\,\sqrt{P|\psi\rangle\langle\psi|P^\perp P^\perp|\psi\rangle\langle\psi|P} \\
&= \sqrt{\langle\psi|P^\perp P^\perp|\psi\rangle}\,\mathrm{tr}\,\sqrt{P|\psi\rangle\langle\psi|P} \\
&= \sqrt{\langle\psi|P^\perp|\psi\rangle}\sqrt{\langle\psi|P|\psi\rangle} \\
&= \sqrt{\mathrm{tr}(P|\psi\rangle\langle\psi|)}\sqrt{\mathrm{tr}(P^\perp|\psi\rangle\langle\psi|)}.
\end{aligned}$$

Therefore, for any density operators ρ with spectral decomposition $\rho = \sum_i p_i|\psi_i\rangle\langle\psi_i|$, we have:

$$\begin{aligned}
\mathrm{tr}\,|P\rho P^\perp| &= \mathrm{tr}\,|P\sum_i p_i|\psi_i\rangle\langle\psi_i|P^\perp| \\
&\leq \sum_i p_i\,\mathrm{tr}\,|P|\psi_i\rangle\langle\psi_i|P^\perp| \\
&= \sum_i \sqrt{p_i\,\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|)}\sqrt{p_i\,\mathrm{tr}(P^\perp|\psi_i\rangle\langle\psi_i|)} \\
&\leq \sqrt{\sum_i p_i\,\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|)}\sqrt{\sum_i p_i\,\mathrm{tr}(P^\perp|\psi_i\rangle\langle\psi_i|)} \\
&= \sqrt{\mathrm{tr}(P\rho)\,\mathrm{tr}(P^\perp\rho)}
\end{aligned}$$

using the Cauchy-Schwarz inequality. Now, it is straightforward to have:

$$\begin{aligned}
& D\Big(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\Big) \\
&= \frac{1}{2}\,\mathrm{tr}\Big|P\rho P + P^{\perp}\rho P + P\rho P^{\perp} + P^{\perp}\rho P^{\perp} - \frac{P\rho P}{\mathrm{tr}(P\rho P)}\Big| \\
&\leq \frac{1}{2}\,\mathrm{tr}\,|P\rho P|\Big|1 - \frac{1}{\mathrm{tr}(P\rho P)}\Big| + \frac{1}{2}|P\rho P^{\perp} + P^{\perp}\rho P| \\
&\quad + \frac{1}{2}|P^{\perp}\rho P^{\perp}| \\
&\leq \frac{1}{2}(1 - \mathrm{tr}(P\rho)) + \mathrm{tr}\,\big|P\sqrt{\rho}\sqrt{\rho}P^{\perp}\big| + \frac{1}{2}\,\mathrm{tr}((I-P)\rho) \\
&\leq \frac{\epsilon}{2} + \sqrt{\mathrm{tr}(P\rho)\,\mathrm{tr}(P^{\perp}\rho)} + \frac{\epsilon}{2} \\
&\leq \epsilon + \sqrt{\epsilon(1-\epsilon)}.
\end{aligned}$$

The restriction of P makes it a slightly stronger than the original one in [246].

2. For pure state $|\psi\rangle$, we have:

$$\begin{aligned}
F\left(|\psi\rangle\langle\psi|, \frac{P|\psi\rangle\langle\psi|P}{\mathrm{tr}(P|\psi\rangle\langle\psi|P)}\right) &= \sqrt{\frac{\langle\psi|P|\psi\rangle\langle\psi|P|\psi\rangle}{\mathrm{tr}(P|\psi\rangle\langle\psi|P)}} \\
&= \sqrt{\mathrm{tr}(P|\psi\rangle\langle\psi|P)}.
\end{aligned}$$

Now, for any density operators ρ with spectral decomposition $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$, we have:

$$\begin{aligned}
& F\left(\rho, \frac{P\rho P}{\mathrm{tr}(P\rho P)}\right) \\
=& F\left(\sum_i p_i|\psi_i\rangle\langle\psi_i|, \sum_i \frac{p_i \,\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|P)}{\mathrm{tr}(P\rho P)} \frac{P|\psi_i\rangle\langle\psi_i|P}{\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|P)}\right) \\
\geq& \sum_i \sqrt{p_i \frac{p_i \,\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|P)}{\mathrm{tr}(P\rho P)}} F\left(|\psi_i\rangle\langle\psi_i|, \frac{P|\psi_i\rangle\langle\psi_i|P}{\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|P)}\right) \\
=& \sum_i \frac{p_i \,\mathrm{tr}(P|\psi_i\rangle\langle\psi_i|P)}{\sqrt{\mathrm{tr}(P\rho P)}} \\
=& \frac{\mathrm{tr}(P\rho P)}{\sqrt{\mathrm{tr}(P\rho P)}} \\
=& \sqrt{1-\epsilon}
\end{aligned}$$

using strong concavity of the fidelity. ■

B.1.3 Proof of Theorem 7.3.2

Theorem: *Assume that all ϵ_i are small ($\epsilon_m \ll 1$). Execute S' for k times ($k \gg l^2$) with input ρ, and we count k_m for the occurrence of error message for assertion* $\mathbf{assert}(\overline{q}_m, P_m)$.

1. *The 95% confidence interval of real ε_m is $[w_m^-, w_m^+]$. Thus, with confidence 95%, if $\epsilon_m < w_m^-$, we conclude S_m is incorrect; and if $\epsilon_m > w_m^+$, we conclude S_m is correct. Here, w_m^-, w_m^+ and w_m^c are $B\left(\alpha, k_m + 1, k - \sum_{i=1}^m k_i\right)$ with $\alpha = 0.025, 0.975$ and 0.5 respectively, where $B(P, A, B)$ is the Pth quantile from a beta distribution with shape parameters A and B.*

2. *If no segment is appeared to be incorrect, i.e., all $\epsilon_m \geq w_m^-$, then after executing the original program S with input ρ, the output state σ approximately satisfies P_l with*

error parameter δ*, i.e.,* $\sigma \models_\delta P_l$*, where* $\delta = \sum_{m=1}^{l} \sqrt{w_m^c} + \sqrt{\sum_{m=1}^{l} (\sqrt{w_m^+} - \sqrt{w_m^c})^2}$.

Proof:

The proof is similar to Appendix B.1.1.

We assume that for the original program S, the state before and after S_m is ρ_{m-1} and ρ_m for $1 \leq m \leq l$; and for the debugging scheme S', the state after $\mathbf{assert}(\overline{q}_m; P_m)$ is ρ'_m for $1 \leq m \leq l$ and set $\rho'_0 = \rho$.

Realize that, the $k - \sum_{i=1}^{m-1} k_i$ executions of assertion $\mathbf{assert}(\overline{q}_m; P_m)$ are $k - \sum_{i=1}^{m-1} k_i$ independent Bernoulli trials with success (report error message) probability $\varepsilon_m = 1 - \mathrm{tr}\big(P_m[\![S_m]\!](\rho'_{m-1})\big)$. With the result that there is m_m success in $k - \sum_{i=1}^{m-1} k_i$ trials, we use the Clopper-Pearson interval to estimate the actual value of probability ε_m. Set confidence level 95%, the CI $[w_m^-, w_m^+]$ is calculated by:

$$w_m^- = B\left(0.025, k_m + 1, k - \sum_{i=1}^{m} k_i\right), \quad w_m^+ = B\left(0.975, k_m + 1, k - \sum_{i=1}^{m} k_i\right),$$

where $B(P, A, B)$ is the Pth quantile from a beta distribution with shape parameters A and B.

Proof of (1): If the desired ϵ_m is smaller than the lower bound w_m^-, i.e., with confidence 95%, the real value of ε_m is larger than w_m^- and also ϵ_m, the segment S_m is incorrect. And if the desired ϵ_m is larger than the upper bound w_m^+, i.e., with confidence 95%, the real value of ε_m is smaller than w_m^+ and also ϵ_m, the segment S_m is correct when the input of S is ρ as the output approximately satisfies P_m with error ε_m less than ϵ_m.

Proof of (2): We set $w_m^c = B\left(0.5, k_m + 1, k - \sum_{i=1}^{m} k_i\right)$. According to Lemma 7.3.1, we know that:

$$D([\![S_m]\!](\rho'_{m-1}), \rho'_m) \leq \varepsilon_m + \sqrt{\varepsilon_m(1 - \epsilon_m)} =: Y_m$$

Since ε_m is a beta distribution and small (because $\epsilon_m \geq w_m^-$ and ϵ_m is small), one can prove that:

1. The mean $\overline{Y_m}$ is smaller than $Y_m^c \triangleq w_m^c + \sqrt{w_m^c(1-w_m^c)}$;
2. $\left[Y_m^- \triangleq w_m^- + \sqrt{w_m^-(1-w_m^-)}, Y_m^+ \triangleq w_m^+ + \sqrt{w_m^+(1-w_m^+)}\right]$ is also the 95% CI of Y_m;
3. $Y_m^+ - Y_m^c > Y_m^c - Y_m^-$;

and thus, it is possible to choose Y_m^c as the center estimate and $Y_m^+ - Y_m^c$ the standard deviation of CI. As a result, the estimate mean of $\sum_{m=1}^l Y_m$ is smaller than $\sum_{m=1}^l Y_m^c$ and thus its CI is

$$\left[\sum_{m=1}^{l} Y_m^c - \sqrt{\sum_{m=1}^{l}(Y_m^+ - Y_m^c)^2},\ \sum_{m=1}^{l} Y_m^c + \sqrt{\sum_{m=1}^{l}(Y_m^+ - Y_m^c)^2}\right].$$

Recall that $D(\rho_l, \rho_l') \leq \sum_{m=1}^l Y_m$, and since ε_m is small, we may ignore the infinitesimal of higher order and approximate the CI of $D(\rho_l, \rho_l')$ as:

$$\left[\sum_{m=1}^{l} \sqrt{w_m^c} - \sqrt{\sum_{m=1}^{l}(\sqrt{w_m^+} - \sqrt{w_m^c})^2},\ \sum_{m=1}^{l} \sqrt{w_m^c} + \sqrt{\sum_{m=1}^{l}(\sqrt{w_m^+} - \sqrt{w_m^c})^2}\right].$$

Note that $\rho_l' \models P_l$ since it is the post-measurement state, we conclude that the output ρ_l of original program S must approximately satisfy P_l with an error at most $\delta \triangleq \sum_{m=1}^l \sqrt{w_m^c} + \sqrt{\sum_{m=1}^l (\sqrt{w_m^+} - \sqrt{w_m^c})^2}$.

■

B.1.4 Proof of Proposition 7.4.2

Proposition: *For projection P with* $\operatorname{rank} P \leq 2^{n-1}$*, there exist projections P_1, P_2, $\cdots$, P_l satisfying* $\operatorname{rank} P_i = 2^{n_i}$ *for all $1 \leq i \leq l$, such that $P = P_1 \cap P_2 \cap \cdots \cap P_l$.*

Theoretically, $l = 2$ is sufficient.

Proof:

After we diagonalize the projection P with the form $U\Lambda U^\dagger$, where the matrix form of Λ is a diagonal matrix

$$\Lambda = \mathrm{diag}(\underbrace{1, 1, \cdots, 1}_{\text{rank } P}, \underbrace{0, 0, \cdots, 0}_{2^n - \text{rank } P}).$$

Choose following two diagonal matrices

$$\Lambda_1 = \mathrm{diag}(\underbrace{1, \cdots, 1}_{2^{n-1}}, 0, \cdots, 0),$$

$$\Lambda_2 = \mathrm{diag}(\underbrace{1, \cdots, 1}_{\text{rank } P}, \underbrace{0, \cdots, 0}_{2^{n-1} - \text{rank } P}, \underbrace{1, \cdots, 1}_{2^{n-1} - \text{rank } P}, \underbrace{0, \cdots, 0}_{\text{rank } P}),$$

which satisfy $\Lambda_1 \cap \Lambda_2 = \Lambda$ and rank Λ_1 = rank $\Lambda_2 = 2^{n-1}$. Therefore, we set $P_1 = U\Lambda_1 U^\dagger$ and $P_2 = U\Lambda_2 U^\dagger$ as desired.

■

Bibliography

[1] P. Benioff, *The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines*, *Journal of statistical physics* **22** (1980), no. 5 563–591.

[2] R. P. Feynman, *Simulating physics with computers*, *Int. J. Theor. Phys* **21** (1982), no. 6/7.

[3] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, *SIAM review* **41** (1999), no. 2 303–332.

[4] J. P. Dowling and G. J. Milburn, *Quantum technology: the second quantum revolution*, *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* **361** (2003), no. 1809 1655–1674.

[5] S. McArdle, S. Endo, A. Aspuru-Guzik, S. C. Benjamin, and X. Yuan, *Quantum computational chemistry*, *Reviews of Modern Physics* **92** (2020), no. 1 015003.

[6] E. Farhi, J. Goldstone, and S. Gutmann, *A quantum approximate optimization algorithm*, *arXiv preprint arXiv:1411.4028* (2014).

[7] L. K. Grover, *A fast quantum mechanical algorithm for database search*, in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, ACM, 1996.

[8] S. Jordan, *Quantum algorithms zoo (2022)*, *URL: http://quantumalgorithmzoo.org* (2022).

[9] M. H. Devoret and R. J. Schoelkopf, *Superconducting circuits for quantum information: an outlook*, *Science* **339** (2013), no. 6124 1169–1174.

[10] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, *Trapped-ion quantum computing: Progress and challenges*, *Applied Physics Reviews* **6** (2019), no. 2 021314, [https://doi.org/10.1063/1.5088164].

[11] S. Slussarenko and G. J. Pryde, *Photonic quantum information processing: A concise review*, *Applied Physics Reviews* **6** (2019), no. 4 041303, [https://doi.org/10.1063/1.5115814].

[12] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, *Quantum supremacy using a programmable superconducting processor*, *Nature* **574** (Oct, 2019) 505–510.

[13] L. S. Madsen, F. Laudenbach, M. F. Askarani, F. Rortais, T. Vincent, J. F. F. Bulmer, F. M. Miatto, L. Neuhaus, L. G. Helt, M. J. Collins, A. E. Lita, T. Gerrits, S. W. Nam, V. D. Vaidya, M. Menotti, I. Dhand, Z. Vernon, N. Quesada, and J. Lavoie, *Quantum computational advantage with a programmable photonic processor*, *Nature* **606** (Jun, 2022) 75–81.

[14] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu, P. Hu, X.-Y. Yang, W.-J. Zhang, H. Li, Y. Li, X. Jiang, L. Gan, G. Yang, L. You, Z. Wang, L. Li, N.-L. Liu, C.-Y. Lu, and J.-W. Pan, *Quantum computational advantage using photons*, *Science* **370** (2020), no. 6523 1460–1463, [https://www.science.org/doi/pdf/10.1126/science.abe8770].

[15] H.-S. Zhong, Y.-H. Deng, J. Qin, H. Wang, M.-C. Chen, L.-C. Peng, Y.-H. Luo, D. Wu, S.-Q. Gong, H. Su, Y. Hu, P. Hu, X.-Y. Yang, W.-J. Zhang, H. Li, Y. Li, X. Jiang, L. Gan, G. Yang, L. You, Z. Wang, L. Li, N.-L. Liu, J. J. Renema, C.-Y. Lu, and J.-W. Pan, *Phase-programmable gaussian boson sampling using stimulated squeezed light*, *Phys. Rev. Lett.* **127** (Oct, 2021) 180502.

[16] H. H. Goldstine and A. Goldstine, *The electronic numerical integrator and computer (eniac)*, *Mathematical Tables and Other Aids to Computation* **2** (1946), no. 15 97–110.

[17] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, *On the co-design of quantum software and hardware*, in *International Conference on Nanoscale Computing and Communication (NANOCOM)*, ACM, 2021.

[18] G. Li, Y. Ding, and Y. 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*,

ASPLOS '19, (New York, NY, USA), p. 1001–1014, Association for Computing Machinery, 2019.

[19] G. Li, Y. Ding, and Y. Xie, *Towards efficient superconducting quantum processor architecture design*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1031–1045, 2020.

[20] G. Li, Y. Shi, and A. Javadi-Abhari, *Software-hardware co-optimization for computational chemistry on superconducting quantum processors*, in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, p. 832–845, IEEE Press, 2021.

[21] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, *Paulihedral: a generalized block-wise compiler optimization framework for quantum simulation kernels*, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 554–569, 2022.

[22] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, *Projection-based runtime assertions for testing and debugging quantum programs*, in *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2020.

[23] G. Li, Y. Ding, and Y. Xie, *Sanq: A simulation framework for architecting noisy intermediate-scale quantum computing system*, *arXiv preprint arXiv:1904.11590* (2019).

[24] G. Li, Y. Ding, and Y. Xie, *Eliminating redundant computation in noisy quantum computing simulation*, in *2020 57nd Design Automation Conference (DAC)*, 2020.

[25] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, *Quantum Computation and Quantum Information, by Michael A. Nielsen, Isaac L. Chuang, Cambridge, UK: Cambridge University Press, 2010* (2010).

[26] R. Van Meter and C. Horsman, *A blueprint for building a quantum computer*, *Communications of the ACM* **56** (2013) 84–93.

[27] F. T. Chong, D. Franklin, and M. Martonosi, *Programming languages and compiler design for realistic quantum hardware*, *Nature* **549** (2017) 180.

[28] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, *An experimental microarchitecture for a superconducting quantum processor*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 813–825, IEEE/ACM, 2017.

[29] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, *Quantum computation by adiabatic evolution*, *arXiv preprint quant-ph/0001106* (2000).

[30] R. Raussendorf, D. E. Browne, and H. J. Briegel, *Measurement-based quantum computation on cluster states*, *Phys. Rev. A* **68** (Aug, 2003) 022312.

[31] Z. Wang, *Topological quantum computation*. No. 112. American Mathematical Soc., 2010.

[32] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, *Elementary gates for quantum computation*, *Physical review A* **52** (1995), no. 5 3457.

[33] L. Henriet, L. Beguin, A. Signoles, T. Lahaye, A. Browaeys, G.-O. Reymond, and C. Jurczak, *Quantum computing with neutral atoms*, *Quantum* **4** (2020) 327.

[34] C. G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K. Bertels, *The engineering challenges in quantum computing*, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 836–845, 2017.

[35] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver, *Superconducting qubits: Current state of play*, *Annual Review of Condensed Matter Physics* **11** (2020), no. 1 369–395, [https://doi.org/10.1146/annurev-conmatphys-031119-050605].

[36] A. Blais, A. L. Grimsmo, S. M. Girvin, and A. Wallraff, *Circuit quantum electrodynamics*, *Rev. Mod. Phys.* **93** (May, 2021) 025005.

[37] IBM, "IBM Q Experience Device." `https://www.research.ibm.com/ibm-q/technology/devices/`, 2018.

[38] J. Chow, O. Dial, and J. Gambetta, *Ibm quantum breaks the 100-qubit processor barrier*, *IBM Research Blog* (2021).

[39] J. Kelly, *A preview of bristlecone, google's new quantum processor*, *Google Research Blog* **5** (2018).

[40] R. Computing, *Rigetti computing announces next-generation 40q and 80q quantum systems*, 2021.

[41] Y. Zhao, Y. Ye, H.-L. Huang, Y. Zhang, D. Wu, H. Guan, Q. Zhu, Z. Wei, T. He, S. Cao, F. Chen, T.-H. Chung, H. Deng, D. Fan, M. Gong, C. Guo, S. Guo, L. Han, N. Li, S. Li, Y. Li, F. Liang, J. Lin, H. Qian, H. Rong, H. Su, L. Sun, S. Wang, Y. Wu, Y. Xu, C. Ying, J. Yu, C. Zha, K. Zhang, Y.-H. Huo, C.-Y. Lu,

C.-Z. Peng, X. Zhu, and J.-W. Pan, *Realization of an error-correcting surface code with superconducting qubits*, *Phys. Rev. Lett.* **129** (Jul, 2022) 030501.

[42] Z. Chen, K. J. Satzinger, J. Atalaya, A. N. Korotkov, A. Dunsworth, D. Sank, C. Quintana, M. McEwen, R. Barends, P. V. Klimov, S. Hong, C. Jones, A. Petukhov, D. Kafri, S. Demura, B. Burkett, C. Gidney, A. G. Fowler, A. Paler, H. Putterman, I. Aleiner, F. Arute, K. Arya, R. Babbush, J. C. Bardin, A. Bengtsson, A. Bourassa, M. Broughton, B. B. Buckley, D. A. Buell, N. Bushnell, B. Chiaro, R. Collins, W. Courtney, A. R. Derk, D. Eppens, C. Erickson, E. Farhi, B. Foxen, M. Giustina, A. Greene, J. A. Gross, M. P. Harrigan, S. D. Harrington, J. Hilton, A. Ho, T. Huang, W. J. Huggins, L. B. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, K. Kechedzhi, S. Kim, A. Kitaev, F. Kostritsa, D. Landhuis, P. Laptev, E. Lucero, O. Martin, J. R. McClean, T. McCourt, X. Mi, K. C. Miao, M. Mohseni, S. Montazeri, W. Mruczkiewicz, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Newman, M. Y. Niu, T. E. O'Brien, A. Opremcak, E. Ostby, B. Pató, N. Redd, P. Roushan, N. C. Rubin, V. Shvarts, D. Strain, M. Szalay, M. D. Trevithick, B. Villalonga, T. White, Z. J. Yao, P. Yeh, J. Yoo, A. Zalcman, H. Neven, S. Boixo, V. Smelyanskiy, Y. Chen, A. Megrant, J. Kelly, and G. Q. AI, *Exponential suppression of bit or phase errors with cyclic error correction*, *Nature* **595** (Jul, 2021) 383–387.

[43] S. Krinner, N. Lacroix, A. Remm, A. Di Paolo, E. Genois, C. Leroux, C. Hellings, S. Lazar, F. Swiadek, J. Herrmann, G. J. Norris, C. K. Andersen, M. Müller, A. Blais, C. Eichler, and A. Wallraff, *Realizing repeated quantum error correction in a distance-three surface code*, *Nature* **605** (May, 2022) 669–674.

[44] E. H. Chen, T. J. Yoder, Y. Kim, N. Sundaresan, S. Srinivasan, M. Li, A. D. Córcoles, A. W. Cross, and M. Takita, *Calibrated decoders for experimental quantum error correction*, *Physical Review Letters* **128** (2022), no. 11 110504.

[45] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, S. Boixo, M. Broughton, B. B. Buckley, D. A. Buell, B. Burkett, N. Bushnell, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, S. Demura, A. Dunsworth, E. Farhi, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, S. Habegger, M. P. Harrigan, A. Ho, S. Hong, T. Huang, W. J. Huggins, L. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, C. Jones, D. Kafri, K. Kechedzhi, J. Kelly, S. Kim, P. V. Klimov, A. Korotkov, F. Kostritsa, D. Landhuis, P. Laptev, M. Lindmark, E. Lucero, O. Martin, J. M. Martinis, J. R. McClean, M. McEwen, A. Megrant, X. Mi, M. Mohseni, W. Mruczkiewicz, J. Mutus, O. Naaman, M. Neeley, C. Neill, H. Neven, M. Y. Niu, T. E. O'Brien, E. Ostby, A. Petukhov, H. Putterman, C. Quintana, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, D. Strain, K. J. Sung, M. Szalay, T. Y. Takeshita, A. Vainsencher, T. White, N. Wiebe, Z. J. Yao, P. Yeh, and A. Zalcman, *Hartree-fock on a superconducting*

qubit quantum computer, *Science* **369** (2020), no. 6507 1084–1089, [https://www.science.org/doi/pdf/10.1126/science.abb9811].

[46] M. P. Harrigan, K. J. Sung, M. Neeley, K. J. Satzinger, F. Arute, K. Arya, J. Atalaya, J. C. Bardin, R. Barends, S. Boixo, M. Broughton, B. B. Buckley, D. A. Buell, B. Burkett, N. Bushnell, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, S. Demura, A. Dunsworth, D. Eppens, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, S. Habegger, A. Ho, S. Hong, T. Huang, L. B. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, C. Jones, D. Kafri, K. Kechedzhi, J. Kelly, S. Kim, P. V. Klimov, A. N. Korotkov, F. Kostritsa, D. Landhuis, P. Laptev, M. Lindmark, M. Leib, O. Martin, J. M. Martinis, J. R. McClean, M. McEwen, A. Megrant, X. Mi, M. Mohseni, W. Mruczkiewicz, J. Mutus, O. Naaman, C. Neill, F. Neukart, M. Y. Niu, T. E. O'Brien, B. O'Gorman, E. Ostby, A. Petukhov, H. Putterman, C. Quintana, P. Roushan, N. C. Rubin, D. Sank, A. Skolik, V. Smelyanskiy, D. Strain, M. Streif, M. Szalay, A. Vainsencher, T. White, Z. J. Yao, P. Yeh, A. Zalcman, L. Zhou, H. Neven, D. Bacon, E. Lucero, E. Farhi, and R. Babbush, *Quantum approximate optimization of non-planar graph problems on a planar superconducting processor*, *Nature Physics* **17** (Mar, 2021) 332–336.

[47] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, *Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets*, *Nature* **549** (2017), no. 7671 242–246.

[48] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien, *A variational eigenvalue solver on a photonic quantum processor*, *Nature Communications* **5** (Jul, 2014) 4213.

[49] 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.

[50] Jeremy Hsu, "CES 2018: 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.

[51] 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.

[52] Rigetti, "The Quantum Processing Unit (QPU)." `https://www.rigetti.com/qpu`, 2018.

[53] J. Preskill, *Quantum computing in the nisq era and beyond*, *arXiv preprint arXiv:1801.00862* (2018).

[54] J. Preskill, *Quantum computing and the entanglement frontier*, *arXiv preprint arXiv:1203.5813* (2012).

[55] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven, *Characterizing quantum supremacy in near-term devices*, *Nature Physics* **14** (2018), no. 6 595.

[56] M. Y. Siraichi, V. F. d. Santos, C. Collange, and F. M. Q. Pereira, *Qubit allocation*, in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), p. 113–125, Association for Computing Machinery, 2018.

[57] D. Maslov, S. M. Falconer, and M. Mosca, *Quantum circuit placement*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27** (2008), no. 4 752–763.

[58] A. Chakrabarti, S. Sur-Kolay, and A. Chaudhury, *Linear nearest neighbor synthesis of reversible circuits by graph partitioning*, *arXiv preprint arXiv:1112.0564* (2011).

[59] A. Shafaei, M. Saeedi, and M. Pedram, *Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures*, in *Proceedings of the 50th Annual Design Automation Conference*, p. 41, ACM, 2013.

[60] A. Shafaei, M. Saeedi, and M. Pedram, *Qubit placement to minimize communication overhead in 2d quantum architectures*, in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pp. 495–500, IEEE, 2014.

[61] R. Wille, A. Lye, and R. Drechsler, *Optimal swap gate insertion for nearest neighbor quantum circuits*, in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pp. 489–494, IEEE, 2014.

[62] A. Lye, R. Wille, and R. Drechsler, *Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits*, in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pp. 178–183, IEEE, 2015.

[63] D. Venturelli, M. Do, E. Rieffel, and J. Frank, *Temporal planning for compilation of quantum approximate optimization circuits*, in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*, pp. 4440–4446, 2017.

[64] D. Venturelli, M. Do, E. Rieffel, and J. Frank, *Compiling quantum circuits to realistic hardware architectures using temporal planners*, *Quantum Science and Technology* **3** (2018), no. 2 025004.

[65] K. E. Booth, M. Do, J. C. Beck, E. Rieffel, D. Venturelli, and J. Frank, *Comparing and integrating constraint programming and temporal planning for quantum circuit compilation*, *arXiv preprint arXiv:1803.06775* (2018).

[66] A. Oddi and R. Rasconi, *Greedy randomized search for scalable compilation of quantum circuits*, in *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 446–461, Springer, 2018.

[67] D. Bhattacharjee and A. Chattopadhyay, *Depth-optimal quantum circuit placement for arbitrary topologies*, *arXiv preprint arXiv:1703.08540* (2017).

[68] M. AlFailakawi, I. Ahmad, and S. Hamdan, *Lnn reversible circuit realization using fast harmony search based heuristic*, in *Asia-Pacific Conference on Computer Science and Electrical Engineering*, 2014.

[69] M. Saeedi, R. Wille, and R. Drechsler, *Synthesis of quantum circuits for linear nearest neighbor architectures*, *Quantum Information Processing* **10** (2011), no. 3 355–377.

[70] C.-C. Lin, S. Sur-Kolay, and N. K. Jha, *Paqcs: Physical design-aware fault-tolerant quantum circuit synthesis*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **23** (2015), no. 7 1221–1234.

[71] R. Wille, O. Keszocze, M. Walter, P. Rohrs, A. Chattopadhyay, and R. Drechsler, *Look-ahead schemes for nearest neighbor optimization of 1d and 2d quantum circuits*, in *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pp. 292–297, IEEE, 2016.

[72] R. R. Shrivastwa, K. Datta, and I. Sengupta, *Fast qubit placement in 2d architecture using nearest neighbor realization*, in *Nanoelectronic and Information Systems (iNIS), 2015 IEEE International Symposium on*, pp. 95–100, IEEE, 2015.

[73] A. Kole, K. Datta, and I. Sengupta, *A heuristic for linear nearest neighbor realization of quantum circuits by swap gate insertion using n-gate lookahead.*, *IEEE J. Emerg. Sel. Topics Circuits Syst.* **6** (2016), no. 1 62–72.

[74] A. Kole, K. Datta, and I. Sengupta, *A new heuristic for n-dimensional nearest neighbor realization of a quantum circuit*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2018), no. 1 182–192.

[75] A. Bhattacharjee, C. Bandyopadhyay, R. Wille, R. Drechsler, and H. Rahaman, *A novel approach for nearest neighbor realization of 2d quantum circuits*, in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, IEEE, 2018.

[76] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernádez, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. D. L. P. González, E. D. L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Ł. Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O'Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A. R. Davila, R. H. P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, *Qiskit: An open-source framework for quantum computing*, 2019.

[77] A. Zulehner, A. Paler, and R. Wille, *Efficient mapping of quantum circuits to the ibm qx architectures*, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 1135–1138, IEEE, 2018.

[78] J. Koch, M. Y. Terri, J. Gambetta, A. A. Houck, D. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, *Charge-insensitive qubit design derived from the cooper pair box*, *Physical Review A* **76** (2007), no. 4 042319.

[79] D. Nigg, M. Mueller, E. A. Martinez, P. Schindler, M. Hennrich, T. Monz, M. A. Martin-Delgado, and R. Blatt, *Quantum computations on a topologically encoded qubit*, *Science* (2014) 1253742.

[80] D. Zajac, T. Hazard, X. Mi, E. Nielsen, and J. Petta, *Scalable gate architecture for a one-dimensional array of semiconductor spin qubits*, *Physical Review Applied* **6** (2016), no. 5 054013.

[81] M. Saffman, T. G. Walker, and K. Mølmer, *Quantum information with rydberg atoms*, *Reviews of Modern Physics* **82** (2010), no. 3 2313.

[82] J. Kelly, R. Barends, A. Fowler, A. Megrant, E. Jeffrey, T. White, D. Sank, J. Mutus, B. Campbell, Y. Chen, and Z. Chen, *State preservation by repetitive error detection in a superconducting quantum circuit*, *Nature* **519** (2015), no. 7541 66.

[83] T. Walter, P. Kurpiers, S. Gasparinetti, P. Magnard, A. Potočnik, Y. Salathé, M. Pechal, M. Mondal, M. Oppliger, C. Eichler, and A. Wallraff, *Rapid*

high-fidelity single-shot dispersive readout of superconducting qubits, *Physical Review Applied* **7** (2017), no. 5 054020.

[84] Y. Chen, C. Neill, P. Roushan, N. Leung, M. Fang, R. Barends, J. Kelly, B. Campbell, Z. Chen, B. Chiaro, and A. Dunsworth, *Qubit architecture with high coherence and fast tunable coupling*, *Physical review letters* **113** (2014), no. 22 220502.

[85] S. S. Tannu and M. 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, (New York, NY, USA), p. 987–999, Association for Computing Machinery, 2019.

[86] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, *Compiler management of communication and parallelism for quantum computation*, *ACM SIGARCH Computer Architecture News* **43** (2015), no. 1 445–456.

[87] A. Paler, I. Polian, K. Nemoto, and S. J. Devitt, *Fault-tolerant, high-level quantum circuits: form, compilation and description*, *Quantum Science and Technology* **2** (2017), no. 2 025003.

[88] A. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, *Optimized surface code communication in superconducting quantum computers*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 692–705, ACM, 2017.

[89] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. Almudever, *Mapping of lattice surgery-based quantum circuits on surface code architectures*, *arXiv preprint arXiv:1805.11127* (2018).

[90] R. W. Floyd, *Algorithm 97: shortest path*, *Communications of the ACM* **5** (1962), no. 6 345.

[91] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, *Revlib: An online resource for reversible functions and reversible circuits*, in *Multiple Valued Logic, 2008. ISMVL 2008. 38th International Symposium on*, pp. 220–225, IEEE, 2008.

[92] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, *Quipper: A scalable quantum programming language*, in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), p. 333–342, Association for Computing Machinery, 2013.

[93] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, *Scaffcc: a framework for compilation and analysis of quantum computing programs*, in *Proceedings of the 11th ACM Conference on Computing Frontiers*, p. 1, ACM, 2014.

[94] Robert Wille, "Mapping to the IBM QX Architectures." `http://iic.jku.at/eda/research/ibm_qx_mapping/`, 2018.

[95] E. A. Sete, W. J. Zeng, and C. T. Rigetti, *A functional architecture for scalable quantum computing*, in *Rebooting Computing (ICRC), IEEE International Conference on*, pp. 1–6, IEEE, 2016.

[96] G. J. Chaitin, *Register allocation spilling via graph coloring*, in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, (New York, NY, USA), p. 98–105, Association for Computing Machinery, 1982.

[97] M. Poletto and V. Sarkar, *Linear scan register allocation*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **21** (1999), no. 5 895–913.

[98] R. M. Tomasulo, *An efficient algorithm for exploiting multiple arithmetic units*, *IBM Journal of research and Development* **11** (1967), no. 1 25–33.

[99] J. L. Hennessy and T. Gross, *Postpass code optimization of pipeline constraints*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **5** (1983), no. 3 422–448.

[100] J. M. Codina, J. Sánchez, and A. González, *A unified modulo scheduling and register allocation technique for clustered processors*, in *pact*, p. 0175, IEEE, 2001.

[101] H. Paik, D. I. Schuster, L. S. Bishop, G. Kirchmair, G. Catelani, A. P. Sears, B. R. Johnson, M. J. Reagor, L. Frunzio, L. I. Glazman, S. M. Girvin, M. H. Devoret, and R. J. Schoelkopf, *Observation of high coherence in josephson junction qubits measured in a three-dimensional circuit qed architecture*, *Phys. Rev. Lett.* **107** (Dec, 2011) 240501.

[102] R. Barends, J. Kelly, A. Megrant, D. Sank, E. Jeffrey, Y. Chen, Y. Yin, B. Chiaro, J. Mutus, C. Neill, P. O'Malley, P. Roushan, J. Wenner, T. C. White, A. N. Cleland, and J. M. Martinis, *Coherent josephson qubit suitable for scalable quantum integrated circuits*, *Phys. Rev. Lett.* **111** (Aug, 2013) 080502.

[103] Y. Shi, N. Leung, P. Gokhale, Z. Rossi, D. I. Schuster, H. Hoffmann, and F. T. Chong, *Optimized compilation of aggregated instructions for realistic quantum computers*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 1031–1044, Association for Computing Machinery, 2019.

[104] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. 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, (New York, NY, USA), p. 1015–1029, Association for Computing Machinery, 2019.

[105] J. P. van Dijk, E. Charbon, and F. Sebastiano, *The electronic interface for quantum processors*, *arXiv preprint arXiv:1811.01693* (2018).

[106] D. C. McKay, S. Filipp, A. Mezzacapo, E. Magesan, J. M. Chow, and J. M. Gambetta, *Universal gate for fixed-frequency qubits via a tunable bus*, *Physical Review Applied* **6** (2016), no. 6 064007.

[107] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, *Validating quantum computers using randomized model circuits*, *arXiv preprint arXiv:1811.12926* (2018).

[108] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete, *Full-stack, real-system quantum computer studies: Architectural comparisons and design insights*, in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 527–540, Association for Computing Machinery, 2019.

[109] S. Rosenblatt, J. Hertzberg, J. Chavez-Garcia, N. Bronn, H. Paik, M. Sandberg, E. Magesan, J. Smolin, J.-B. Yau, V. Adiga, M. Brink, and J. M. Chow, *Enablement of near-term quantum processors by architectural yield engineering*, *Bulletin of the American Physical Society* (2019).

[110] E. Magesan and J. M. Gambetta, *Effective hamiltonian models of the cross-resonance gate*, *arXiv preprint arXiv:1804.04073* (2018).

[111] M. Brink, J. M. Chow, J. Hertzberg, E. Magesan, and S. Rosenblatt, *Device challenges for near term superconducting quantum processors: frequency collisions*, in *2018 IEEE International Electron Devices Meeting (IEDM)*, pp. 6–1, IEEE, 2018.

[112] S. Rosenblatt, J. S. Orcutt, and J. M. Chow, *Laser annealing qubits for optimized frequency allocation*, July 2, 2019. US Patent App. 10/340,438.

[113] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), p. 269–284, Association for Computing Machinery, 2014.

[114] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, *Eie: efficient inference engine on compressed deep neural network*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, IEEE, 2016.

[115] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, *Graphicionado: A high-performance and energy-efficient accelerator for graph analytics*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[116] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, *ACM SIGARCH Computer Architecture News* **43** (2016), no. 3 105–117.

[117] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, *Scaffcc: Scalable compilation and analysis of quantum programs*, *Parallel Computing* **45** (2015) 2–17.

[118] M. Ying and Y. Feng, *Quantum loop programs*, *Acta Informatica* **47** (2010), no. 4 221–250.

[119] M. Ying, N. Yu, Y. Feng, and R. Duan, *Verification of quantum programs*, *Science of Computer Programming* **78** (2013), no. 9 1679–1700.

[120] S. Ying, Y. Feng, N. Yu, and M. Ying, *Reachability probabilities of quantum markov chains*, in *International Conference on Concurrency Theory*, pp. 334–348, Springer, 2013.

[121] K. Honda, *Analysis of quantum entanglement in quantum programs using stabilizer formalism*, *arXiv preprint arXiv:1511.01572* (2015).

[122] S. Perdrix, *Quantum entanglement analysis based on abstract interpretation*, in *International Static Analysis Symposium*, pp. 270–282, Springer, 2008.

[123] P.-L. Dallaire-Demers and F. K. Wilhelm, *Quantum gates and architecture for the quantum simulation of the fermi-hubbard model*, *Physical Review A* **94** (2016), no. 6 062304.

[124] P. J. Liebermann, P.-L. Dallaire-Demers, and F. K. Wilhelm, *Implementation of the ifredkin gate in scalable superconducting architecture for the quantum simulation of fermionic systems*, *arXiv preprint arXiv:1701.07870* (2017).

[125] C. Chamberland, G. Zhu, T. J. Yoder, J. B. Hertzberg, and A. W. Cross, *Topological and subsystem codes on low-degree graphs with flag qubits*, *arXiv preprint arXiv:1907.09528* (2019).

[126] C. Rigetti and M. Devoret, *Fully microwave-tunable universal gates in superconducting qubits with linear couplings and fixed transition frequencies*, *Physical Review B* **81** (2010), no. 13 134507.

[127] S. Sheldon, E. Magesan, J. M. Chow, and J. M. Gambetta, *Procedure for systematically tuning up cross-talk in the cross-resonance gate*, *Physical Review A* **93** (2016), no. 6 060302.

[128] M. Hutchings, J. B. Hertzberg, Y. Liu, N. T. Bronn, G. A. Keefe, M. Brink, J. M. Chow, and B. Plourde, *Tunable superconducting qubits with flux-independent coherence*, *Physical Review Applied* **8** (2017), no. 4 044003.

[129] S. Rosenblatt, J. Hertzberg, M. Brink, J. Chow, J. Gambetta, Z. Leng, A. Houck, J. Nelson, B. Plourde, X. Wu, *et. al.*, *Variability metrics in josephson junction fabrication for quantum computing circuits*, in *APS Meeting Abstracts*, 2017.

[130] J. Ghosh, A. Galiautdinov, Z. Zhou, A. N. Korotkov, J. M. Martinis, and M. R. Geller, *High-fidelity controlled-σ z gate for resonator-based superconducting quantum computers*, *Physical Review A* **87** (2013), no. 2 022309.

[131] D-Wave Systems Inc., "D-Wave System Documentation." `https://docs.dwavesys.com/docs/latest/`, 2018.

[132] A. Ash-Saki, M. Alam, and S. Ghosh, *Qure: Qubit re-allocation in noisy intermediate-scale quantum computers*, in *Proceedings of the 56th Annual Design Automation Conference 2019*, p. 141, ACM, 2019.

[133] F. Jensen, *Introduction to computational chemistry*. John wiley & sons, 2017.

[134] A. Aspuru-Guzik, R. Lindh, and M. Reiher, *The matter simulation (r) evolution*, *ACS central science* **4** (2018), no. 2 144–152.

[135] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, *Elucidating reaction mechanisms on quantum computers*, *Proceedings of the National Academy of Sciences* **114** (2017), no. 29 7555–7560.

[136] R. Babbush, N. Wiebe, J. McClean, J. McClain, H. Neven, and G. K.-L. Chan, *Low-depth quantum simulation of materials*, *Physical Review X* **8** (2018), no. 1 011044.

[137] P. A. M. Dirac, *Quantum mechanics of many-electron systems*, *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* **123** (1929), no. 792 714–733.

[138] Oak Ridge National Lab, "ALCC program awards nearly 6 million summit node hours across 31 projects." https://www.olcf.ornl.gov/2020/08/05/alcc-program-awards-nearly-6-million-summit-node-hours-across-31-projects/. Accessed: 2020-08-16.

[139] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, *The theory of variational hybrid quantum-classical algorithms*, *New Journal of Physics* **18** (2016), no. 2 023023.

[140] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White, P. V. Coveney, P. J. Love, H. Neven, A. Aspuru-Guzik, and J. M. Martinis, *Scalable quantum simulation of molecular energies*, *Phys. Rev. X* **6** (Jul, 2016) 031007.

[141] J. I. Colless, V. V. Ramasesh, D. Dahlen, M. S. Blok, M. Kimchi-Schwartz, J. McClean, J. Carter, W. De Jong, and I. Siddiqi, *Computation of molecular spectra on a quantum processor with an error-resilient algorithm*, *Physical Review X* **8** (2018), no. 1 011021.

[142] Google AI Quantum and Collaborators, *Hartree-fock on a superconducting qubit quantum computer*, *Science* **369** (2020), no. 6507 1084–1089, [https://science.sciencemag.org/content/369/6507/1084.full.pdf].

[143] Y. Shen, X. Zhang, S. Zhang, J.-N. Zhang, M.-H. Yung, and K. Kim, *Quantum implementation of the unitary coupled cluster for simulating molecular electronic structure*, *Physical Review A* **95** (2017), no. 2 020501.

[144] C. Hempel, C. Maier, J. Romero, J. McClean, T. Monz, H. Shen, P. Jurcevic, B. P. Lanyon, P. Love, R. Babbush, A. Aspuru-Guzik, R. Blatt, and C. F. Roos, *Quantum chemistry calculations on a trapped-ion quantum simulator*, *Phys. Rev. X* **8** (Jul, 2018) 031022.

[145] C. Kokail, C. Maier, R. van Bijnen, T. Brydges, M. K. Joshi, P. Jurcevic, C. A. Muschik, P. Silvi, R. Blatt, C. F. Roos, and P. Zoller, *Self-verifying variational quantum simulation of lattice models*, *Nature* **569** (May, 2019) 355–360.

[146] Y. Nam, J.-S. Chen, N. C. Pisenti, K. Wright, C. Delaney, D. Maslov, K. R. Brown, S. Allen, J. M. Amini, J. Apisdorf, K. M. Beck, A. Blinov, V. Chaplin, M. Chmielewski, C. Collins, S. Debnath, K. M. Hudek, A. M. Ducore, M. Keesan, S. M. Kreikemeier, J. Mizrahi, P. Solomon, M. Williams, J. D. Wong-Campos, D. Moehring, C. Monroe, and J. Kim, *Ground-state energy estimation of the*

water molecule on a trapped-ion quantum computer, *npj Quantum Information* **6** (Apr, 2020) 33.

[147] J. Staunstrup and W. Wolf, *Hardware/software co-design: principles and practice*. Springer Science & Business Media, 2013.

[148] J. Lee, W. J. Huggins, M. Head-Gordon, and K. B. Whaley, *Generalized unitary coupled cluster wave functions for quantum computation*, *Journal of chemical theory and computation* **15** (2018), no. 1 311–324.

[149] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall, *An adaptive variational algorithm for exact molecular simulations on a quantum computer*, *Nature communications* **10** (2019), no. 1 1–9.

[150] P.-L. Dallaire-Demers, J. Romero, L. Veis, S. Sim, and A. Aspuru-Guzik, *Low-depth circuit ansatz for preparing correlated fermionic states on a quantum computer*, *Quantum Science and Technology* **4** (2019), no. 4 045005.

[151] I. G. Ryabinkin, T.-C. Yen, S. N. Genin, and A. F. Izmaylov, *Qubit coupled cluster method: a systematic approach to quantum chemistry on a quantum computer*, *Journal of chemical theory and computation* **14** (2018), no. 12 6317–6326.

[152] I. G. Ryabinkin, R. A. Lang, S. N. Genin, and A. F. Izmaylov, *Iterative qubit coupled cluster approach with efficient screening of generators*, *Journal of Chemical Theory and Computation* **16** (2020), no. 2 1055–1063.

[153] H. L. Tang, E. Barnes, H. R. Grimsley, N. J. Mayhall, and S. E. Economou, *qubit-adapt-vqe: An adaptive algorithm for constructing hardware-efficient ansatze on a quantum processor*, *arXiv preprint arXiv:1911.10205* (2019).

[154] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, *Barren plateaus in quantum neural network training landscapes*, *Nature communications* **9** (2018), no. 1 1–6.

[155] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, *Validating quantum computers using randomized model circuits*, *Physical Review A* **100** (2019), no. 3 032328.

[156] P. Jurcevic, A. Javadi-Abhari, L. S. Bishop, I. Lauer, D. F. Bogorin, M. Brink, L. Capelluto, O. Günlük, T. Itoko, N. Kanazawa, *et. al.*, *Demonstration of quantum volume 64 on a superconducting quantum computing system*, *Quantum Science and Technology* **6** (2021), no. 2 025020.

[157] C. Chamberland, G. Zhu, T. J. Yoder, J. B. Hertzberg, and A. W. Cross, *Topological and subsystem codes on low-degree graphs with flag qubits*, *Phys. Rev. X* **10** (Jan, 2020) 011022.

[158] P. Murali, D. C. Mckay, M. Martonosi, and A. Javadi-Abhari, *Software mitigation of crosstalk on noisy intermediate-scale quantum computers*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), p. 1001–1016, Association for Computing Machinery, 2020.

[159] H. Abraham, AduOffei, R. Agarwal, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, M. Amy, E. Arbel, Arijit02, A. Asfaw, A. Avkhadiev, C. Azaustre, AzizNgoueya, A. Banerjee, A. Bansal, P. Barkoutsos, A. Barnawal, G. Barron, G. S. Barron, L. Bello, Y. Ben-Haim, D. Bevenius, A. Bhobe, L. S. Bishop, C. Blank, S. Bolos, S. Bosch, Brandon, S. Bravyi, Bryce-Fuller, D. Bucher, A. Burov, F. Cabrera, P. Calpin, L. Capelluto, J. Carballo, G. Carrascal, A. Chen, C.-F. Chen, E. Chen, J. C. Chen, R. Chen, J. M. Chow, S. Churchill, C. Claus, C. Clauss, R. Cocking, F. Correa, A. J. Cross, A. W. Cross, S. Cross, J. Cruz-Benito, C. Culver, A. D. Córcoles-Gonzales, S. Dague, T. E. Dandachi, M. Daniels, M. Dartiailh, DavideFrr, A. R. Davila, A. Dekusar, D. Ding, J. Doi, E. Drechsler, Drew, E. Dumitrescu, K. Dumon, I. Duran, K. EL-Safty, E. Eastman, G. Eberle, P. Eendebak, D. Egger, M. Everitt, P. M. Fernández, A. H. Ferrera, R. Fouilland, FranckChevallier, A. Frisch, A. Fuhrer, B. Fuller, M. GEORGE, J. Gacon, B. G. Gago, C. Gambella, J. M. Gambetta, A. Gammanpila, L. Garcia, T. Garg, S. Garion, A. Gilliam, A. Giridharan, J. Gomez-Mosquera, Gonzalo, S. de la Puente González, J. Gorzinski, I. Gould, D. Greenberg, D. Grinko, W. Guan, J. A. Gunnels, M. Haglund, I. Haide, I. Hamamura, O. C. Hamido, F. Harkins, V. Havlicek, J. Hellmers, Ł. Herok, S. Hillmich, H. Horii, C. Howington, S. Hu, W. Hu, J. Huang, R. Huisman, H. Imai, T. Imamichi, K. Ishizaki, R. Iten, T. Itoko, JamesSeaward, A. Javadi, A. Javadi-Abhari, W. Javed, Jessica, M. Jivrajani, K. Johns, S. Johnstun, Jonathan-Shoemaker, V. K, T. Kachmann, A. Kale, N. Kanazawa, Kang-Bae, A. Karazeev, P. Kassebaum, J. Kelso, S. King, Knabberjoe, Y. Kobayashi, A. Kovyrshin, R. Krishnakumar, V. Krishnan, K. Krsulich, P. Kumkar, G. Kus, R. LaRose, E. Lacal, R. Lambert, J. Lapeyre, J. Latone, S. Lawrence, C. Lee, G. Li, D. Liu, P. Liu, Y. Maeng, K. Majmudar, A. Malyshev, J. Manela, J. Marecek, M. Marques, D. Maslov, D. Mathews, A. Matsuo, D. T. McClure, C. McGarry, D. McKay, D. McPherson, S. Meesala, T. Metcalfe, M. Mevissen, A. Meyer, A. Mezzacapo, R. Midha, Z. Minev, A. Mitchell, N. Moll, J. Montanez, G. Monteiro, M. D. Mooring, R. Morales, N. Moran, M. Motta, MrF, P. Murali, J. Müggenburg, D. Nadlinger, K. Nakanishi, G. Nannicini, P. Nation, E. Navarro, Y. Naveh, S. W. Neagle, P. Neuweiler, J. Nicander, P. Niroula, H. Norlen, NuoWenLei, L. J. O'Riordan, O. Ogunbayo, P. Ollitrault, R. Otaolea, S. Oud, D. Padilha, H. Paik, S. Pal, Y. Pang, V. R. Pascuzzi, S. Perriello, A. Phan, F. Piro, M. Pistoia, C. Piveteau, P. Pocreau, A. Pozas-Kerstjens, M. Prokop, V. Prutyanov, D. Puzzuoli, J. Pérez, Quintiii, R. I. Rahman, A. Raja,

N. Ramagiri, A. Rao, R. Raymond, R. M.-C. Redondo, M. Reuter, J. Rice, M. Riedemann, M. L. Rocca, D. M. Rodríguez, RohithKarur, M. Rossmannek, M. Ryu, T. SAPV, SamFerracin, M. Sandberg, H. Sandesara, R. Sapra, H. Sargsyan, A. Sarkar, N. Sathaye, B. Schmitt, C. Schnabel, Z. Schoenfeld, T. L. Scholten, E. Schoute, J. Schwarm, I. F. Sertage, K. Setia, N. Shammah, Y. Shi, A. Silva, A. Simonetto, N. Singstock, Y. Siraichi, I. Sitdikov, S. Sivarajah, M. B. Sletfjerding, J. A. Smolin, M. Soeken, I. O. Sokolov, I. Sokolov, SooluThomas, Starfish, D. Steenken, M. Stypulkoski, S. Sun, K. J. Sung, H. Takahashi, T. Takawale, I. Tavernelli, C. Taylor, P. Taylour, S. Thomas, M. Tillet, M. Tod, M. Tomasik, E. de la Torre, K. Trabing, M. Treinish, TrishaPe, D. Tulsi, W. Turner, Y. Vaknin, C. R. Valcarce, F. Varchon, A. C. Vazquez, V. Villar, D. Vogt-Lee, C. Vuillot, J. Weaver, J. Weidenfeller, R. Wieczorek, J. A. Wildstrom, E. Winston, J. J. Woehr, S. Woerner, R. Woo, C. J. Wood, R. Wood, S. Wood, S. Wood, J. Wootton, D. Yeralin, D. Yonge-Mallo, R. Young, J. Yu, C. Zachow, L. Zdanski, H. Zhang, C. Zoufal, and M. Čepulkovskis, *Qiskit: An open-source framework for quantum computing*, 2019.

[160] R. S. Smith, E. C. Peterson, M. Skilbeck, and E. Davis, *An open-source, industrial-strength optimizing compiler for quantum programs*, *Quantum Science and Technology* (2020).

[161] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, *t|ket⟩: a retargetable compiler for NISQ devices*, *Quantum Science and Technology* **6** (nov, 2020) 014003.

[162] M. Soeken and M. K. Thomsen, *White dots do matter: Rewriting reversible logic circuits*, in *Proceedings of the 5th International Conference on Reversible Computation*, RC'13, (Berlin, Heidelberg), p. 196–208, Springer-Verlag, 2013.

[163] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, *Automated optimization of large quantum circuits with continuous parameters*, *npj Quantum Information* **4** (May, 2018) 23.

[164] D. Maslov, G. W. Dueck, D. M. Miller, and C. Negrevergne, *Quantum circuit simplification and level compaction*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27** (2008), no. 3 436–444.

[165] Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, S. Wouters, and G. K. Chan, *Pyscf: the python-based simulations of chemistry framework*, 2017.

[166] T. Helgaker, P. Jorgensen, and J. Olsen, *Molecular electronic-structure theory*. John Wiley & Sons, 2014.

[167] J. Paldus, M. Takahashi, and B. W. H. Cho, *Degeneracy and coupled-cluster approaches*, *International Journal of Quantum Chemistry* **26** (1984), no. S18 237–244, [https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560260824].

[168] R. J. Bartlett, *Coupled-cluster approach to molecular structure and spectra: a step toward predictive quantum chemistry*, *The Journal of Physical Chemistry* **93** (Mar, 1989) 1697–1708.

[169] I. Quantum, *IBM Quantum Experience*, August 2020.

[170] W. J. Hehre, R. F. Stewart, and J. A. Pople, *self-consistent molecular-orbital methods. i. use of gaussian expansions of slater-type atomic orbitals*, *The Journal of Chemical Physics* **51** (1969), no. 6 2657–2664.

[171] P. Jordan and E. Wigner, *Über das paulische äquivalenzverbot*, *Zeitschrift für Physik* **47** (1928), no. 9-10 631–651.

[172] D. Kraft, *A software package for sequential quadratic programming*, .

[173] M. Malekakhlagh, E. Magesan, and D. C. McKay, *First-principles analysis of cross-resonance gate operation*, *Phys. Rev. A* **102** (Oct, 2020) 042605.

[174] J. Hubbard, *Electron correlations in narrow energy bands*, *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* **276** (1963), no. 1365 238–257.

[175] T. Patel and D. Tiwari, *Veritas: accurately estimating the correct output on noisy intermediate-scale quantum computers*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020* (C. Cuicchi, I. Qualters, and W. T. Kramer, eds.), p. 15, IEEE/ACM, 2020.

[176] S. S. Tannu and M. Qureshi, *Ensemble of diverse mappings: Improving reliability of quantum computers by orchestrating dissimilar mistakes*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 253–265, Association for Computing Machinery, 2019.

[177] S. S. Tannu and M. K. Qureshi, *Mitigating measurement errors in quantum computers by exploiting state-dependent bias*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 279–290, 2019.

[178] A. Eddins, M. Motta, T. P. Gujarati, S. Bravyi, A. Mezzacapo, C. Hadfield, and S. Sheldon, *Doubling the size of quantum simulators by entanglement forging*, *arXiv preprint arXiv:2104.10220* (2021).

[179] A. Jena, S. Genin, and M. Mosca, *Pauli partitioning with respect to gate sets*, *arXiv preprint arXiv:1907.07859* (2019).

[180] P. Gokhale, O. Angiuli, Y. Ding, K. Gui, T. Tomesh, M. Suchara, M. Martonosi, and F. T. Chong, *Minimizing state preparations in variational quantum eigensolver by partitioning into commuting families*, *arXiv preprint arXiv:1907.13623* (2019).

[181] V. Verteletskyi, T. Yen, and A. Izmaylov, *Measurement optimization in the variational quantum eigensolver using a minimum clique cover. arxiv 2019*, *arXiv preprint arXiv:1907.03358* (2019).

[182] T.-C. Yen, V. Verteletskyi, and A. F. Izmaylov, *Measuring all compatible operators in one series of single-qubit measurements using unitary transformations*, *Journal of Chemical Theory and Computation* **16** (2020), no. 4 2400–2409.

[183] A. F. Izmaylov, T.-C. Yen, R. A. Lang, and V. Verteletskyi, *Unitary partitioning approach to the measurement problem in the variational quantum eigensolver method*, *Journal of Chemical Theory and Computation* **16** (2019), no. 1 190–195.

[184] A. M. Childs, E. Schoute, and C. M. Unsal, *Circuit transformations for quantum architectures*, *arXiv preprint arXiv:1902.09102* (2019).

[185] A. Cowtan, W. Simmons, and R. Duncan, *A generic compilation strategy for the unitary coupled cluster ansatz*, *arXiv preprint arXiv:2007.10515* (2020).

[186] A. M.-v. de Griend and R. Duncan, *Architecture-aware synthesis of phase polynomials for nisq devices*, *arXiv preprint arXiv:2004.06052* (2020).

[187] K. R. Brown, J. Kim, and C. Monroe, *Co-designing a scalable quantum computer with trapped atomic ions*, *npj Quantum Information* **2** (2016), no. 1 1–10.

[188] P. Murali, D. M. Debroy, K. R. Brown, and M. Martonosi, *Architecting noisy intermediate-scale trapped ion quantum computers*, in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, p. 529–542, IEEE Press, 2020.

[189] S. Lloyd, *Universal quantum simulators*, *Science* **273** (1996), no. 5278 1073–1078, [https://www.science.org/doi/pdf/10.1126/science.273.5278.1073].

[190] D. S. Abrams and S. Lloyd, *Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors*, *Phys. Rev. Lett.* **83** (Dec, 1999) 5162–5165.

[191] A. W. Harrow, A. Hassidim, and S. Lloyd, *Quantum algorithm for linear systems of equations*, *Phys. Rev. Lett.* **103** (Oct, 2009) 150502.

[192] S. Lloyd, M. Mohseni, and P. Rebentrost, *Quantum principal component analysis*, *Nature Physics* **10** (Sep, 2014) 631–633.

[193] P. Rebentrost, M. Mohseni, and S. Lloyd, *Quantum support vector machine for big data classification*, *Phys. Rev. Lett.* **113** (Sep, 2014) 130503.

[194] R. S. Smith, E. C. Peterson, M. G. Skilbeck, and E. J. Davis, *An open-source, industrial-strength optimizing compiler for quantum programs*, *Quantum Science and Technology* **5** (jul, 2020) 044001.

[195] M. B. Hastings, D. Wecker, B. Bauer, and M. Troyer, *Improving quantum algorithms for quantum chemistry*, *Quantum Info. Comput.* **15** (jan, 2015) 1–21.

[196] K. Gui, T. Tomesh, P. Gokhale, Y. Shi, F. T. Chong, M. Martonosi, and M. Suchara, *Term grouping and travelling salesperson for digital quantum simulation*, *arXiv preprint arXiv:2001.05983* (2020).

[197] E. van den Berg and K. Temme, *Circuit optimization of Hamiltonian simulation by simultaneous diagonalization of Pauli clusters*, *Quantum* **4** (Sept., 2020) 322.

[198] A. Cowtan, S. Dilkes, R. Duncan, W. Simmons, and S. Sivarajah, *Phase gadget synthesis for shallow circuits*, *Electronic Proceedings in Theoretical Computer Science* **318** (May, 2020) 213–228.

[199] V. Vandaele, S. Martiel, and T. G. de Brugière, *Phase polynomials synthesis algorithms for nisq architectures and beyond*, *arXiv preprint arXiv:2104.00934* (2021).

[200] M. Alam, A. Ash-Saki, and S. Ghosh, *Circuit compilation methodologies for quantum approximate optimization algorithm*, in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 215–228, IEEE, 2020.

[201] B. Tan and J. Cong, *Optimal layout synthesis for quantum computing*, in *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[202] L. Lao and D. Browne, *2qan: A quantum compiler for 2-local qubit hamiltonian simulation algorithms*, *arXiv preprint arXiv:2108.02099* (2021).

[203] M. Suzuki, *Generalized trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems*, *Communications in Mathematical Physics* **51** (1976), no. 2 183–190.

[204] P. Jordan and E. Wigner, *Über das paulische äquivalenzverbot*, *Zeitschrift für Physik* **47** (Sep, 1928) 631–651.

[205] S. B. Bravyi and A. Y. Kitaev, *Fermionic quantum computation*, *Annals of Physics* **298** (2002), no. 1 210–226.

[206] M. Alam, A. Ash-Saki, and S. Ghosh, *An efficient circuit compilation flow for quantum approximate optimization algorithm*, in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[207] M. Alam, A. Ash-Saki, J. Li, A. Chattopadhyay, and S. Ghosh, *Noise resilient compilation policies for quantum approximate optimization algorithm*, in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–7, 2020.

[208] H. F. Trotter, *On the product of semi-groups of operators*, *Proceedings of the American Mathematical Society* **10** (1959), no. 4 545–551.

[209] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles, *Variational quantum algorithms*, *Nature Reviews Physics* **3** (Sep, 2021) 625–644.

[210] Z. H. Saleem, B. Tariq, and M. Suchara, *Approaches to constrained quantum approximate optimization*, *arXiv preprint arXiv:2010.06660* (2020).

[211] B. T. Gard, L. Zhu, G. S. Barron, N. J. Mayhall, S. E. Economou, and E. Barnes, *Efficient symmetry-preserving state preparation circuits for the variational quantum eigensolver algorithm*, *npj Quantum Information* **6** (Jan, 2020) 10.

[212] A. Tranter, P. J. Love, F. Mintert, and P. V. Coveney, *A comparison of the bravyi–kitaev and jordan–wigner transformations for the quantum simulation of quantum chemistry*, *Journal of Chemical Theory and Computation* **14** (2018), no. 11 5617–5630, [https://doi.org/10.1021/acs.jctc.8b00450]. PMID: 30189144.

[213] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, *Surface codes: Towards practical large-scale quantum computation*, *Phys. Rev. A* **86** (Sep, 2012) 032324.

[214] D. Maslov, *Optimal and asymptotically optimal nct reversible circuits by the gate types*, *Quantum Info. Comput.* **16** (oct, 2016) 1096–1112.

[215] S. Nishio, Y. Pan, T. Satoh, H. Amano, and R. V. Meter, *Extracting success from ibm's 20-qubit machines using error-aware compilation*, *J. Emerg. Technol. Comput. Syst.* **16** (May, 2020).

[216] A. Zulehner, A. Paler, and R. Wille, *An efficient methodology for mapping quantum circuits to the ibm qx architectures*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **38** (2019), no. 7 1226–1236.

[217] P. Gokhale, A. Javadi-Abhari, N. Earnest, Y. Shi, and F. T. Chong, *Optimized quantum compilation for near-term algorithms with openpulse*, in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 186–200, 2020.

[218] J. Cheng, H. Deng, and X. Qia, *Accqoc: Accelerating quantum optimal control based pulse generation*, in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 543–555, 2020.

[219] X.-C. Wu, D. M. Debroy, Y. Ding, J. M. Baker, Y. Alexeev, K. R. Brown, and F. T. Chong, *Tilt: Achieving higher fidelity on a trapped-ion linear-tape quantum computing architecture*, .

[220] J. M. Arrazola, V. Bergholm, K. Brádler, T. R. Bromley, M. J. Collins, I. Dhand, A. Fumagalli, T. Gerrits, A. Goussev, L. G. Helt, J. Hundal, T. Isacsson, R. B. Israel, J. Izaac, S. Jahangiri, R. Janik, N. Killoran, S. P. Kumar, J. Lavoie, A. E. Lita, D. H. Mahler, M. Menotti, B. Morrison, S. W. Nam, L. Neuhaus, H. Y. Qi, N. Quesada, A. Repingon, K. K. Sabapathy, M. Schuld, D. Su, J. Swinarton, A. Száva, K. Tan, P. Tan, V. D. Vaidya, Z. Vernon, Z. Zabaneh, and Y. Zhang, *Quantum circuits with many photons on a programmable nanophotonic chip*, *Nature* **591** (Mar, 2021) 54–60.

[221] G. Brassard and P. Hoyer, *An exact quantum polynomial-time algorithm for simon's problem*, in *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*, pp. 12–23, 1997.

[222] S. Lloyd, M. Mohseni, and P. Rebentrost, *Quantum algorithms for supervised and unsupervised machine learning*, *arXiv preprint arXiv:1307.0411* (2013).

[223] M. Amy and V. Gheorghiu, *staq—a full-stack quantum processing toolkit*, *Quantum Science and Technology* **5** (jun, 2020) 034016.

[224] N. Khammassi, I. Ashraf, J. V. Someren, R. Nane, A. M. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever, *Openql: A portable quantum programming framework for quantum accelerators*, *J. Emerg. Technol. Comput. Syst.* **18** (dec, 2021).

[225] A. McCaskey and T. Nguyen, *A mlir dialect for quantum assembly languages*, in *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, (Los Alamitos, CA, USA), pp. 255–264, IEEE Computer Society, oct, 2021.

[226] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *Open quantum assembly language*, *arXiv preprint arXiv:1707.03429* (2017).

[227] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, *arXiv preprint arXiv:1608.03355* (2016).

[228] A. Kissinger and J. van de Wetering, *Pyzx: Large scale automated diagrammatic reasoning*, *Electronic Proceedings in Theoretical Computer Science* **318** (May, 2020) 229–241.

[229] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, J. Smolin, J. M. Gambetta, and B. R. Johnson, *Openqasm 3: A broader and deeper quantum assembly language*, *arXiv preprint arXiv:2104.14722* (2021).

[230] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, *On the co-design of quantum software and hardware*, in *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication*, NANOCOM '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[231] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, *Q#: Enabling scalable quantum computing and development with a high-level dsl*, in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, p. 7, ACM, 2018.

[232] J. Paykin, R. Rand, and S. Zdancewic, *Qwire: A core language for quantum circuits*, in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 846–858, ACM, 2017.

[233] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, F. Chong, M. Martonosi, M. Suchara, K. Brown, M. Pedram, and T. Brun, *2012. scaffold: Quantum programming language*, tech. rep., Technical Report TR-934-12. Princeton University, 2012.

[234] Rigetti Forest team, "Forest SDK." `https://www.rigetti.com/forest`, 2019.

[235] Google, "Announcing Cirq: An Open Source Framework for NISQ Algorithms." `https://ai.googleblog.com/2018/07/announcing-cirq-open-source-framework.html`, 2018.

[236] Y. Huang and M. Martonosi, *Qdb: From quantum algorithms towards correct quantum programs*, in *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[237] Y. Huang and M. Martonosi, *Statistical assertions for validating patterns and finding bugs in quantum programs*, in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 541–553, ACM, 2019.

[238] IBM, "Gate and operation specification for quantum circuits." `https://github.com/Qiskit/openqasm`, 2019.

[239] Rigetti, "A Python library for quantum programming using Quil.." `https://github.com/rigetti/pyquil`, 2019.

[240] J. Liu, G. T. Byrd, and H. Zhou, *Quantum circuits for dynamic runtime assertions in quantum computation*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1017–1030, 2020.

[241] W. K. Wootters and W. H. Zurek, *A single quantum cannot be cloned*, *Nature* **299** (1982), no. 5886 802.

[242] G. Birkhoff and J. Von Neumann, *The logic of quantum mechanics*, *Annals of mathematics* (1936) 823–843.

[243] Y. Li and M. Ying, *Debugging quantum processes using monitoring measurements*, *Phys. Rev. A* **89** (Apr, 2014) 042338.

[244] M. Ying, *Floyd–hoare logic for quantum programs*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **33** (2011), no. 6 19.

[245] M. Ying, *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[246] A. Winter, *Coding theorem and strong converse for quantum channels*, *IEEE Transactions on Information Theory* **45** (Nov, 1999) 2481–2485.

[247] J. Chen, F. Zhang, C. Huang, M. Newman, and Y. Shi, *Classical simulation of intermediate-size quantum circuits*, *arXiv preprint arXiv:1805.01450* (2018).

[248] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks, *A verified optimizer for quantum circuits*, *arXiv preprint arXiv:1912.02250* (2019).

[249] Y. Shi, X. Li, R. Tao, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu, *Contract-based verification of a realistic quantum compiler*, *arXiv preprint arXiv:1908.08963* (2019).

[250] R. Rand, J. Paykin, and S. Zdancewic, *Qwire practice: Formal verification of quantum circuits in coq*, *arXiv preprint arXiv:1803.00699* (2018).

[251] N. Linden, S. Popescu, and W. Wootters, *Almost every pure state of three qubits is completely determined by its two-particle reduced density matrices*, *Phys. Rev. Lett.* **89** (Oct, 2002) 207901.

[252] J. Chen, Z. Ji, B. Zeng, and D. L. Zhou, *From ground states to local hamiltonians*, *Phys. Rev. A* **86** (Aug, 2012) 022339.

[253] T. Xin, D. Lu, J. Klassen, N. Yu, Z. Ji, J. Chen, X. Ma, G. Long, B. Zeng, and R. Laflamme, *Quantum state tomography via reduced density matrices*, *Phys. Rev. Lett.* **118** (Jan, 2017) 020401.

[254] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, *Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance*, *Nature* **414** (2001), no. 6866 883.

[255] L. Zhou, N. Yu, and M. Ying, *An applied quantum hoare logic*, in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1149–1162, ACM, 2019.

[256] V. B. Braginsky, Y. I. Vorontsov, and K. S. Thorne, *Quantum nondemolition measurements*, *Science* **209** (1980), no. 4456 547–557.

[257] G. Kalmbach, *Orthomodular lattices*, vol. 18. Academic Pr, 1983.

[258] O. Brunet and P. Jorrand, *Dynamic quantum logic for quantum programs*, *International Journal of Quantum Information* **2** (2004), no. 01 45–54.

[259] M. Ying, R. Duan, Y. Feng, and Z. Ji, *Predicate transformer semantics of quantum programs*, *Semantic Techniques in Quantum Computation* **8** (2010) 311–360.

[260] D. Unruh, *Quantum relational hoare logic*, *Proceedings of the ACM on Programming Languages* **3** (2019), no. POPL 33.

[261] N. Yu, *Quantum temporal logic*, 2019.

[262] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, *A heterogeneous quantum computer architecture*, in *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 323–330, ACM, 2016.

[263] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, *eqasm: An executable quantum instruction set architecture*, *arXiv preprint arXiv:1808.02449* (2018).

[264] N. M. Linke, D. Maslov, M. Roetteler, S. Debnath, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe, *Experimental comparison of two quantum computing architectures*, *Proceedings of the National Academy of Sciences* **114** (2017), no. 13 3305–3310.

[265] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, *gem5-gpu: A heterogeneous cpu-gpu simulator*, *IEEE Computer Architecture Letters* **14** (2015), no. 1 34–36.

[266] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, *Analyzing cuda workloads using a detailed gpu simulator*, in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, IEEE, 2009.

[267] G. F. Viamontes, I. L. Markov, and J. P. Hayes, *Quantum circuit simulation*. Springer Science & Business Media, 2009.

[268] I. L. Markov and Y. Shi, *Simulating quantum computation by contracting tensor networks*, *SIAM Journal on Computing* **38** (2008), no. 3 963–981.

[269] S. Aaronson and D. Gottesman, *Improved simulation of stabilizer circuits*, *Physical Review A* **70** (2004), no. 5 052328.

[270] S. Anders and H. J. Briegel, *Fast simulation of stabilizer circuits using a graph-state representation*, *Physical Review A* **73** (2006), no. 2 022334.

[271] A. Zulehner and R. Wille, *Advanced simulation of quantum computations*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[272] G. F. Viamontes, I. L. Markov, and J. P. Hayes, *High-performance quidd-based simulation of quantum circuits*, in *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, p. 21354, IEEE Computer Society, 2004.

[273] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, *Qx: A high-performance quantum computer simulation platform*, in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 464–469, IEEE, 2017.

[274] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, *qhipster: the quantum high performance software testing environment*, *arXiv preprint arXiv:1601.07195* (2016).

[275] D. Wecker and K. M. Svore, *Liqui—¿: A software design architecture and domain-specific language for quantum computing*, *arXiv preprint arXiv:1402.4467* (2014).

[276] D. S. Steiger, T. Häner, and M. Troyer, *Projectq: an open source software framework for quantum computing*, *Quantum* **2** (2018) 49.

[277] Brian Tarasinski, "quantumsim, A GPU-accellerated full density matrix simulator of quantum circuits." `https://gitlab.com/quantumsim/quantumsim`, 2018.

[278] T. Jones, A. Brown, I. Bush, and S. Benjamin, *Quest and high performance simulation of quantum computers*, *arXiv preprint arXiv:1802.08032* (2018).

[279] X. FU, *Quantum Control Architecture: Bridging the Gap between Quantum Software and Hardware*. PhD thesis, Delft University of Technology, 2018.

[280] M. Zhang, *Qumasim: A quantum architecture simulation and verification platform*, .

[281] Antonio Corcoles, Maika Takita, Ken Inoue, Scott Lekuch, Abhinav Kandala, Jay Gambetta, Jerry M. Chow, " Integration of classical electronics for quantum computing tasks in superconducting qubit systems." APS March Meeting, 2019.

[282] B. Lekitsch, S. Weidt, A. G. Fowler, K. Mølmer, S. J. Devitt, C. Wunderlich, and W. K. Hensinger, *Blueprint for a microwave trapped ion quantum computer*, *Science Advances* **3** (2017), no. 2 e1601540.

[283] D. Loss and D. P. DiVincenzo, *Quantum computation with quantum dots*, *Physical Review A* **57** (1998), no. 1 120.

[284] J. M. Chow, *Quantum information processing with superconducting qubits*. Yale University, 2010.

[285] J. J. García-Ripoll, P. Zoller, and J. I. Cirac, *Speed optimized two-qubit gates with laser coherent control techniques for ion trap quantum computing*, *Physical Review Letters* **91** (2003), no. 15 157901.

[286] Amit Vainsencher, Ben Chiaro, Roberto Collins, Brooks Foxen, Evan Jeffrey, Erik Lucero, Matthew McEwen, Daniel Sank, John M Martinis , "Superconducting qubit control electronics - Part 1/2: system overview and control hardware." APS March Meeting, 2019.

[287] Glenn Jones, Deanna Abrams, Stephan Brown, Lauren Capelluto, Schuyler Fried, Sabrina Hong, Blake Johnson, Rob Lion, Adam Mocarski, Mike Pelstring, Chad Rigetti, Damon Russell, Michael Rust, Colm Ryan, Diego Scarabelli, Rodney Sinclair, Prasahnt Sivarajah, Chloe Song, Alexa N Staley, John Stevenson, Mark Suska, Nima Taie-Nobarie, Celena Tanguay, Nikolas Tezak, Stefan Turkowski , " Scalable instrumentation for general purpose quantum computers." APS March Meeting, 2019.

[288] IBM, "IBM Quantum Device Backend Information." `https://github.com/Qiskit/ibmq-device-information`, 2018.

[289] Rigetti Computing, " Noise and Quantum Computation." `https://pyquil.readthedocs.io/en/stable/noise.html`, 2019.

[290] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, and H. Neven, *Simulation of low-depth quantum circuits as complex undirected graphical models*, *arXiv preprint arXiv:1712.05384* (2017).

[291] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff, *Breaking the 49-qubit barrier in the simulation of quantum circuits*, *arXiv preprint arXiv:1710.05867* (2017).

[292] R. Li, B. Wu, M. Ying, X. Sun, and G. Yang, *Quantum supremacy circuit simulation on sunway taihulight*, *arXiv preprint arXiv:1804.04797* (2018).

[293] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn, A. Kandala, A. Mezzacapo, P. Müller, W. Riess, G. Salis, J. Smolin, I. Tavernelli, and K. Temme, *Quantum optimization using variational algorithms on near-term quantum devices*, *Quantum Science and Technology* **3** (2018), no. 3 030503.

[294] E. Bernstein and U. Vazirani, *Quantum complexity theory*, *SIAM Journal on computing* **26** (1997), no. 5 1411–1473.

[295] E. Knill, D. Leibfried, R. Reichle, J. Britton, R. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland, *Randomized benchmarking of quantum gates*, *Physical Review A* **77** (2008), no. 1 012307.

[296] J. Joo, Y.-J. Park, S. Oh, and J. Kim, *Quantum teleportation via a w state*, *New Journal of Physics* **5** (2003), no. 1 136.

[297] UFMG Compilers Laboratory, "QUbit Allocation, The Enfield Project." `http://cuda.dcc.ufmg.br/enfield/`, 2018.

[298] D. Aharonov, A. Kitaev, and J. Preskill, *Fault-tolerant quantum computation with long-range correlated noise*, *Physical review letters* **96** (2006), no. 5 050504.

[299] J. Preskill, *Sufficient condition on noise correlations for scalable quantum computing*, *arXiv preprint arXiv:1207.6131* (2012).

[300] J. Werschnik and E. Gross, *Quantum optimal control theory*, *Journal of Physics B: Atomic, Molecular and Optical Physics* **40** (2007), no. 18 R175.

[301] N. Leung, M. Abdelhafez, J. Koch, and D. Schuster, *Speedup for quantum optimal control from automatic differentiation based on graphics processing units*, *Physical Review A* **95** (2017), no. 4 042318.

[302] X. Qin, Z. Shi, Y. Xie, L. Wang, X. Rong, W. Jia, W. Zhang, and J. Du, *An integrated device with high performance multi-function generators and time-to-digital convertors*, *Review of Scientific Instruments* **88** (2017), no. 1 014702.

[303] C. A. Ryan, B. R. Johnson, D. Ristè, B. Donovan, and T. A. Ohki, *Hardware for dynamic quantum computing*, *Review of Scientific Instruments* **88** (2017), no. 10 104703.

[304] Y. Salathé, P. Kurpiers, T. Karg, C. Lang, C. K. Andersen, A. Akin, S. Krinner, C. Eichler, and A. Wallraff, *Low-latency digital signal processing for feedback and feedforward in quantum computing and communication*, *Physical Review Applied* **9** (2018), no. 3 034011.

[305] J. Lin, F.-T. Liang, Y. Xu, L.-H. Sun, C. Guo, S.-K. Liao, and C.-Z. Peng, *High performance and scalable awg for superconducting quantum computing*, *arXiv preprint arXiv:1806.03660* (2018).

[306] L. Riesebos, X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels, *Pauli frames for quantum computer architectures*, in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 76, ACM, 2017.

[307] J. van Dijk, A. Vladimirescu, M. Babaie, E. Charbon, and F. Sebastiano, *A co-design methodology for scalable quantum processors and their classical electronic interface*, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 573–576, IEEE, 2018.

[308] F. Pan and P. Zhang, *Simulation of quantum circuits using the big-batch tensor network method*, *Phys. Rev. Lett.* **128** (Jan, 2022) 030501.

[309] J. Fenn and M. Raskino, *Mastering the hype cycle: how to choose the right innovation at the right time*. Harvard Business Press, 2008.

[310] C. J. CLOPPER and E. S. PEARSON, *THE USE OF CONFIDENCE OR FIDUCIAL LIMITS ILLUSTRATED IN THE CASE OF THE BINOMIAL*, *Biometrika* **26** (12, 1934) 404–413, [https://academic.oup.com/biomet/article-pdf/26/4/404/823407/26-4-404.pdf].