UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Towards Large-scale Quantum Computing

Permalink
https://escholarship.org/uc/item/8x004643

Author
WU, ANBANG

Publication Date
2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Santa Barbara

Towards Large-scale Quantum Computing

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

Doctor of Philosophy
in
Computer Science

by

Anbang Wu

Committee in charge:

Professor Yufei Ding, Chair
Professor Tevfik Bultan
Professor Timothy Sherwood
Professor Jonathan Balkind

March 2024

The Dissertation of Anbang Wu is approved.

Professor Tevfik Bultan

Professor Timothy Sherwood

Professor Jonathan Balkind

Professor Yufei Ding, Committee Chair

March 2024

Towards Large-scale Quantum Computing

Copyright © 2024

by

Anbang Wu

I dedicate this to my parents, whose unwavering love lifted me above the challenges of life and work

Acknowledgements

I have received many help throughout my journey towards completing this dissertation. I would like to express my deepest gratitude to my supervisor, Yufei Ding, for her invaluable guidance, support and encouragement. She is always there when I want a talk and discussion and has dedicated considerable time and effort to help cultivate my skills in research, writing and presentation. I really admire her research philosophy which has motivated me to approach problems with depth, express them with clarity, and solve them with elegance. She has taught me an very important lesson: it is never too late to pursue your passions. This make me determined to continue my research journey.

I am also grateful to my committee members, Professor Tevfik Bultan, Professor Timothy Sherwood, and Professor Jonathan Balkind. Their constructive feedback and insightful advice have immensely contributed to the refinement and completion of this dissertation.

I would also express my sincere appreciation to my collaborators, Doctor Ang Li, Doctor Gian Giacomo Guerreschi, Doctor Alireza Shabani, Doctor Andrew W. Cross, Doctor Yunong Shi, Professor Gushu Li, Professor Yuan Xie and Professor Xinyu Wang, for offering valuable suggestions and discussions for my research projects in the Ph.D. career.

I will never forget my lab mates and friends at UC Santa Barbara, whose names would be too numerous to list here, for fostering a friendly and engaging atmosphere for studying and working.

Last but not least, I want to thank my family, whose unwavering love lifted me above the challenges of life and work. Words alone just cannot convey my love and gratitude for them.

Curriculum Vitæ

Anbang Wu

Education

2024	Ph.D. in Computer Science, University of California, Santa Barbara.
2023	M.S. in Computer Science, University of California, Santa Barbara.
2020	Master in Computer Science and Technology, Zhejiang University.
2017	Bachelor in Computer Science and Technology, Zhejiang University.

Publications

[1] Anbang Wu, Keyi Yin, Andrew W Cross, Ang Li, and Yufei Ding. Enabling full-stack quantum computing with changeable error-corrected qubits. ArXiv: 2305.07072, 2023.
[2] Hezi Zhang, Keyi Yin, Anbang Wu, Hassan Shapourian, Alireza Shabani, and Yufei Ding. Mech: Multi-entry communication highway for quantum computation on chiplets. ASPLOS 2024.
[3] Anbang Wu, Yufei Ding, and Ang Li. Qucomm: Optimizing collective communication for distributed quantum computing. MICRO 2023.
[4] Hezi Zhang, Anbang Wu, Yuke Wang, Gushu Li, Hassan Shapourian, Alireza Shabani, and Yufei Ding. OneQ: A Compilation Framework for Photonic One-Way Quantum Computation. ISCA 2023.
[5] Anbang Wu, Hezi Zhang, Gushu Li, Alireza Shabani, Yuan Xie, and Yufei Ding. Autocomm: A framework for enabling efficient communication in distributed quantum programs. MICRO 2022.
[6] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, and Yuan Xie. A synthesis framework for stitching surface code with superconducting quantum devices. ISCA 2022.
[7] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. Paulihedral: A generalized block-wise compiler optimization framework for quantum simulation kernels. ASOLOS 2022.
[8] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yuan Xie, and Yufei Ding. Qecv: Quantum error correction verification. ArXiv: 2111.13728, 2021.
[9] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. On the co-design of quantum software and hardware. NanoComm 2021.
[10] Anbang Wu, Gushu Li, Yuke Wang, Boyuan Feng, Yufei Ding, and Yuan Xie. Towards efficient ansatz architecture for variational quantum algorithms. ArXiv: 2111.13730, 2021.
[11] Anbang Wu, Gushu Li, Yufei Ding, and Yuan Xie. Mitigating noise-induced gradient vanishing in variational quantum algorithm training. ArXiv: 2111.13209, 2021.

Please Note: Permissions are obtained for reusing contents from the above papers in the dissertation.

Abstract

Towards Large-scale Quantum Computing

by

Anbang Wu

With the rapid advancements in quantum hardware, we stand on the brink of the large-scale quantum computing (LSQC) era, poised to harness the power of thousands of even millions of noisy qubits. This heralds a transformative period where the computational prowess of LSQC holds promises for addressing practical scientific challenges, notably in the realms of molecule simulation and drug discovery. However, despite this exciting progress, a noticeable gap persists between the current Noisy Intermediate-Scale Quantum (NISQ)-era ecosystem and the full potential of LSQC. On the one hand, the scale of LSQC makes manual hardware/software optimizations untenable. On the other hand, the noisy nature of quantum hardware, accompanying the instruction scale of large quantum applications, will surely destroy the quantum computing outcome. These aspects of LSQC induces a series of challenges and requirements not well optimized and supported by NISQ computing stack: automation, scalability, and robustness.

The dissertation aims to fill this gap and synthesize a full-stack design for the LSQC computing stack. The proposed framework presents systematic optimizations for quantum software and quantum architecture. The vision is that significant advances in LSQC require full-stack infrastructure that could not only vertically integrate the advances from both the higher problem level and the lower hardware level. Concerning rapidly evolving problem size and hardware magnitude, the value of the proposed design will only increase. It provides a clearer road-map for building the LSQC in the near future. The evaluation demonstrates the superiority of the proposed framework.

Contents

Curriculum Vitae **vi**

Abstract **vii**

1 Introduction **1**
1.1 Motivation 1
1.2 Overview and Outline 2

2 Background **8**
2.1 Quantum Computer Basics 8
2.2 Distributed Quantum Computing 9
2.3 Quantum Fault Tolerance 11

3 Optimizing Burst Communication for Distributed Quantum Computing **19**
3.1 Introduction 19
3.2 Problem and Motivation 23
3.3 Burst Communication Framework 30
3.4 Evaluation 41

4 Optimizing Collective Communication for Distributed Quantum Computing **51**
4.1 Introduction 51
4.2 Problem and Motivation 55
4.3 Collective Communication System Design 59
4.4 Evaluation 69

5 Synthesizing an Error-Corrected Qubit **84**
5.1 Introduction 84
5.2 Problem Formulation 88
5.3 Synthesis Algorithm Design 90
5.4 Evaluation 100

6 Synthesizing a Reliable Computing Platform **109**
6.1 Introduction 109
6.2 Design Considerations 113
6.3 QEC-based Computing Platform Design 118
6.4 Evaluation 125

7 Synthesizing Verified Quantum Operations **137**
7.1 Introduction 137
7.2 Motivating Example 142
7.3 Programming Language Designs for QEC 149
7.4 Weakest Precondition Computation 158
7.5 Evaluation 163

8 Related Work **171**
8.1 Optimization of Distributed Quantum Computing 171
8.2 QEC Code Synthesis 172
8.3 Verification of Quantum Programs 174

9 Conclusion and Discussion **177**

Bibliography **179**

Chapter 1

Introduction

1.1 Motivation

With the rapid advancements in quantum hardware, we stand on the brink of the large-scale quantum computing (LSQC) era, which guarantees sufficient quantum resources for demonstrating practical quantum advantage on well-known quantum applications, like Shor's algorithm [1], chemistry simulation [2], and quantum machine learning [3]. Those applications often require thousands of even millions of qubits, which is beyond the reach of NISQ (Near-term Intermediate-Scale Quantum) era, which only considers computation with dozens of qubits. The system and architecture optimizations proposed in the NISQ era are far from optimal for LSQC due to two aspects.

Firstly, NISQ quantum compilers lack high-level understanding of program patterns in practical quantum applications, making their optimizations general but far from optimal. While seemingly causing only minor effect on small-scale quantum circuits, the inefficiency can destroy LSQC. As the quantum hardware size grows, the instruction-level overhead induced by communication (i.e., multi-qubit gate) between distant quantum data significantly increase, just as in classical data-center computing where the com-

munication bottlenecks the overall computing performance. Secondly, large quantum applications will consist of orders of magnitude more instructions than NISQ programs, amplifying the noise effect of quantum instructions. This fact motivates noise-robust system/architecture designs. For example, guaranteeing 10^{-9} operation error rate over quantum hardware of 10^{-3} physical error rate is needed for a reliable execution of Shor's algorithm, which consists of millions of quantum operations. Unfortunately, The NISQ computing stack lacks support of noise resilience for executed quantum circuits.

To tackle these challenges, this dissertation proposes comprehensive and systematic hardware-software optimizations for LSQC. The proposed framework for LSQC not only implements scalable compiler optimizations for large-scale quantum circuits to enhance communication efficiency between qubits across multiple quantum processors but also facilitates automated fault tolerance over quantum hardware to shield quantum data from quantum noise. Scalability and automation are crucial for LSQC, where manual optimizations of quantum applications are impractical due to the extensive number of computing components and the resulting computational complexity.

1.2 Overview and Outline

The goal of the dissertation is to design a large-scale and fault-tolerant quantum computing (FTQC) framework (system + architecture), which is essential and indispensable for demonstrating practical quantum advantage. To achieve this goal, the dissertation mainly focuses on the following two lines of work: a) enhancing large-scale communication efficiency, and b) enabling fault tolerance on quantum hardware.

1.2.1 Enhancing Large-Scale Communication Efficiency

The availability of qubits on a quantum chip is frequently restricted due to fabrication limitations [4]. This challenge escalates for LSQC which are featured large-scale quantum circuits. A single quantum device may only accommodate a few dozen logical qubits. In this scenario, distributed quantum computing (DQC) emerges as a promising approach for scaling up [5, 6, 7, 8]. DQC integrates the computing resources of multiple quantum processors with inter-node quantum communication. However, the efficiency of quantum communication in DQC is not well studied previously, leading to great resource and time overhead. The dissertation significantly reduces the communication overhead of DQC by integrating high-level program patterns into compiler-level optimizations, enabling DQC in the near term.

Chapter 3: Optimizing Burst Communication in DQC

Quantum communication and thus inter-node operations between different quantum computers are often far more error-prone and time-consuming than intra-node operations [7], demanding extra compiler optimizations. Previous compilers for DQC only focus on reducing the communication cost of each nonlocal CX or SWAP, regardless of quantum program contexts. The dissertation points out that enhanced communication optimization can be achieved by inspecting quantum programs' data footprints in the DQC network.

The dissertation proposes AutoComm [9], a compiler which identifies the burst communication patterns widely existing in distributed quantum programs and reduces the overall communication overhead by devising specific communication optimizations. The insight of AutoComm is that a group of nonlocal CX between two nodes often involve the same nonlocal quantum data. This implies that a single transmission of the required

quantum data between the two nodes is sufficient for these nonlocal CX, rather than repeatedly sending the data for each nonlocal CX. This efficient data transmission is named **burst communication**. AutoComm demonstrates the abundance of burst communication in distributed quantum programs and provides optimized data transmission schemes for burst communication of different patterns. Compared to previous works, AutoComm increases the communication throughput by up to 18x, and reduces the communication request and latency of distributed programs by 72.9% and 69.2%, respectively.

Chapter 4: Optimizing Collective Communication in DQC

The dissertation further proposes QuComm [10], a framework that extends AutoComm and proposes an efficient communication system for multi-node communication over restrained quantum hardware. QuComm unveils more optimization opportunities by examining the data flow between multiple nodes, i.e., **collective communication**. In theory, more nodes (extremely, all nodes) and more nonlocal gates/data are involved in one collective communication, more communication reduction can be achieved. However, a large collective communication requires many concurrent communication channels on related nodes. It is often not feasible to execute a too large collective communication due to the limitation of communication resources. To optimize collective communication, QuComm comprises of two key contributions. Firstly, QuComm establishes the communication buffer, which can temporarily store the pre-established communication channels to enable a larger collective communication. Secondly, QuComm proposes a heuristic to determine the largest collective communication to be accommodated within the communication buffer and efficiently arrange the data flow according to high-level communication patterns extracted from collective communication blocks. Compared to state-of-the-art (i.e., AutoComm), this communication system further reduces quantum communication requests by 54.9%.

1.2.2 Enabling Fault Tolerance on Quantum Hardware

Quantum computing is error-prone (e.g., the control infidelity can be up to 1%) due to physical and engineering limitations, requiring quantum error correction (QEC) to correct errors in the computing results [1]. Most existing works focus on the theoretical performance of the QEC code (QECC) and lacks systematic studies about efficiently synthesizing QECC on quantum hardware, let alone a comprehensive and verified computing stack based on the synthesized QECC. The dissertation efficiently tackled these gaps by orchestrating architecture, system, and programming language designs.

Chapter 5: Synthesizing an Error-Corrected Qubit

The qubit connectivity assumption of most QECC does not match the constrained topology of real quantum devices. For example, the surface code necessitates square-grid connectivity, while mainstream quantum devices equip with sparse connectivity, like IBM heavy hexagon devices. It is indispensable to overcome this architectural gap to enforce QECC on hardware. Motivated by the idea from classical EDA tools, the dissertation abstract the QECC synthesis problem as the layout problem in computer architecture but with constraints imposed by QECC and quantum mechanics. The dissertation points out a middleware that separates QEC design and hardware fabrication but able to stitch them is the most viable solution to mitigate the gap and accommodate any change in QECC or quantum device.

Based on this insight, the dissertation proposes surf-stitch [11], the first automation tool that transforms and fits the surface code to arbitrary quantum devices. Surf-stitch automatically extracts features from the surface code and mainstream quantum architectures and exploits the extracted features to implement a transformed yet functional-equivalent variant of the surface code on each quantum hardware. The synthesized QECC

can be regarded as a new 'layouted' QECC tailored for the deploying hardware. The evaluation of the synthesized QECC demonstrates that they can achieve equivalent or even better error correction capability, compared to manually designed QECCs for specific quantum devices.

Chapter 6: Synthesizing a Reliable Computing Platform

After synthesising a single 'logical' qubit that is protected by QECC, the next step is to explore efficient QECC-based computing platform which involves designs over multiple logical qubits. The insight is that it is important to orchestrate both architectural and compiler optimizations of QECC. The dissertation further proposes FLEX [12], which provides the first full-stack framework for FTQC based on code switching. FLEX identifies the architecture and compiler challenges induced by operations in quantum programs and crystallizes the large design space motivated by these challenges. FLEX tackles the architecture design by exploiting profiling data of QECC and crafts compiler optimizations for the expensive code switching operation with program context information. FLEX further employs the qubit communication pattern to guide the architecture-compiler co-designs over each program for even better efficiency. The evaluation shows the great advantage of FLEX over existing QECC designs, indicating potential FTQC in the near term.

Chapter 7: Synthesizing Verified Quantum Operations

The next key enabler of FTQC is trustworthy and verified quantum operations over logical qubits. The implementation of QECC-based quantum operations is complicated, requiring intensive interaction between quantum hardware and classical controller. Even experts may make mistakes due to the complexity of QECC. It is thus important to synthesize verified quantum operations. However, formulating and verifying QECC imple-

mentations are challenging as each QECC operation involves a series of physical quantum gates over plenty of physical qubits, causing exponential overhead for qubit-wise reasoning. While it is possible to treat QECC implementations as general quantum programs and adopt existing works to verify them, the scalability or accuracy of such verification is not satisfactory.

This dissertation indicates that the structure information of QECC can be the key to improve the scalability of the verification of QECC operations while guarantee the correctness. Based on the insight, this dissertation proposes a scalable automated verification framework VERITA [13] for stabilizer-based QECC. VERITA prioritizes the structural information (i.e., stabilizers) of QECC, shaping it as the foundational element of the proposed QECC-based programming language and assertions. With these high-level information, VERITA derives a sound quantum Hoare logic, containing scalable inference rules that avoid exponential qubit-wise reasoning overhead in most cases. VERITA demonstrates exponential time and space savings for verifying QECC operations. It can verify operations of surface codes with up to 10081 qubits, suggesting the effectiveness of QECC's structure information for the verification of QECC-based quantum operations.

Chapter 2

Background

This chapter is devoted to providing background knowledge for the following chapters that addresses specific large-scale quantum computing system design problems. The material [1] also provides sufficient and excellent supplemental materials for the background chapter and the audience may refer to it for more details.

2.1 Quantum Computer Basics

Similar to the bit which is information unit in classical computing, quantum computing employs the qubit as the basic information unit. The state/information of a qubit is represented as a unit vector $a\left|0\right\rangle + b\left|1\right\rangle$ in the Hilbert space spanned by the computational basis states $\left|0\right\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\left|1\right\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, which are built upon quantum effects of quantum computing materials like the superconductor, trapped ion, neutral atom, and photon. Furthermore, the state of n-qubits are represented by unit vectors in the 2^n-dimension Hilbert space, of which the computational basis will be $\{\prod_{x_i \in \{0,1\}, i=1,\cdots,n} \left|x_i\right\rangle\}$.

Likewise, quantum computing relies on quantum (logic) gates to manipulate the state of qubits. A commonly-used and universal gate basis for quantum computing is demon-

strated in the following equation:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, RX(\theta) = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix}, CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.1}$$

For a general n-qubit system, the H or the $RX(\theta)$ gate is local to alter the state of a single qubit while the CX gate is used to manipulate the state of two specific qubits, which are its operands. An arbitrary n-qubit quantum circuit that manipulates a general n-qubit system can be composed of those single-qubit and two-qubit quantum gates.

2.2 Distributed Quantum Computing

This section serves as the foundation of Chapter 3 and 4.

Quantum Communication. Like in classical distributed computing, remote/inter-node quantum communication is the bedrock of distributed quantum computing (DQC) but is also the bottleneck of DQC. Different from classical distributed computing, quantum data cannot be easily shared across quantum nodes due to the quantum no-cloning theorem [1]. The workaround is to exploit inter-node quantum entanglement. If two qubits are in the entangled quantum state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, that is to say they form an *EPR pair* [1]. The two qubits of an EPR pair can be distributed to different quantum nodes, formulating a remote EPR pair [14]. The remote EPR pair is the most widely-used quantum communication resource, providing the necessary quantum entanglement for transferring quantum data between nodes. Actually, based on EPR pairs, two communication protocols emerge, named *Cat-Comm* and *TP-Comm* respectively. Figure 2.1 illustrates how to use these two schemes to implement one *remote CX gate*, with the control qubit q_0 residing in quantum node A and the target qubit q_0' in node B. Qubits in Figure 2.1 fall into two categories. The first category of qubits is used to store program

information and is called *data qubits*, e.g., q_0 and q'_0. The second category of qubits, called *communication qubits*, is used to hold remote EPR pairs, e.g. q_{c0} and q'_{c0}.

Cat-Comm and TP-Comm. As shown in Figure 2.1(a), Cat-Comm utilizes cat-entangler to transfer the state of the control qubit q_0 to node B, execute the target CX gate, and then use cat-disentangler to transfer the state back to node A. TP-Comm in Figure 2.1(b) employs quantum teleportation [1] to transfer the state of q_0 to node B, and then execute the target CX gate. Note that in Figure 2.1(b), another teleportation is included to move the state teleported to q'_{c0} back to q_0. Essentially, the second teleportation is used to handle the side effect of TP-Comm: the communication qubit q'_{c0} will be occupied by the teleported state of q_0 and thus cannot be used to establish new EPR pairs. The second teleportation is needed to set q'_{c0} free. For the second teleportation, besides moving the teleported state of q_0 back to its original location as in Figure 2.1(b), the teleported state can also be moved to any other location as long as the occupation on the current communication qubit q'_{c0} is released. Overall, two EPR pairs are required to implement one remote CX gate by TP-Comm, with one of them handling the side effect. To avoid ambiguity, here the saying of *one invocation of TP-Comm* actually refers to one invocation of quantum teleportation. That's to say, one invocation of TP-Comm consumes one EPR pair, just like one call of Cat-Comm. Thus, to implement one remote CX, two invocations of TP-Comm are needed.

Figure 2.1 only shows how to implement one remote CX gate. To implement a complex remote interaction between quantum nodes, one simple strategy is to first decompose the remote interaction into several remote CX gates and implement each remote CX as in Figure 2.1. However, this strategy may incur heavy communication costs. Eisert et al. observe that optimized implementations exist for specific inter-node interactions, as shown in Figure 2.2. The proposed communication optimization in this dissertation will cleverly take advantage of the implementations in Figure 2.2 based on the observa-

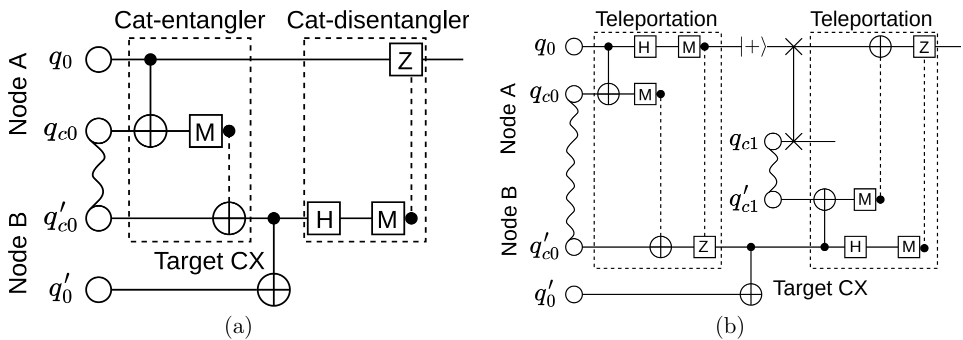


Figure 2.1: The implementation of one remote CX. (a) The Cat-Comm version. (b) The TP-Comm version. Wavy lines and dashed lines denote EPR pairs and classical communication bits respectively. M denotes measurement.

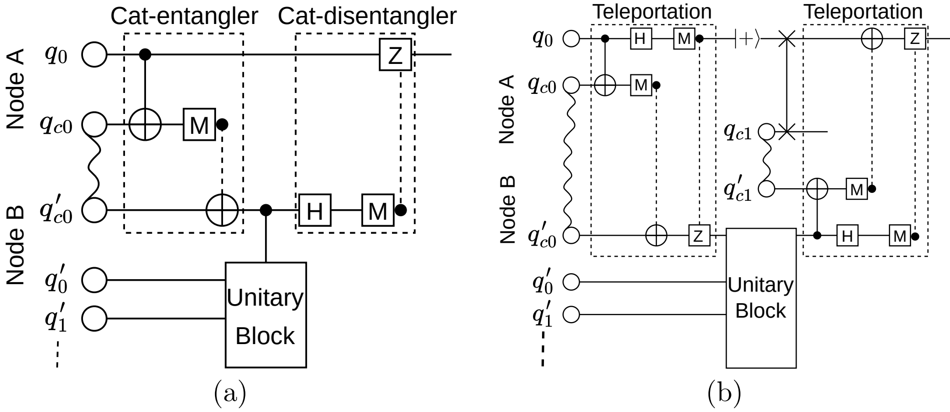


Figure 2.2: The optimized implementation of complex remote interactions. (a) *Controlled-unitary* block by one call of Cat-Comm. (b) *Unitary* block by two calls of TP-Comm.

tion of the burst and collective communication, to optimize the communication cost of distributed quantum programs, as demonstrated in Chapter 3 and 4.

2.3 Quantum Fault Tolerance

This section serves as the foundation of Chapter 5, 6 and 7, which aims to design an efficient system to enable fault-tolerant quantum computing.

2.3.1 QEC Code basics

Quantum computation is fragile without error correction. Information in qubits can be easily distorted by the decoherence error [1]. The imprecise quantum operation and erroneous quantum measurement further worsen the situation [1]. To ensure fault-tolerant quantum computation, various QEC codes [15, 16, 17, 18, 19, 20] are proposed. In these QEC codes, the surface code is among the most popular ones due to its excellent error correction capability [20]. The rest of this section will use surface code as the example to demonstrate basic components and concepts of QEC codes.

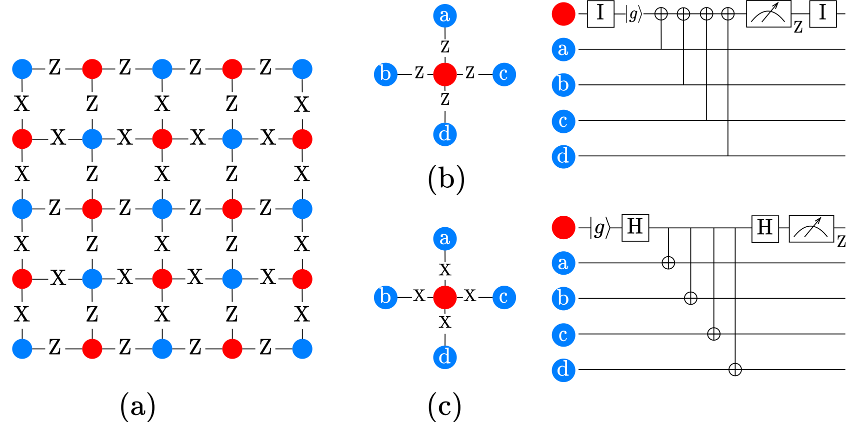


Figure 2.3: Common components of the surface code. (a) The surface code lattice with data qubits (blue dots) and syndrome qubits (red dots). (b) Z-type syndrome extraction and its circuit. (c) X-type syndrome extraction and its circuit.

Data and syndrome qubits. The surface code encodes a logical qubit in a 2D lattice of physical qubits, as shown in Figure 2.3(a). The physical qubits in the code lattice can be divided into two types: data qubits and syndrome qubits, denoted as blue and red dots, respectively in Figure 2.3(a). The encoded logical information is stored in data qubits. Error information on data qubits can be extracted by measuring the syndrome qubit (a.k.a measurement qubit). Each syndrome qubit is coupled with its (up to) four neighboring data qubits, using the syndrome extraction circuit (a.k.a measurement circuit) shown in Figure 2.3(b)(c) to gather the error information on data

qubits.

Pauli operator and stabilizer. In surface codes, the relationship between a syndrome qubit and its neighboring data qubits is represented by the product of Pauli operators (a.k.a Pauli string [1]), as shown in Figure 2.3(a) where Pauli operators are labeled on the edges between data qubits and syndrome qubits. For each syndrome qubit, the Pauli string on its edges can be in one of two possible patterns. The first one (Z-type) is shown in Figure 2.3(b). The connections between the center syndrome qubit and the four data qubits are all labeled by the operator Z, and together they are represented by the Pauli string $Z_aZ_bZ_cZ_d$. The second one (X-type) as shown in Figure 2.3(c) is similar, except that all connections are labeled by the operator X, together represented by the Pauli string $X_aX_bX_cX_d$.

For these two different patterns, corresponding error detection (or syndrome extraction) circuits are devised to detect errors on data qubits (shown on the right side of Figure 2.3(b)(c)). Syndrome extraction circuits in Figure 2.3(b)(c) project the state of data qubits $\{a, b, c, d\}$ onto the eigenstates of corresponding Pauli strings, which are also referenced by *stabilizers* [21] in the context of QEC. Syndrome extraction is thus known as the *stabilizer measurement* [22]. Without ambiguity, the stabilizer notation will be used to represent the syndrome extraction circuit, and the stabilizer $Z_aZ_bZ_cZ_d$ ($X_aX_bX_cX_d$) will be abbreviated into Z_{abcd} (X_{abcd}).

Error detection cycle. Surface codes can detect Pauli X- and Z-errors on data qubits with Z- and X-type stabilizer measurement circuits, respectively. Errors on a data qubit can affect the measurement results of stabilizers associated with it. In an *error detection cycle*, the surface code would run all stabilizer measurements once and collect the measurement results. With these results, a surface code error correction protocol can infer what errors have occurred in the code lattice and apply corrections accordingly. For more details, please refer to [20].

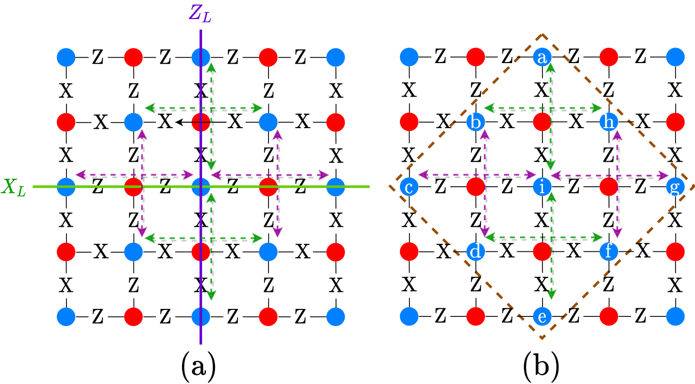


Figure 2.4: Code distance and the compact lattice. (a) Logical operations of the distance-3 surface code. (b) Inside the rotated rectangle (dashed brown line) is a compact surface code lattice with the same code distance as in (a).

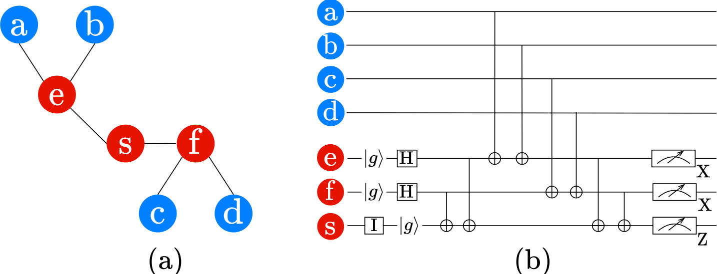


Figure 2.5: Z-type stabilizer measurement circuit synthesis. (a) The connected graph of data qubits (blue) and ancillary qubits (red). Qubit s is the syndrome qubit. (b) The synthesized stabilizer measurement circuit that is executable on the connected graph in (a).

Code distance. The error correction capability of the surface code is related to the code distance [23, 17], which is defined as the minimum number of physical qubits that support the logical X or Z operation on the encoded logical qubit (denoted by X_L or Z_L in Figure 2.4). Usually, surface codes with larger code distances can correct more complex errors, but their implementation overhead is also higher. Figure 2.4(a) shows the logical operations in a distance-3 surface code. Figure 2.4(b) indicates that a more compact surface code lattice can be obtained without changing the code distance.

2.3.2 Logical Operations upon QEC Codes

Transverse logical gate. To enable FTQC over QEC codes, logical gates that fault-tolerantly manipulate logical qubits are required. Many logical gates can be easily implemented by applying data-qubit-wise physical gates. Such logical gates are called to be transverse. For instance, on a logical qubit $\bar{q}$ of the Steane code (in short, a Steane logical qubit) whose data qubits are $q_1, q_2, \cdots, q_7$, the logical H gate (H_L) is transverse and is defined by seven physical H gates: $H_L\bar{q} = \otimes_{i=1}^{7} Hq_i$. Logical CX ($CX_L$) is also transverse and is defined by $CX_L\bar{q}_0\bar{q}_1 = \otimes_{i=1}^{7} CXq_{0i}q_{1i}$. Logical gates comprise another critical part of the QEC architecture design, especially for the logical CX where is control and target data qubits are not neighboring to each other.

Non-transverse logical gate. Unfortunately, not all logical gates of a QEC code admit a transverse implementation [24]. For instance, the logical T gate (T_L) is transverse in the 15-qubit Reed-Muller (RM) code (defined as $T_L\bar{q} = \otimes_{i=1}^{15} T^\dagger q_i$) but is non-transverse in the Steane code. Likewise, H_L is transverse in the Steane code but non-transverse in the RM code. To achieve universal FTQC with the Clifford+T gate basis, various schemes (e.g., magic distillation [25] and code switching [26]) are proposed to provide robust implementations of the 'difficult-to-implement' non-transverse logical gates. Among these schemes, code switching stands out for its potentially smaller resource-time overhead [27].

Code switching. Specifically, the core idea of code switching is to encode the logical qubit in different QEC codes along the time dimension, e.g., switching between the Steane and the RM code, which is widely studied in existing works [27, 26]. To implement T_L on a Steane logical qubit, the logical qubit can be switched into the RM code. After T_L is transversely implemented, one more code switching is needed to transform the logical qubit back into the Steane code. For RM code $\{q_{r1}, \cdots, q_{r15}\}$, $\{q_{r1} \cdots, q_{r7}\}$ and $\{q_{r8} \cdots, q_{r14}\}$ separately form two Steane codes. The code switching from Steane code to

RM code contains one logical CX between Steane code $\{q_{r1} \cdots, q_{r7}\}$ and $\{q_{r8} \cdots, q_{r14}\}$ and then three physical CX gates $(CX q_{c15}\, q_{c5}, CX q_{c15}\, q_{c6}, CX q_{c15}\, q_{c7})$, with each logical/physical CX followed by one Steane error correction round. The following RM error correction round will then finish the code switching process (please see [27, 26] for more details). The code switching from the RM to the Steane code is just the reverse of Steane code to RM code. The correctness and fault tolerance of code switching is guaranteed [27, 26].

The implementation of code switching is another critical design aspect of the QEC architecture. Compiler optimizations can be further incorporated to reduce the utilization of the time-consuming and error-prone code switching protocol.

2.3.3 Enforcing QEC on Hardware

In this section, necessary underlying toolkit for implementing QEC operations on hardware will be introduced. In many quantum devices (e.g., superconducting [28] and neutral atom hardware [29]), the connectivity between physical logical qubits is limited. In such a case, the physical CX gate in stabilizer measurement circuits, logical CX, and code switching may be on non-neighboring physical qubits.

Stabilizer measurement implementation. There may be a structural gap between the measurement circuit and the hardware connectivity. As an example, to map a stabilizer measurement circuit that requires a four-degree qubit to the sparsely-connected superconducting device that lacks such high-degree qubits, various methods have been proposed, such as the degree-deduction technique [30], and the flag-bridge circuit [31, 32, 33, 34]. Once data qubits and the required ancillary qubits (including the syndrome qubit) for the stabilizer measurement are specified, those methods can generate corresponding measurement circuits executable on superconducting devices. Figure 2.5

shows the generated flag-bridge circuit for the stabilizer Z_{abcd} with a tree of ancillary qubits $\{e, s, f\}$. These ancillary qubits are also called *bridge qubits*, and the tree they form is called the *bridge tree*. Here the qubit s is set as the tree root, which acts as the syndrome qubit for Z_{abcd}, collecting error information from data qubits $\{a, b, c, d\}$. The construction of the flag-bridge circuit consists of five components:

1) *Initialization:* bridge qubits except the tree root s are initialized to $|+\rangle$ while s is initialized to $|0\rangle$.

2) *Encoding circuit:* Starting from $k = 1$ (i.e., from the root s), for each node at the k-th level of the bridge tree, one CNOT gate is added from this node to its parent node in the encoding circuit.

3) *Coupling circuit for data qubits:* The data qubits are coupled in a zigzag way with leaf bridge qubits instead of the tree root to respect the device connectivity limitation.

4) *Decoding circuit:* The decoding circuit is the mirror of the encoding circuit. Those two circuits together ensure the fault tolerance of the stabilizer measurement.

5) *Measurements:* The tree root is measured on the Z basis while other ancillary qubits are measured on the X basis. The measurement results on the X basis can be used to detect Pauli Z errors on ancillary qubits.

Other stabilizer measurement circuits can be constructed in a similar way. For more details, please refer to Lao et al. [31]. Note that methods discussed in this section only solve the low-level circuit generation problem of one stabilizer measurement, far from tackling the overall surface code implementation.

Implementation of logical CX and code switching. To perform a physical CX between data qubits that are not physically coupled, the GHZ state (see Figure 2.6) rather than SWAP gates will be used for low cost, as practiced widely by existing works [35, 36], where the GHZ state is used to build the fault-tolerant logical CX. In Figure 2.6, a GHZ state can be built using $\{q_i, \cdots, q_{i+k}\}$ and the remote CX between q_1 and q_0 is

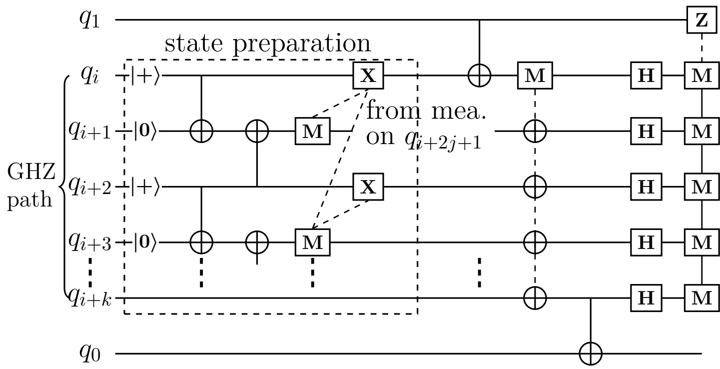


Figure 2.6: Remote CX between q_1 and q_0 using the GHZ state. The state preparation part of the circuit will generate a GHZ state over $q_i, q_{i+2}, q_{i+4}, \cdots, q_{i+k}$.

implemented by using the prepared GHZ state. Qubits in the state preparation part of Figure 2.6 (i.e., $q_i, \cdots, q_{i+k}$) form a **GHZ path** to connect two data qubits.

In the proposed architecture designs for QEC-based quantum computing, the flag-bridge circuit in Figure 2.5 will be used for error detection and the GHZ-state-based method in Figure 2.6 will be used for the CX gate between non-neighboring physical qubits. Using these toolkit is not the contribution of this dissertation. Rather, this dissertation concerns higher-level architectural designs upon these toolkit.

Chapter 3

Optimizing Burst Communication for Distributed Quantum Computing

In this chapter, we will delve into optimizing the burst communication pattern commonly found in distributed quantum programs.

3.1 Introduction

Quantum computing is promising with its great potential of providing significant speedup to many problems, such as large-number factorization with an exponential speedup [37] and unordered database search with a quadratic speedup [38]. A large number of qubits is required in order to solve practical problems with quantum advantage and the qubit count requirement is even higher after taking quantum error correction [1] into consideration. However, it has turned out that extending the number of qubits on a single quantum processor is exceedingly difficult due to various hardware-level challenges such as crosstalk errors [39, 40], qubit addressability [40], fabrication difficulty [41], etc. Those challenges usually increase with the size of quantum hardware and may limit the

number of qubits in a single quantum processor.

Rather than relying on the advancement of a single quantum processor, an alternative way to scale up quantum computing is to explore distributed quantum computing (DQC) [5, 8], which integrates the computing resources of multiple quantum processors. In DQC, remote quantum communication involving qubits in different compute nodes is essential yet far more expensive than the local communication between same-node qubits (e.g., 5-100x time consumption and up to 40x fidelity degradation [42, 43]). There are two major schemes for inter-node communication: one built upon cat-entangler and -disentangler protocols [44], and the other based on quantum teleportation [1]. We refer to the former scheme as *Cat-Comm* and the latter one as *TP-Comm*. Both schemes consume remote EPR pairs [14], which are pre-distributed entangled qubit pairs, as a resource to establish quantum communication. Cat-Comm can implement a remote CX gate [1] with only one EPR pair, but for general two-qubit gates such as SWAP gate [1], Cat-Comm requires up to three EPR pairs [45]. In contrast, TP-Comm can execute any remote two-qubit gate with two EPR pairs [14] and is thus more efficient for the SWAP gate. For a distributed quantum program, more complex remote operations or more quantum information transferred per EPR pair would lead to less communication cost.

The overall compiling flow for DQC is similar to that of single-node quantum programs, except with more emphasis on remote communication overhead. One compiler design proposed by Ferrari et al. [45] adopts a similar compilation strategy to single-node compilers [46, 28, 47, 48, 49], using Cat-Comm for implementing the remote CX. This strategy has a low communication throughput due to the low information of the remote CX gate. The compiler by Baker et al. [42] and another design by Ferrari et al. [45] instead eliminate all remote CX gates by using remote SWAP, which only requires two EPR pairs for implementation but contains the information of three CX gates. Unfortunately, bounded by the information of a single two-qubit gate, these compilers cannot

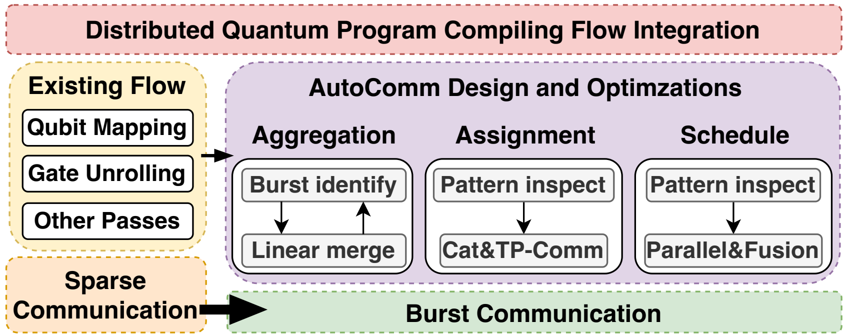


Figure 3.1: AutoComm Overview.

achieve higher throughput of information per EPR pair. Eisert et al. [14] suggest that a higher throughput could be achieved by considering multi-qubit gates. Diadamo et al. [6] compiles distributed VQE by using Cat-Comm to implement controlled-unitary-unitary and controlled-controlled-unitary gates. However, their work can only optimize gates written in the controlled-unitary form, not applicable to the decomposed circuit that consists of only quantum basis gates (e.g., CX+U3 [28]). Besides, their work cannot optimize programs lacking controlled-unitary blocks.

Besides increasing the 'height' (number of qubits) of remote operations, we observe that the throughput of information per EPR pair can also be significantly boosted up by expanding the 'width' (number of gates) of each remote communication. Specifically, we discover that it is possible to implement a group of remote two-qubit gates collectively through one or two EPR pairs. On top of the observation, we propose optimizing the communication overhead of distributed quantum programs based on the *burst communication*, which denotes a group of continuous remote two-qubit gates between one qubit and one node. Burst communication is powerful as it is more information-intensive than a single two-qubit gate and contains but is not limited to controlled-unitary blocks. Burst communication is also flexible for optimization as it does not require specialized circuit representation and is available in decomposed circuits.

To this end, we develop the first burst-communication-centric optimization framework, *AutoComm*. Our framework focuses on the optimization of remote communication and leverages existing compiling flows [42, 45, 46, 28, 47, 48, 49] on other program optimization aspects (e.g., qubit mapping to assign qubits over different DQC nodes), as shown in Figure 3.1. This two-step design not only makes our work extensible but also outstands our significant novel contributions on communication optimization. In existing compiling flows, each CX is implemented independently (i.e., sparse communication); while with *AutoComm*, sparse communication is converted into burst communication and specifically optimized for higher communication throughput.

Overall, our framework consists of three key stages. Firstly, we perform a communication aggregation pass to group remote gates and extract burst communication blocks. Due to the wide existence of burst communication in distributed quantum programs, this pass could generate a large amount of burst communication blocks for the following optimizations. Secondly, we propose a hybrid communication scheme that examines the patterns of each burst communication block and assigns the optimal communication scheme for each block. The insight is that TP-Comm and Cat-Comm are more resource-efficient for different types of burst communication, thus considering only one scheme would incur extra resource consumption. Finally, we adopt an adaptive schedule for burst communication blocks of different patterns to squeeze out the parallelism between them and thus reduce the overall program latency. We observe that it is possible to execute burst communication with shared qubits or nodes in parallel, and we can fuse some burst communication blocks to cut down the communication footprint.

Our contributions are summarized as follows:

- We identify the burst communication feature in DQC and promote its importance in optimizing distributed quantum programs. we further propose the first commu-

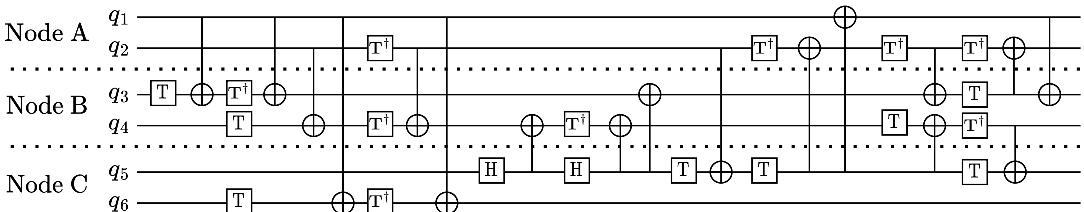


Figure 3.2: Program snippet extracted from quantum arithmetic circuits [51].

nication optimization framework based on it.

- We propose a communication aggregation pass to expose burst communications and design a hybrid communication scheme, using both Cat- and TP-Comm to accommodate different communication patterns.

- We propose an efficient communication scheduling method to optimize the program latency adaptively, squeezing out the parallelism of burst communication.

- Compared to state-of-the-art baselines, AutoComm reduces the EPR pair consumption and the program latency by 72.9% and 69.2% on average, respectively.

3.2 Problem and Motivation

In this section, we first introduce the communication problem in DQC and then identify optimization opportunities by considering burst communication. For the rest of the discussion, we assume that quantum communication can be established between any two quantum nodes, a typical assumption in data-center distributed computing [50]. We also assume that each quantum node has only two communication qubits, which is realistic for near-term DQC [45].

3.2.1 Communication Problem

The example distributed program in Figure 3.2 is modified from quantum arithmetic circuits [51]. This program contains many remote CX gates, e.g., $CX\, q_1, q_3$. Remote CX gates are inevitable in DQC especially when the program's qubit number is substantially larger than that of each quantum node. To execute a distributed program, we need to invoke either Cat-Comm or TP-Comm to implement remote operations, as shown in Figures 2.1 and 2.2. Due to the noisy nature of quantum communication, remote operations are far more error-prone than local quantum gates. The long runtime of quantum communication would also lead to the decoherence of quantum states. As a result, to produce high fidelity outcome, we hope the number of remote communication to be as small as possible, and so is the latency induced.

While one remote CX gate requires at least one remote EPR pair and there is little room for optimizing the communication cost of one remote CX, there is a large optimization space when considering burst communication, which involves a group of remote CX gates. For example in Figure 3.2, we can execute the first two CX gates on q_1, q_3 collectively, with only one EPR pair by using the circuit in Figure 2.2(a). From the perspective of information theory, burst communication is more informative than communication that carries only one remote CX. The overall communication overhead would be considerably lowered if handling all remote CX gates in this burst manner.

Fortunately, as we can see in the next section, burst communication is prevalent in diverse distributed quantum programs.

3.2.2 Burst Communication in DQC

Aside from the arithmetic program shown in Figure 3.2, we also see burst communication in a variety of quantum programs. As examples, we examine the burst com-

munication of the Quantum Fourier Transform (QFT) program [1] and the Quantum Approximate Optimization Algorithm (QAOA) [52] by hand. These two represent different categories of quantum programs: QAOA is one of the most important applications in near-term quantum computing whereas QFT is the building block circuit of quantum algorithms.

We first give a formal definition of burst communication in DQC. We refer to a group of continuous remote two-qubit gates between one qubit and one node as *burst communication*. For two remote two-qubit gates g_1 and g_2, the continuity of these two gates means there are no other remote gates between g_1 and g_2.

To characterize the burst communication of a distributed program $dprog$, for a remote gate g in $dprog$, we define function $\epsilon(g)$ to be the largest burst communication block that contains g. The gate order of $dprog$ may affect the burst communication block found. $\epsilon(g)$ is defined to be the largest over all functional-equivalent gate order of $dprog$. We then define $len(\epsilon(g))$ to be the number of remote CX gates in $\epsilon(g)$ if compiled to the CX+U3 basis [28]. Finally, we are ready to define the *inverse-burst distribution* as follows:

$$P(x) = \frac{|\{g|len(\epsilon(g)) < x\}|}{\#g}. \tag{3.1}$$

A lower $P(x)$ suggests more burst communication. Specifically, for a given x, the lower $P(x)$ is, the larger $1 - P(x)$ is and the more remote CX gates belong to burst communication blocks that each possesses more than x remote CX gates, indicating more burst communication opportunities. On the other hand, for the given probability $P_0 = P(x)$, we hope the corresponding x to be as large as possible since it means there are $1 - P_0$ of remote CX gates belonging to burst communication blocks that each possesses more than x remote CX gates. The distribution $1 - P(x)$ actually provides an ideal upper bound for the burst communication existing in distributed quantum programs and can

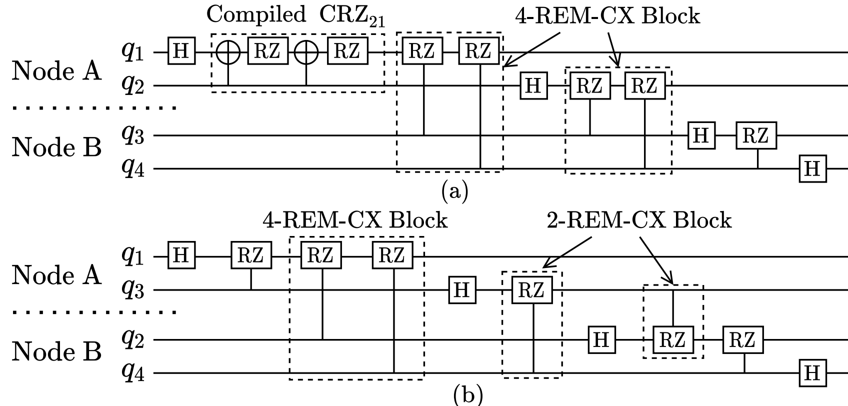


Figure 3.3: (a) QFT program with two nodes and two qubits per node. (b) The layout for the maximal P_4. Parameters of CRZ gates are omitted here for simplicity. For the purpose of demonstration, we do not combine CRZ_{43} and CRZ_{32} to form a 4-REM-CX block. The 4-REM-CX (or 2-REM-CX) block denotes the gate block which contains four (resp. two) remote CX gates when compiled to the CX+U3 basis.

serve as a metric to evaluate the communication efficiency of various distributed quantum algorithms.

We begin by examining the QFT program using the aforementioned definition. We assume the total qubit number is n, the quantum node number is k, and qubits are evenly distributed across all nodes, with $t = \frac{n}{k}$ qubits per node. Figure 3.3 shows the QFT program with $k = 2$ and $t = 2$. For the QFT program, each q_i is controlled by all qubits q_j (through the CRZ gate) that satisfies $j > i$ [1], as shown in Figure 3.3. Then, we have $P(2) = 0$ because each CRZ gate in QFT is compiled into two CX gates, as illustrated in Figure 3.3(a). Now, we consider $P(4)$. For the i-th qubit satisfies $i \leq n-k$, the number of j s.t. $\epsilon(\mathrm{CRZ}_{ji}) < 4$ is at most $\lfloor \frac{i-1}{t-1} \rfloor$ because for one node, if at least two of its qubits have subscripts $> i$, this node would have at least two qubits interacting with qubit i. Since CRZ gates are commutable with each other, we could then form a communication block with at least 4 CX gates. It's easy to see that there are at most $\lfloor \frac{i-1}{t-1} \rfloor$ nodes, each containing no more than one qubit with subscript $> i$. On the other hand, if $i > n - k$, then the i-th qubit is at most controlled by $n - i$ qubits, thus the

number of j s.t. $\epsilon(\text{CRZ}_{ji}) < 4$ is at most $n-i$. Therefore, we have

$$P(4) \leq \frac{\sum_{i=1}^{n-k} \lfloor \frac{i-1}{t-1} \rfloor + \sum_{i=n-k+1}^{n} (n-i)}{\sum_{i=1}^{n} (n-i) - k \sum_{l=1}^{t} (t-l)} = \frac{1}{t}.$$

This indicates there are $1 - P(4) \geq 1 - \frac{1}{t}$ remote gates within burst communication blocks that each possesses more than 4 remote CX gates. Generally, we can prove that $P(x) \leq \frac{x/2-1}{t}$, for $x > 2$. This upper bound is quite promising when t is large and it is actually loose. For Figure 3.3(b) which approximates the upper bound of $P(4)$, there may be $\frac{1}{t}$ of remote CRZ gates, i.e., CRZ_{43} and CRZ_{32} not in a block with 4 remote CX gates at the first glance. But as shown in Figure 3.3(b), we can actually combine CRZ_{43} and CRZ_{32} to form a 4-REM-CX block (the block that contains 4 remote CX gates when compiled to the CX+U3 basis). More 4-REM-CX blocks or n-REX-CX blocks ($n > 4$) can enable more burst communication opportunities. This indicates that the distributed QFT program has more abundant burst communication than the upper bound of $P(x)$ (thus the lower bound of $1 - P(x)$) suggests.

Similarly, for the QAOA program, we assume k nodes and t qubits per node. We also suppose r remote ZZ interactions between any two nodes. Figure 3.4 shows the QAOA program with $k = 2$ and $t = 3$. Likewise, $P(2) = 0$ since each ZZ interaction is compiled into two CX gates, as shown in Figure 3.4(a). For every two nodes, the qubit layout to minimize $len(\epsilon(ZZ))$ for each ZZ interaction is to make every two ZZ interactions have no shared qubits, i.e., not adjacent. However, this layout at most accommodates t ZZ interactions. For $r > t$, considering $m^2 + (t-m) < r \leq (m+1)^2 + (t-m-1)$, there are at most $t-m-1$ ZZ interactions that are not adjacent to any other ZZ interactions. Thus, we have $P(4) = \frac{t-m-1}{r} < \frac{t-m-1}{m^2+(t-m)}$. For example in Figure 3.4(b), we have $1^2 + (t-1) = 3 < r < 2^2 + (t-2) = 5$, thus we predict $P(4) < \frac{t-1-1}{1^2+(t-1)} = \frac{1}{3}$. This bound is correct because in Figure 3.4(b), only $\frac{1}{4}$ of remote ZZ interactions are

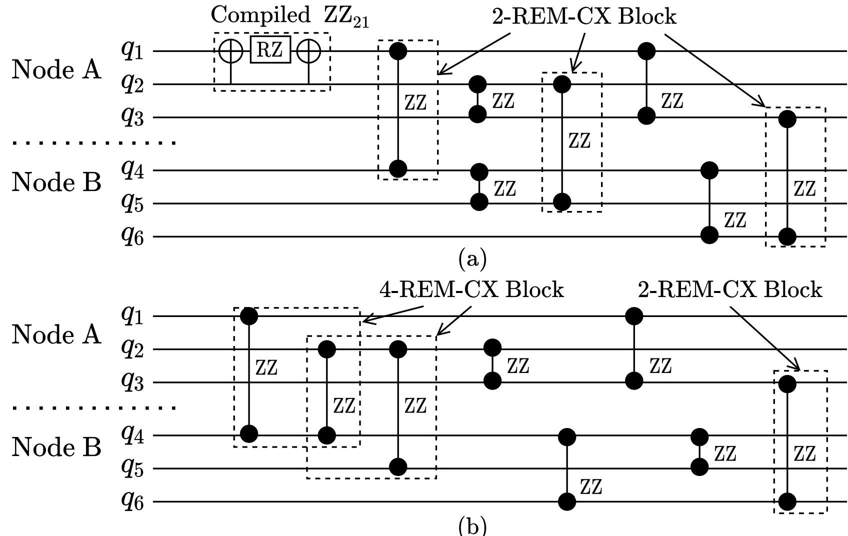


Figure 3.4: QAOA program with two nodes and three qubits per node. Parameters of ZZ interactions are omitted here for simplicity. (a) inter-node communication number $r = 3$. (b) $r = 4$.

not in a 4-REM-CX block. It is easy to see that $P(4)$ would quickly decrease if m becomes large. For the general $P(x)$, a similar conclusion can be reached. Therefore, burst communication is also broadly available in the distributed QAOA program.

We could derive a similar analysis for other distributed quantum programs. Further numerical evidence for the richness of burst communication in various distributed quantum programs is shown in Figure 3.13. The next step is to figure out how to utilize abundant burst communication to optimize the communication cost of distributed quantum programs, as discussed in the next section.

3.2.3 Optimization Opportunities

To exploit burst communication in distributed quantum programs, we need to answer three key questions:

How to unveil burst communication? Burst communication is a type of high-level program information and cannot be easily deduced from the low-level circuit language, especially when remote interactions between multiple nodes are all mixed together. For example in Figure 3.2, gate $CX\ q_2; q_4$ between node A and node B is followed by $CX\ q_1; q_6$, which is the interaction between node A and node C. Such a disorder in distributed quantum programs causes great difficulty in utilizing the benefits of burst communication.

How to select the best communication scheme? Burst communication comes in various forms. While being more efficient for implementing one remote CX gate, Cat-Comm is not always better than TP-Comm for burst communication. For example in Figure 3.2, if we use Cat-Comm to implement the last three remote CX gates between q_3 and node A, three EPR pairs are needed. However, with TP-Comm to teleport q_3 to node A, at most two EPR pairs are needed. Thus, to reduce the communication cost, we should examine the pattern of burst communication and choose the communication scheme wisely.

How to schedule burst communication? Finally, we need to schedule the execution of burst communication blocks. If we arrange all burst communication in a sequential way, the large time overhead would impose non-negligible decoherence errors on quantum states. As a result, we should maximize the parallelism in burst communication to generate high-fidelity output. To achieve this goal, we should first identify the relationships between communication blocks and then reduce the time gaps caused by communication blocks adaptively.

3.3 Burst Communication Framework

In this section, we first give an overview of the AutoComm framework and then introduce each component in detail.

3.3.1 Design overview

We propose the *AutoComm* framework as shown in Figure 3.1. AutoComm focuses on the communication optimization of distributed quantum programs and serves as the back-end of front compiling flows (e.g., mapping qubits to quantum nodes). We would adopt existing technologies for these front compiling stages, as we would see in Section 3.4.

To reduce the communication overhead in distributed quantum programs, AutoComm comes with three stages to utilize burst communication. Firstly, it aggregates remote two-qubit gates by gate commutation. Gate commutation is common in quantum programs [53]. Commutable gates, on the one hand, may be ordered arbitrarily and hide the burst communication. On the other hand, we could also utilize gate commutation to uncover burst communication blocks. In this stage, we first identify potential burst communication and then employ a linear merge step to combine isolated burst communication blocks.

Secondly, it assigns an optimal communication scheme for each burst communication block. We observe that the pattern of burst communication impacts the efficiency of communication schemes. Cat-Comm is less expensive for some patterns, while TP-Comm may be more cost-effective for others. It is thus important to examine the patterns of burst communication and consider both Cat-Comm and TP-Comm for implementing them, rather than only focusing on one scheme.

Thirdly, it performs a block-level schedule of burst communication. It is possible to execute communication blocks with shared nodes or qubits concurrently or shorten the

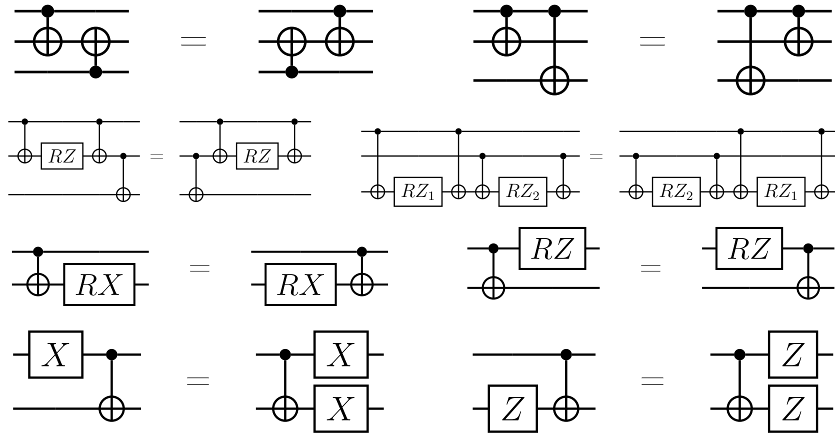


Figure 3.5: Representative gate commutation rules used in AutoComm.

quantum state transfer path across quantum nodes. Combined with these optimizations, a greedy schedule is effective for burst communication.

3.3.2 Communication Aggregation

Burst communication is prevalent in distributed programs, but may not be immediately available due to two factors: CX gates may be scattered across the program, and whether CX gates are remote or not depends on the qubit mapping to quantum nodes. To make our framework able to uncover hidden burst communication regardless of qubit mappings, we need to rewrite the circuit and aggregate remote CX gates.

Figure 3.5 shows a fraction of circuit rewriting rules for remote gate aggregation. The first two rows of Figure 3.5 contain gate commutation rules for two-qubit gates, from simple ones to complex ones. These rules enable flexible two-qubit gate relocation so that burst communication can be automatically exposed and utilized. The remaining two rows of Figure 3.5 are about exchanging single-qubit gates with the CX gate and affect the pattern of the aggregated burst communication (more details in Section 3.3.3). Based on these circuit rewriting rules, we design the following steps to aggregate remote

Algorithm 1: Linear merge procedure

```
   Input: An array of communication blocks blk_list
   Output: Merged communication blocks blk_list_merge
1  blk_list_merge = [];
2  blk = blk_list[0] ;
3  while there are blocks in blk_list not visited do
4      non_commute_gates = [];
5      for blk_next in unvisited blocks of blk_list do
           // Attempt merge blk to blk_next
6          for gate between blk and blk_next do
7              if gate is single-qubit and not commutes with blk then
8                  non_commute_gates.append(gate);
9              if gate is two-qubit then
10                 check if gate is commutable with non_commute_gates and blk;
11                 if not commutable then
12                     if gate is in-node two-qubit then
13                         non_commute_gates.append(gate);
14                     else
15                         break;
16         end
17         blk =merge blk, non_commute_gates and blk_next;
18     end
19     if the above merge failed then
20         Try to merge blk_next to blk similarly;
21         if succeeds then
22             blk =merge blk, non_commute_gates and blk_next;
23         else
24             blk = blk_next;
25 end
26 output the merged blocks and adjust the order of commutable gates;
```

gates.

(a) Identifying potential burst communication As burst communication is defined between one qubit and one node, the first step of communication aggregation is thus to identify the qubit-node pair of potential burst communication. We start with the qubit-node pair associated with most remote gates as it would likely lead to a large burst communication block. For example in Figure 3.2, the chosen qubit-node pair is (q_3, node A) as it is associated with 5 remote CX gates. We then search for consecutive remote CX gates related to this qubit-node pair. In this step, circuit rewriting is not applied

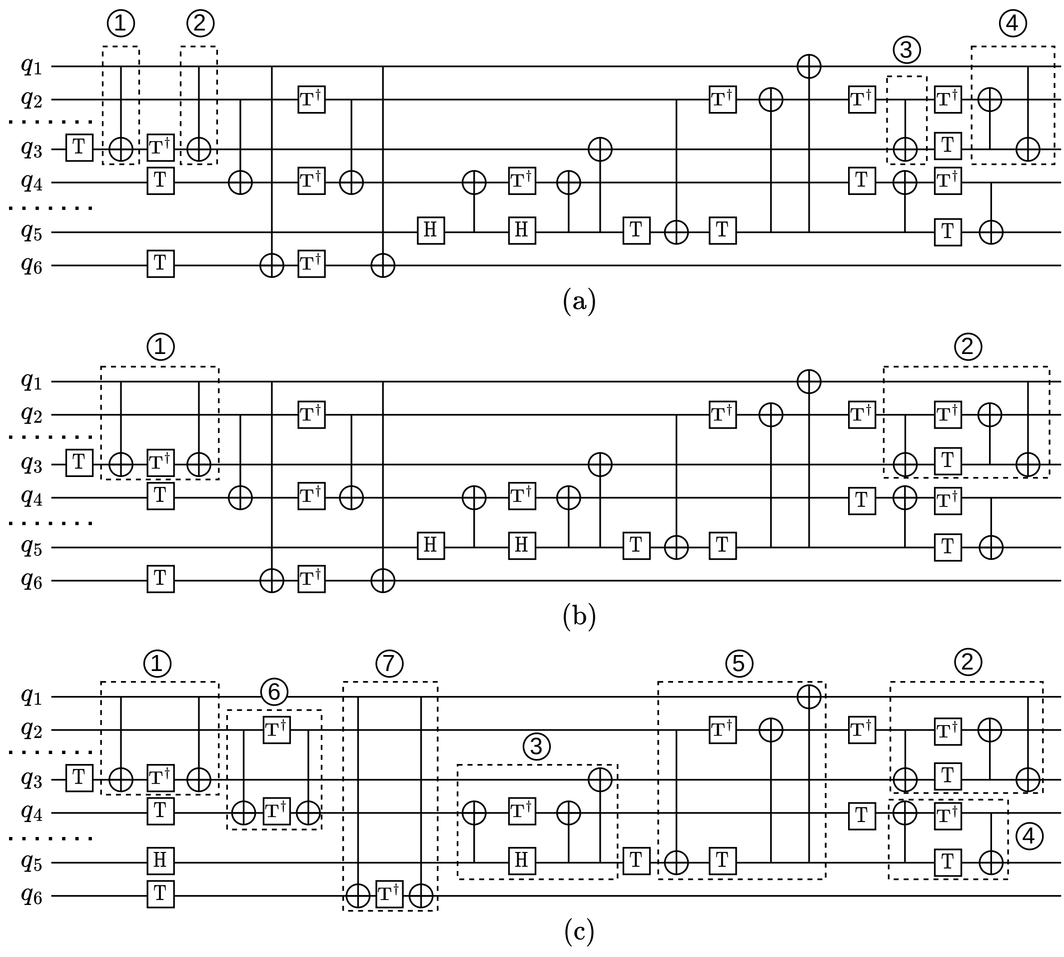


Figure 3.6: Communication aggregation for the example program in Figure 3.2. (a) Identifying potential burst communication. (b) Linear merge. (c) Iterative refinement.

yet and the search would result in many isolated communication blocks. For example in Figure 3.6(a), we obtain four small blocks.

(b) Linear merge The next step is to merge isolated small communication blocks obtained in step (a). As illustrated in Algorithm 1, we merge related communication blocks in a linear and greedy manner. For communication blocks ①, ②, ③, ④ in Figure 3.6(a), we can easily merge block ① and ② since only single-qubit gates exist between those two blocks. We call denote the merged block of ① and ② by *blk_ new*. Unfortunately, we can not merge block *blk_ new* and block ③. On the one hand, $CX\ q_5, q_3$

is not commutable with *blk_new*, so we cannot move *blk_new* close to ③. On the other hand, $CX\, q_5, q_2$ is not commutable with ③, making it impossible to move ③ close to *blk_new*. We then skip *blk_new* and start from block ③ to find other merge opportunities. The linear merge procedure will visit all blocks at least once. Finally, we obtain two larger communication blocks after the linear merge, as shown in Figure 3.6(b).

(c) Iterative refinement Then we repeat steps (a) and (b) for other qubit-node pairs until no more merge opportunities exist. The final result of communication aggregation is shown in Figure 3.6(c). Identified burst communication blocks are ordered by the time being discovered.

3.3.3 Communication Assignment

With burst communication blocks, the next optimization is to find the best way to execute them. We address this problem by first examining the pros and cons of Cat-Comm and TP-Comm, and then assigning the optimal communication scheme based on the pattern analysis of burst communication blocks. Since we assume only two communication qubits in each quantum node, the communication patterns discussed here center on interactions between one qubit and one node. Extending burst communication to the node-to-node situation is promising when communication qubits are plentiful. We leave it for future work.

Cat-Comm vs. TP-Comm: Suppose we have a burst communication block between q_1 in node A and several qubits in node B, with n remote CX gates totally. If the block can be executed by one invocation of Cat-Comm, the savings on EPR pairs would be up to n times, compared to executing each remote CX gate individually. However, Cat-Comm only supports controlled-unitary blocks and needs many (≥ 2) EPR pairs to

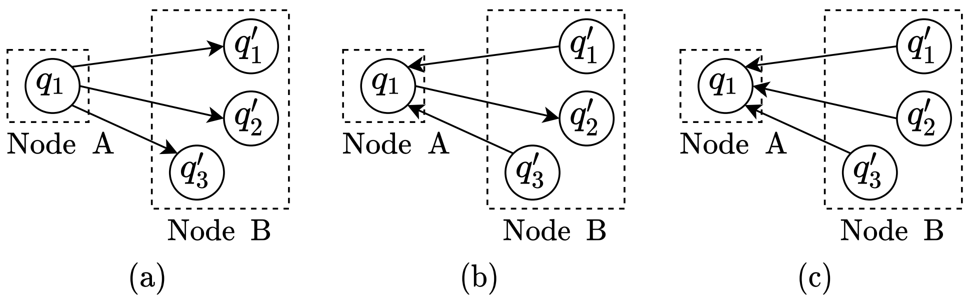


Figure 3.7: Two primitive communication patterns (a)(b) and the variant (c).

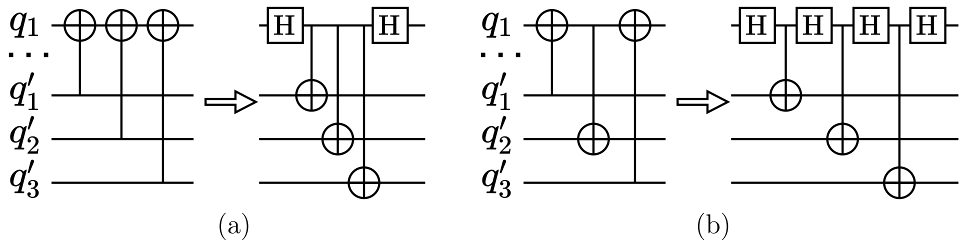


Figure 3.8: The transformation between communication patterns by using Hadamard gates.

implement communication blocks not being controlled-unitary. Compared to Cat-Comm, TP-Comm can implement any burst communication block with two EPR pairs: one to teleport q_1 to node B, the other to teleport q_1 back to node A, in order to handle the side effect of TP-Comm. There are cases we would like to teleport q_1 to another node instead of simply moving it back. We postpone the details to Section 3.3.4. Compared to Cat-Comm, the disadvantage of TP-Comm is that its EPR pair saving is at most $\frac{n}{2}$ times. Overall, Cat-Comm provides higher EPR pair savings for specific burst communication blocks, while TP-Comm can handle any burst communication block with up to two EPR pairs.

Pattern analysis: Figure 3.7(a)(b) shows two primitive patterns of burst communication. For the unidirectional communication pattern in Figure 3.7(a) where one qubit, i.e., q_1 always serves as the control qubit, the communication block can be implemented by Cat-Comm with only one EPR pair if no single-qubit gate on q_1 separates two-

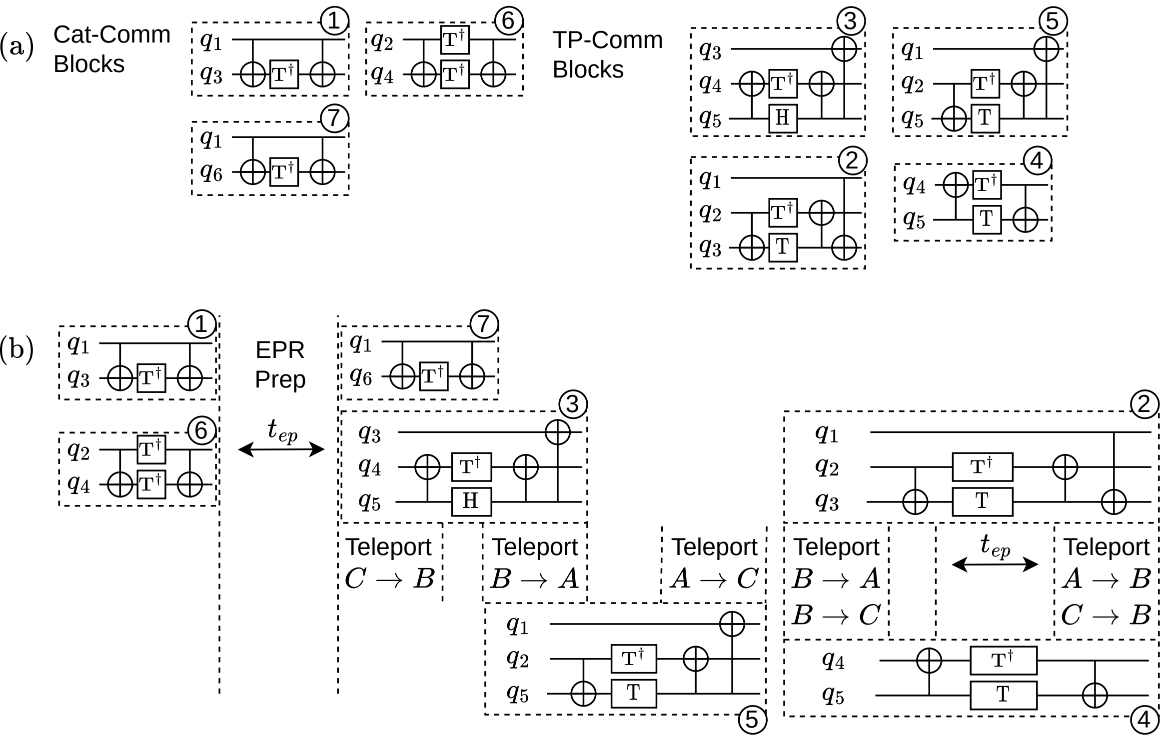


Figure 3.9: (a) The result of the communication assignment pass. (b) The result of the communication scheduling pass.

qubit gates [44]. For example, one call of Cat-Comm can handle the gate sequence $CX\, q_1, q_1'$; $CX\, q_1, q_2'$, but cannot address $CX\, q_1, q_1'$; $RZ\, q_1$; $CX\, q_1, q_2'$ due to the middle RZ gate. However, by moving RZ behind $CX\, q_1, q_2'$, the resulted gate sequence $CX\, q_1, q_1'$; $CX\, q_1, q_2'$; $RZ\, q_1$ only requires one invocation of Cat-Comm. Thus, to execute a communication block of unidirectional pattern with Cat-Comm, we should move single-qubit gates on the control qubit outside the communication block. Otherwise, we resort to TP-Comm. For example, to implement $CX\, q_1, q_1'$; $H\, q_1$; $CX\, q_1, q_2'$; $H\, q_1$; $CX\, q_1, q_3'$, it is better to use TP-Comm.

A varied unidirectional pattern in which q_1 always serves as the target qubit, as shown in Figure 3.7(c), also occurs frequently in distributed quantum programs. This pattern can be transformed into the pattern in Figure 3.7(a) by applying a series of Hadamard gates, as shown in Figure 3.8(a).

Figure 3.7(b) shows a bidirectional pattern in which q_1 serves as both control qubit and target qubit. Although we can transform a bidirectional communication block to be unidirectional as in Figure 3.8(b) with Hadamard gates, single-qubit gates on the control qubit may still prevent a cheap implementation by Cat-Comm. In fact, for the block in Figure 3.8(b), TP-Comm is more efficient as it only requires two EPR pairs, while Cat-Comm requires three EPR pairs.

To summarize, for unidirectional communication patterns in Figure 3.7(a)(c), we will try Cat-Comm first, while for the bidirectional pattern in Figure 3.7(b), TP-Comm is mostly preferred. The insight behind the conclusion is that, analogous to classical distributed computing, Cat-Comm only shares its read-only copy to another node, thus it is not a natural fit for bidirectional communications which involve read and write on the shared qubit. TP-Comm, in contrast, migrates data to another node, allowing read and write operations on the migrated qubit.

Communication scheme assignment: Now, we are ready to assign an optimal communication scheme to each burst communication block. Taking Figure 3.6(c) as an example, we assign Cat-Comm to unidirectional blocks ①, ⑥ (we can move the $T^\dagger$ gate on q_2 outside the communication block) and ⑦. We call these blocks *Cat-Comm blocks* for simplicity. We then assign TP-Comm to bidirectional blocks ②, ④ and ⑤. Likewise, we call them *TP-Comm blocks*. For ③, although being unidirectional, it cannot be executed by one invocation of Cat-Comm due to the H gate on q_5. Since executing it with either Cat-Comm or TP-Comm requires two EPR pairs, we set the TP-Comm assignment as default. The finalized communication assignment is shown in Figure 3.9(a).

3.3.4 Communication Scheduling

After optimizing the EPR pair consumption, we then schedule the execution of burst communication blocks to reduce the overall program latency and mitigate the effect of decoherence. Based on the quantitative data shown in Table 3.1, the preparation of remote EPR pairs is the most time-consuming one among various operations and hence should be carefully optimized to hide its latency. While the quantitative data may vary across quantum computing platforms, the schedule design in this section should be also effective.

Operation	Variable Name	Latency
Single-qubit gates	t_{1q}	$\sim$ 0.1 CX
CX and CZ gates	t_{2q}	1 CX
Measure	t_{ms}	5 CX
EPR preparation	t_{ep}	$\sim$ 12 CX
One-bit classical comm	t_{cb}	$\sim$ 1 CX

Table 3.1: The quantitative data of operations in DQC, extracted from [54, 55]. Latencies are normalized to CX counts.

The designs here aim to maximize the parallelism of communication blocks and shorten the latency of sequential communication by fusion.

More block-level parallelism: The essence of scheduling is to maximize the parallelism of a circuit. For burst communication blocks without nodes or qubits in common, they can be concurrently executed in nature. For blocks with shared nodes or qubits, their parallelism is limited by their commutability, as well as the communication resource each node holds. With the constraint that each node can establish only two communications in parallel, there is little room for lazy operations. We adopt a greedy strategy to execute commutable blocks, i.e., execute as many blocks as possible simultaneously, as soon as EPR pairs are prepared.

For Cat-Comm blocks, we can execute two commutable blocks in parallel at most if

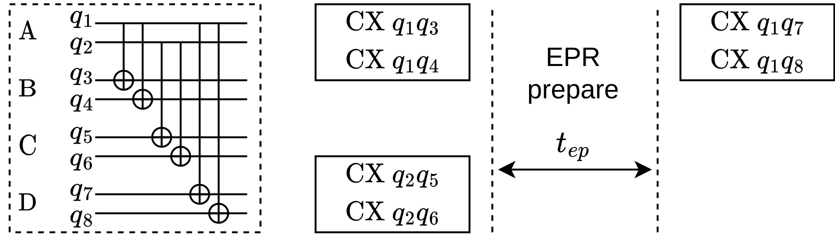


Figure 3.10: The schedule optimization for commutable Cat-Comm blocks, with shared qubit or node.

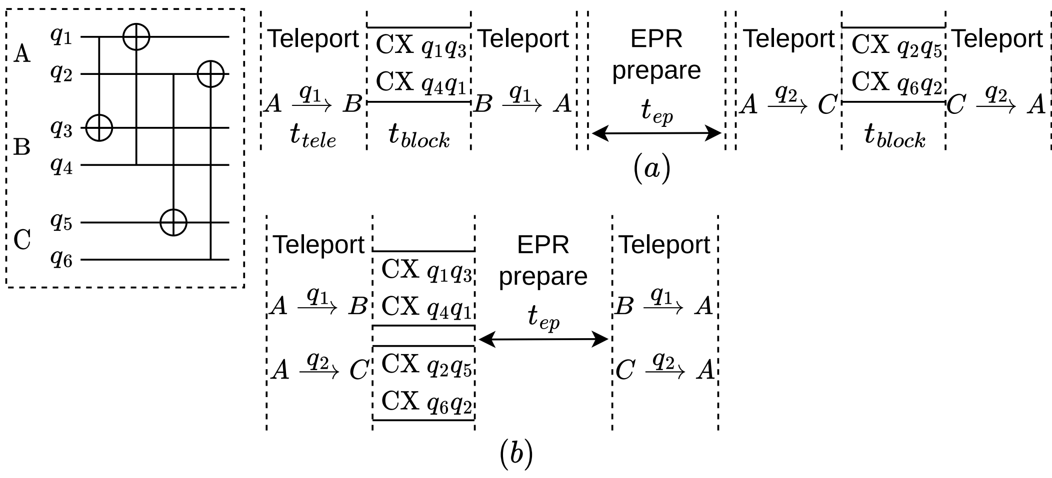


Figure 3.11: The schedule optimization for TP-Comm blocks. Aligned qubit teleportation in (b) is better than the independent qubit teleportation in (a).

they share nodes, as shown in Figure 3.10. For TP-Comm blocks, the situation is complex as each TP-Comm block requires two EPR pairs. For two commutable TP-Comm blocks, rather than prioritizing the completion of one TP-comm block as in Figure 3.11(a), we observe that parallelism can be enabled by communication alignment, as shown in Figure 3.11(b). Compared to Figure 3.11(a), Figure 3.11(b) aligns the qubit teleportation of two TP-Comm blocks, leading to a latency saving of $t_{block} + 2t_{tele}$. This TP-Comm alignment technique can be generalized to the case of n commutable TP-Comm blocks. With TP-Comm alignment, the total latency saving can be up to $(n-1)(t_{block}+2t_{tele})$ (e.g., if those TP-Comm blocks are on nodes $\{A_1, A_2\}, \{A_2, A_3\}, \cdots, \{A_n, A_{n+1}\}$ respectively).

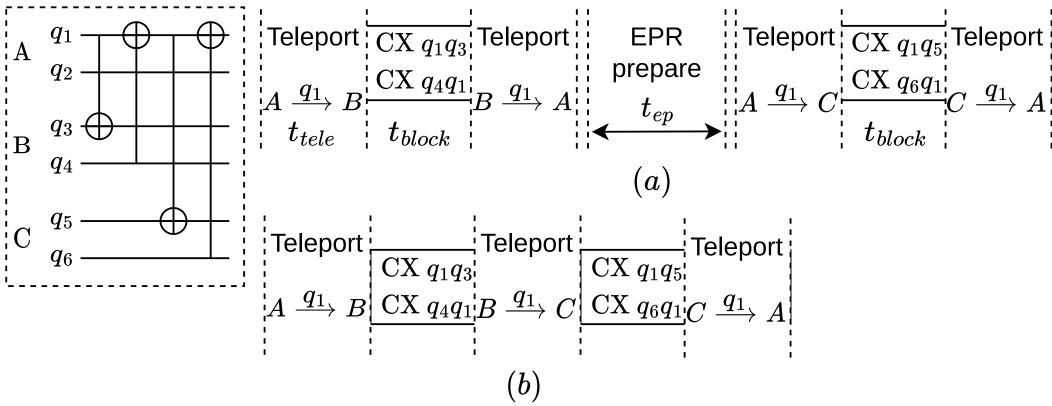


Figure 3.12: The schedule optimization for TP-Comm blocks. Cyclic qubit teleportation in (b) is better than the SWAP-style qubit teleportation in (a).

Fusion of sequential blocks: Sometimes communication blocks have to be executed in sequence. However, if one qubit is teleported across many TP-Comm blocks, we can shorten the latency of executing those TP-Comm blocks by fusing the teleportations, as shown in Figure 3.12. Figure 3.12(a) shows a simple schedule where each TP-Comm block is executed independently. As each node has only two communication qubits, we need to wait for t_{ep} before executing the next TP-Comm block. In contrast, Figure 3.12(b) fuses the teleportations between quantum nodes, forming a cycle: $A \to B \to C \to A$. With TP-Comm fusion, the number of teleportations is reduced by one and the overall execution time is reduced by $t_{ep} + t_{tele}$. Generally, if we have n TP-Comm blocks with the same teleported qubit, the total number of teleportation would be reduced by $n-2$, and the overall latency saving would be $(n-2)(t_{ep}+t_{tele})$. From another view, the fusion also optimizes the token passing problem in classical distributed computing [50], which also appears in Section 3.3.3, about whether to move the teleported qubit back or to another node, in order to handle the side effect of TP-Comm.

With the designs above, the communication scheduling pass should apply block-level commutation analysis to unveil the patterns discussed above and then apply corresponding optimizations. We omit the details since this procedure is very similar to the com-

munication aggregation except working at the block level. With all those optimizations applied, Figure 3.9(b) shows the optimized communication schedule for the example program in Figure 3.2. In total, 58.3% latency saving is achieved compared to executing each remote CX gate independently.

3.4 Evaluation

In this section, we first compare the performance of AutoComm to two baselines and then evaluate the effect of optimization passes in AutoComm. We finally perform a sensitivity analysis on AutoComm to study how its performance evolves as the experiment configuration changes.

3.4.1 Experiment Setup

(a) Platforms We perform all experiments on a Ubuntu 18.04 server with a 6-core Intel E5-2603v4 CPU and 32GB RAM. Other software includes Python 3.8.3 and Qiskit 0.18.3 [28].

(b) Benchmark programs We consider two categories of benchmark programs, as shown in Table 3.2. The first category of benchmarks focuses on implementing elementary functions, e.g., arithmetic operations and Fourier transformation. These quantum programs are often used as building blocks of large quantum applications. The second category of benchmarks aims to solve real-world problems, including Bernstein-Vazirani (BV) algorithm, Quantum Approximate Optimization algorithm (QAOA), and Unitary Coupled Cluster ansatzes (UCCSD). Specifically, for BV, we choose 1000 randomized secret strings which on average contain $\frac{2}{3}\#\,qubit$ nonzeros. For QAOA, we choose the graph maxcut problem (over 1000 randomized graphs), and for UCCSD, we select molecules

LiH, BeH_2, and CH_4 which correspond to programs with 8, 12 and 16 qubits, respectively. All benchmark programs used in the evaluation are collected from IBM Qiskit [28] and RevLib [51].

(c) Baseline We implement two state-of-the-art compilers, GP-Cat and GP-TP, which, to the best of our knowledge, represent the best efforts for distributed quantum compilation. GP-Cat implements one of the compiler designs proposed by [45] which exploits the Cat-Comm scheme for remote CX gates but does not consider burst communication. We did not extend GP-Cat to use TP-Comm as TP-Comm is not efficient at implementing a single remote CX gate. For the GP-TP baseline, we adopt a similar compiler design to [42, 45] where nonlocal operations are turned into local operations by swapping qubits between nodes. In GP-TP, we use TP-Comm to implement nonlocal qubit swapping operations as TP-Comm is better at implementing the remote SWAP gate than Cat-Comm. For both baselines and AutoComm, we map qubits to compute nodes by the 'Static Overall Extreme Exchange' (Abbrev. *SOEE*) strategy [42], which aims to reduce inter-node communication. To reduce the program latency, the baselines adopt a greedy scheduling method, i.e., executing operations as soon as possible.

(d) Distributed quantum computing model We assume a uniform DQC system. Each node in the DQC system has the same number of qubits. The fidelity of EPR pairs between any two nodes is the same, and so is the latency of preparing them. We also assume that the EPR pair can be established between any two nodes and each node has *two communication qubits* as in [42, 6]. Since our framework focuses on communication optimization, we assume a trapped-ion style device [40] for each compute node that any two local qubits can communicate with each other. Our work can also be easily applied to superconducting devices [56] with sparse two-qubit connections.

For Table 3.2, we assume each node has 10 data qubits for benchmark programs except UCCSD. For UCCSD, we assume each node has 2 data qubits. For all experiments, we assume that qubits of the test program are evenly distributed over all nodes.

(e) Metric The first metric is the total number of consumed EPR pairs for executing a distributed quantum program. Each invocation of Cat-Comm or TP-Comm requires one EPR pair. Note that without TP-Comm fusion, two EPR pairs are needed to execute one burst communication block with the TP-Comm, with one of the EPR pairs moving the teleported qubit back to its original node. The number of consumed EPR pairs models the resource overhead of executing distributed quantum programs and a lower value is favored.

The second metric is the maximum number of inter-node two-qubit gates got executed with one EPR pair. We denote this metric by 'Peak # REM CX'. To give a concrete example, assuming a distributed quantum program where the largest Cat-Comm block contains 10 remote CX gates and the largest TP-Comm block contains 18 remote CX gates, if without TP-Comm fusion, then for this program, 'Peak # REM CX' is $10 = \max(10, 18/2)$. If we assume the largest TP-Comm block is fused with the next TP-Comm block, then 'Peak # REM CX' is $18 = \max(10, 18)$ because TP-Comm fusion reduces the EPR pair consumption of a TP-Comm block. The metric 'Peak # REM CX' characterizes the communication throughput and a higher value is preferred.

Finally, we consider two metrics that model the relative performance of AutoComm to baselines, with respect to EPR consumption and program latency. The first one is the 'improv. factor', which is defined to be '# total EPR pairs by baseline/# total EPR pairs by AutoComm'. The second one is the 'LAT-DEC factor' that is defined to be 'program latency by baseline/program latency by AutoComm'. The target of AutoComm is to make these two metrics as large as possible.

Type	Name	# qubit	# node	# gate	# CX	# REM CX by SOEE
Building Blocks	Multi-Controlled Gate (MCTR)	100	10	10640	4560	1680
		200	20	21840	9360	3568
		300	30	33040	14160	5632
	Ripple-Carry Adder (RCA)	100	10	1569	785	99
		200	20	3169	1585	209
		300	30	4769	2385	319
	Quantum Fourier Transform (QFT)	100	10	19800	9900	9000
		200	20	79600	39800	38000
		300	30	179400	89700	87000
Real World Applications	Bernstein Vazirani (BV)	100	10	265	65	56
		200	20	535	135	126
		300	30	803	203	194
	QAOA	100	10	6000	4000	3144
		200	20	24000	16000	14076
		300	30	54000	36000	32896
	UCCSD	8	4	3129	1420	900
		12	6	40659	19142	15136
		16	8	129829	64956	53426

Table 3.2: Benchmark programs. # qubit is the total number of qubits and each node has exactly '# qubit/# node' data qubits.

3.4.2 Compared to Baselines

We first analyze the ability of AutoComm in exposing burst communications, with communication statistics shown in Figure 3.13. We then evaluate AutoComm and two baselines on benchmark programs in Table 3.2. The results of AutoComm and its relative performance to GP-Cat and GP-TP are shown in Table 3.3. When we say *on average* in this section, we refer to the *geometric mean*.

Burst communication statistics: Figure 3.13 shows the distribution of burst communications assembled by AutoComm. This distribution is closely related to the inverse-burst distribution discussed in Section 3.2.2 but is easier to compute. We can see that burst communications exist widely, no matter in building-block circuits (Figure 3.13(a))

or in real-world applications (Figure 3.13(b)). Moreover, Figure 3.13 demonstrates the effectiveness of AutoComm in unveiling burst communications. In Figure 3.13, the EPR pairs that each support ≥ 2 remote CX gates account for 76.8% of the overall consumed EPR pairs, on average.

Compared to GP-Cat: As shown in Table 3.3, AutoComm achieves significant reduction in both EPR pair consumption and program latency, compared to GP-Cat. Specifically, AutoComm reduces the number of consumed EPR pairs by a factor of 3.9x on average, up to 18.8x (ref. 'Improv. factor'). AutoComm also reduces the program latency by a factor of 3.1x on average, up to 9.4x (ref. 'LAT-DEC factor'). These significant improvements come from the high communication throughput enabled by AutoComm. In GP-Cat, each EPR pair, i.e., each invocation of Cat-Comm is used to implement only one remote CX gate. In contrast, the peak communication throughput (ref. 'Peak #REM CX') by AutoComm is 7.2x on average and up to 20x of that by GP-Cat. Those results indicate that AutoComm can efficiently discover and utilize burst communications, transferring more information with each EPR pair.

Compared to GP-TP: As shown in Table 3.3, AutoComm achieves significant reduction in both EPR pair consumption and program latency, compared to GP-TP. Specifically, AutoComm reduces the number of consumed EPR pairs by a factor of 3.5x on average, up to 13.3x. AutoComm also reduces the program latency by a factor of 3.4x on average, up to 10.7x. On the side of information theory, AutoComm enables a higher throughput of information. Each EPR pair in GP-TP carries 3/2 remote CX gates (i.e., a remote SWAP gate over two EPR pairs), much smaller than the throughput by AutoComm. On the algorithmic side, AutoComm avoids unnecessary qubit movement. For example, consider the gate sequence CX q_1, q_2; CX q_1, q_3; CX q_1, q_4; CX q_2, q_1 where q_1

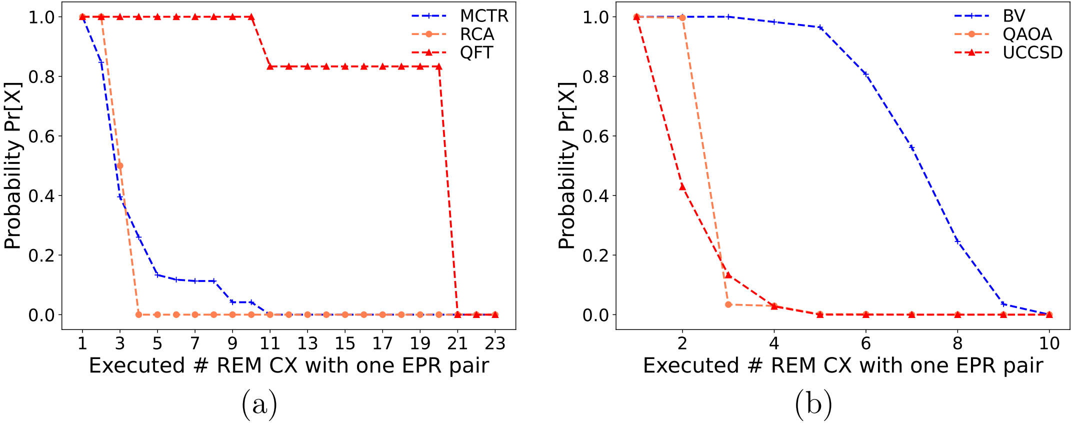


Figure 3.13: Burst communications by AutoComm: Pr[X] = Pr[one EPR pair supports $\geq$ X REM-CXs].

is in node A, q_2 is in node B, and q_3, q_4 are in node C. To execute these remote gates, GP-TP needs to swap q_1 into node B first, then to node C, and back to node B again. However, with AutoComm, we only need to first move q_1 to node C and then to node B, since CX q_1, q_2 is commutable with CX q_1, q_3 and CX q_1, q_4.

3.4.3 Optimization Analysis

In this section, we further analyze the effectiveness of each optimization pass in AutoComm. Again, when we say *on average* in this section, we refer to the *geometric mean*.

For simplicity, we denote the communication aggregation pass by *P1*, the assignment pass by *P2*, and the scheduling pass by *P3*. We first study how P1 and P2 affect the 'improv. factor' of AutoComm to GP-Cat, then evaluate how P3 affects the 'LAT-DEC factor'. The results are shown in Table 3.4. We do not compare P2 to GP-Cat directly as P2 cannot work properly without communication aggregation.

The effect of communication aggregation: As shown in Table 3.4, compared to GP-Cat, 'P1+Cat-Comm' reduces the EPR pair consumption by a factor of 2.6x, on

Name	# Tot. EPR pairs consumed		Peak # REM CX	Compared to GP-Cat		Compared to GP-TP	
	By Cat-Comm	By TP-Comm		Improv. factor	LAT-DEC factor	Improv. factor	LAT-DEC factor
MCTR-100-10	313	220	10	3.15	3.27	2.81	3.90
MCTR-200-20	554	418	10	3.67	3.83	3.26	4.51
MCTR-300-30	932	1112	10	2.76	2.88	2.45	3.39
RCA-100-10	0	36	3	2.75	2.22	2.00	1.37
RCA-200-20	0	76	3	2.75	2.26	2.00	1.38
RCA-300-30	0	116	3	2.75	2.27	2.00	1.38
QFT-100-10	0	540	20	16.67	9.35	4.67	3.24
QFT-200-20	0	2090	20	18.18	9.40	5.27	3.26
QFT-300-30	0	4640	20	18.75	9.41	5.50	3.26
BV-100-10	9	0	8	6.22	4.33	12.22	9.68
BV-200-20	19	0	8	6.63	4.63	13.16	10.47
BV-300-30	29	0	8	6.69	4.69	13.31	10.65
QAOA-100-10	1182	266	6	2.17	1.83	1.56	2.09
QAOA-200-20	6059	728	8	2.07	1.79	1.57	2.52
QAOA-300-30	14915	1138	6	2.05	1.69	1.62	2.68
UCCSD-8-4	464	0	4	1.94	1.74	3.97	4.08
UCCSD-12-6	8973	0	4	1.69	1.55	3.10	3.31
UCCSD-16-8	33303	0	5	1.60	1.50	3.02	3.29

Table 3.3: Results of AutoComm and its comparison to baselines. The first column contains acronyms of programs in Table 3.2.

average. The result indicates the effectiveness of the communication aggregation pass in reducing the communication cost by grouping remote CX gates into a burst communication block. On the other hand, this analysis also shows that burst communication may not be readily available in distributed quantum programs and we need the communication aggregation pass to unveil them.

The effect of communication assignment: As shown in Table 3.4, compared to 'P1+Cat-Comm', 'P1+P2' further reduces the EPR pair consumption by a factor of 1.4x, on average. The result demonstrates the importance of considering both Cat-Comm and TP-Comm for burst communication. The benefit of P2 is even more significant for programs where bidirectional communication patterns appear frequently, e.g., RCA and QFT. This is because Cat-Comm is not as efficient as TP-Comm for implementing bidirectional burst communication.

Name	Improv. factor compared to GP-Cat		LAT-DEC factor compared to GP-Cat	
	P1+Cat-Comm	P1+P2	P1+P2	P1+P2+P3
MCTR	3.05	3.17	2.76	3.30
RCA	1.88	2.75	2.25	2.25
QFT	2.22	10.00	7.14	9.39
BV	6.51	6.51	4.55	4.55
QAOA	2.08	2.10	1.65	1.77
UCCSD	1.74	1.74	1.59	1.59

Table 3.4: Optimization analysis for AutoComm. Results are averaged over programs in Table 3.2. 'P1+P2+P3' is just AutoComm.

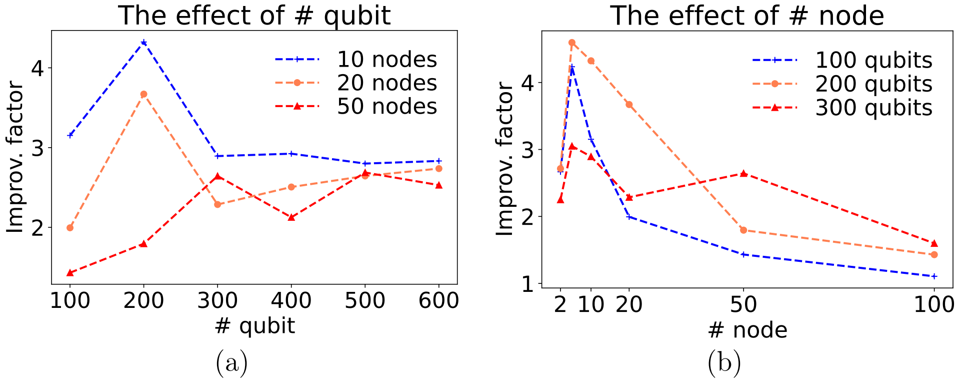


Figure 3.14: The effects of (a) #qubit and (b) #node on the 'improv. factor' of AutoComm when compared to GP-Cat. The test program is MCTR.

The effect of communication scheduling: As shown in Table 3.4, compared to 'P1+P2', 'P1+P2+P3' further reduces the program latency by a factor of 1.1x, on average. The result illustrates the effectiveness of P3 in reducing communication-induced latency. The effectiveness of P3 for scheduling burst communication stems from its smart utilization of communication qubits, especially for TP-Comm blocks, as discussed in Section 3.3.4. As for programs comprised of Cat-Comm blocks, e.g., BV and UCCSD, P3 behaves as efficiently as the default as-soon-as-possible scheduling method.

3.4.4 Sensitivity Analysis

The performance of AutoComm may be affected by factors like the number of program qubits, the number of DQC nodes, the qubit mapping, and the heterogeneity of compute nodes. In this section, we study how the performance of AutoComm changes as those factors varies.

When evaluating the effect of # qubit and # node (ref. Figure 3.14), we assume program qubits are evenly distributed over all nodes: each node has exactly '# qubit/# node' data qubits. We also assume two communication qubits per node.

The effect of # qubit: As shown in Figure 3.14(a), the 'improv. factor' of AutoComm converges when # qubit increases (i.e., # qubit/# node becomes large). The reason may be that the number of burst communication blocks also increases when the total number of remote multi-qubit gates grows with the number of program qubits. Such behavior is preferable because it illustrates that AutoComm can provide a consistent reduction of the communication overhead as the number of program qubits grows.

The effect of # node: As shown in Figure 3.14(b), the 'improv. factor' of AutoComm deteriorates when # node increases (i.e., # qubit/# node becomes small). On the one hand, the remote multi-qubit gate would proliferate when # node increases, potentially providing more chances for burst communication. On the other hand, it is harder to find large burst communication blocks when # qubit/# node becomes small, instead increasing the communication overhead. Overall, we should not use too many nodes for distributing programs.

The effect of qubit mapping: When evaluating the sensitivity to qubit mappings, we adapt two widely used algorithms, NoiseAdaptive [57] and SABRE [46] to benchmark

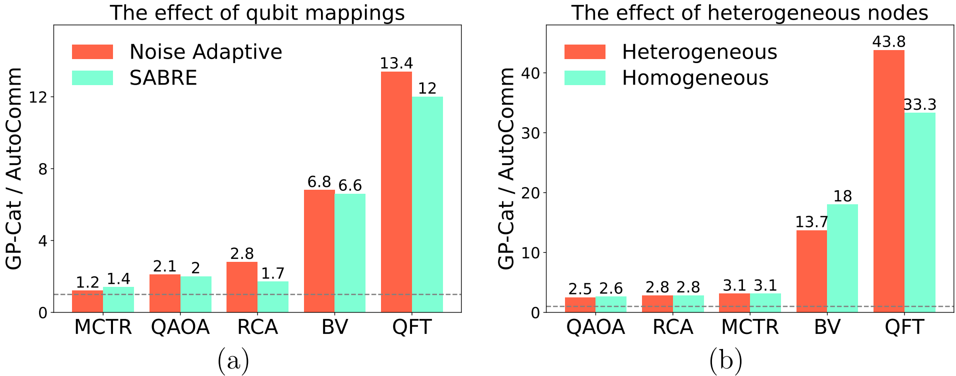


Figure 3.15: The effects of (a) qubit mappings and (b) heterogeneous nodes. Numbers in (a)(b) are averaged (geometric mean) 'improv. factor' of AutoComm to GP-Cat.

programs in Table 3.2. Such adaptations are straightforward as the DQC backend can also be described by the coupling graph. As shown in Figure 3.15(a), our framework still achieves significant communication cost reduction with NoiseAdaptive and SABRE. This indicates the practicality of AutoComm's two-step compilation design (ref. Figure 3.1), which enables us to focus on communication optimization while leveraging tons of existing efforts on qubit mapping.

The effect of heterogeneous nodes: For this analysis, we consider two settings: the heterogeneous setting distributes each 100-qubit program over 4 nodes with 10, 20, 30, and 40 data qubits, respectively; the homogeneous setting evenly distributes each program over 4 nodes with 25 data qubits per node. As shown in Figure 3.15(b), our framework still achieves significant communication cost reduction on heterogeneous nodes. In the heterogeneous setting, nodes with few qubits limit the benefits of burst communication while nodes with many qubits boost them. These two effects cancel out each other and guarantee the performance of AutoComm.

Chapter 4

Optimizing Collective Communication for Distributed Quantum Computing

In this chapter, we will delve into optimizing the collective communication pattern commonly found in distributed quantum programs.

4.1 Introduction

Quantum computing is promising and can be used to solve classically intractable problems [58, 59]. One critical problem that hinders the practical application of quantum computing is the limited qubit resource of quantum computers. Distributed quantum computing (DQC) provides an optimistic way to scale up quantum computing and has been demonstrated in recent experiments [60, 7, 61]. Research attention on DQC emerges both in hardware design [62, 60, 61, 7] and program compilation [63, 64, 5, 65, 66, 67, 45, 6, 68, 9]. Distributed quantum computing, as shown in Figure 4.1(a), integrates many independently fabricated quantum processors (aka compute nodes) to run quantum programs.

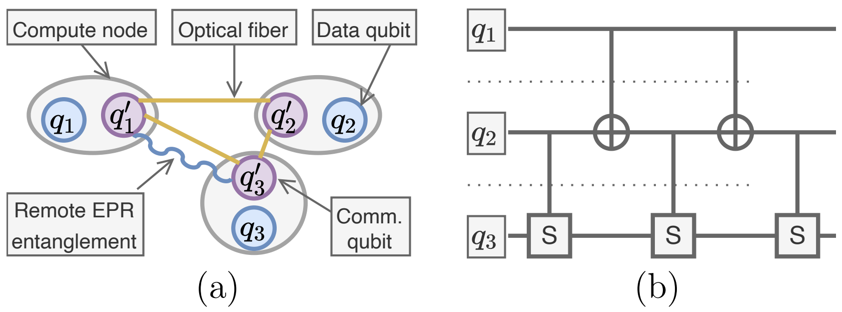


Figure 4.1: (a) The common distributed quantum computing architecture [62]. Nodes form a quantum network. Communication qubits emit photons, which are transferred through optical fibers, to establish remote entanglement. Data qubits are used to store program information. (b) An exemplar distributed circuit of the decomposed CCZ gate.

DQC relies on inter-node quantum communication to share or move quantum data between compute nodes so that nonlocal gates (e.g., the CX gates in Figure 4.1(b)) become executable. In the common DQC model [5, 45, 6, 67, 68, 66, 63, 64, 65, 9], each invocation of inter-node communication consumes one remote EPR pair while the generation of remote EPR pairs between two different nodes is far more error-prone and time-consuming than applying local quantum gates [60]. To mitigate the infidelity caused by inter-node communication, researchers have investigated many strategies, e.g., designing more reliable communication hardware [62], employing quantum error correction (QEC) [69], and using EPR entanglement purification [70]. A much cheaper strategy to mitigate infidelity, however, is to reduce the amount of inter-node communication needed in a distributed quantum program through a DQC compiler. This strategy is always useful, regardless of hardware or QEC/entanglement purification. We focus on designing a compiler for reducing inter-node communication.

Unfortunately, existing compilers for DQC lack deep analysis of distributed quantum programs and are limited to either qubit-to-qubit communication or qubit-to-node communication. Most DQC compilers focus on either optimizing the qubit layout [63, 64, 5, 65, 66, 67] to reduce nonlocal gates or shortening the communication footprint of apply-

ing each nonlocal gate [45, 6]. Those works do not inspect the intrinsic communication patterns in distributed quantum programs. State-of-the-art DQC compilers such as [9] identify the burst communication pattern between one qubit and one node and propose executing a group of nonlocal gates together by 1-2 invocations of inter-node communication. Though significantly better than previous works, their work still does not consider the communication among multiple nodes, similar to other DQC compilers. Considering the example circuit in Figure 4.1(b) where there exists a group of nonlocal gates across three nodes (i.e., collective communication), existing DQC compilers [45, 6, 9] would require 5 invocations of inter-node communication for the 5 inter-node gates. In contrast, if we implement those inter-node gates by moving both q_1 and q_2 to q_3's node and keep them there, only 2 invocations of inter-node communication are needed.

Therefore, considering collective communication would enable a wider scope on optimizing distributed quantum programs and present optimization opportunities invisible from the low level. We formally define a *collective communication* block as a group of inter-node gates whose inter-node qubit interaction forms a connected graph over multiple nodes. We require each collective communication block to absorb quantum gates as much as possible, as long as the overall communication cost is reduced. Moreover, we identify three key challenges for utilizing collective communication patterns to reduce inter-node communication. Firstly, *in the program level*, collective communication is usually not directly accessible. For many quantum circuits, e.g., those decomposed to the Clifford+T basis [1] or the CX+U3 basis [28], collective communication is hidden in the details of scattered inter-node CX. Secondly, *for an arbitrary DQC network topology*, it is unclear how to route collective communication based on its patterns, e.g., into which node we place all involved qubits could lead to the least inter-node communication. No such problem exists in routing the two-qubit gate. Finally, *for the lowest-level implementation*, the DQC system may not have enough resources to directly support a collective

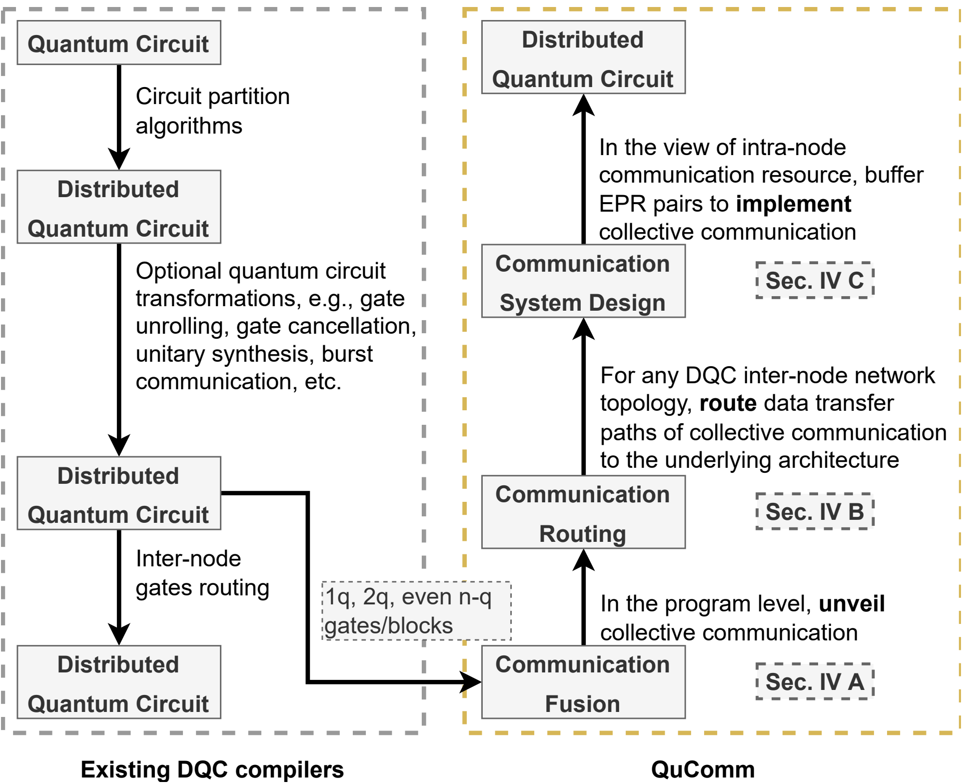


Figure 4.2: The compilation flow of quantum circuits on the DQC architecture and the overview of QuComm.

communication block, e.g., the block may require one node to have several remote EPR pairs ready simultaneously. This is harsh considering the potentially limited number of communication qubits per node [60, 7].

The identified challenges require three continuous compiler optimizations. To this end, we developed the first compiler framework, named *QuComm*. As shown in Figure 4.2, QuComm consists of three key stages for collective communication optimization which is unexplored by existing DQC compilers. The first stage is *communication fusion*, which inspects program information, aiming to unveil collective communication blocks from low-level circuit details. The insight is that a collective communication block should require less inter-node communication for execution, compared to implementing each nonlocal gate independently. The second stage is *communication routing*,

which further incorporates DQC network topology information. The insight is that the overall communication footprint can be reduced by adapting the data transfer path of target collective communication to the underlying DQC architecture. The final stage is *communication system design*, which concerns the lowest-level implementation of routed collective communication blocks in view of intra-node communication resources. We formalize the concept of the communication buffer that utilizes data qubits for buffering remote EPR pairs so that large collective communication blocks are executable. Evaluation shows that, QuComm reduces the amount of inter-node communication by 54.9% on average, over various distributed quantum programs and DQC hardware configurations, compared to the state-of-the-art baseline [9].

4.2 Problem and Motivation

In this section, we study the collective communication hidden in distributed quantum programs and discuss the opportunities and challenges of collective communication optimization.

4.2.1 Collective Communication in DQC

Being essential, inter-node communication greatly degrades the fidelity of distributed quantum programs [54]. The main goal here is to reduce the amount of inter-node communication in distributed quantum programs by efficient compilation. Our insight of compiler optimizations originates from our analysis of collective communication. **We formally define a collective communication block as a group of inter-node gates which has a connected inter-node interaction graph on qubits over multiple nodes**. In the interaction graph, We would draw an edge between any two qubits if they are involved in the same inter-node gate. We require the interaction graph to

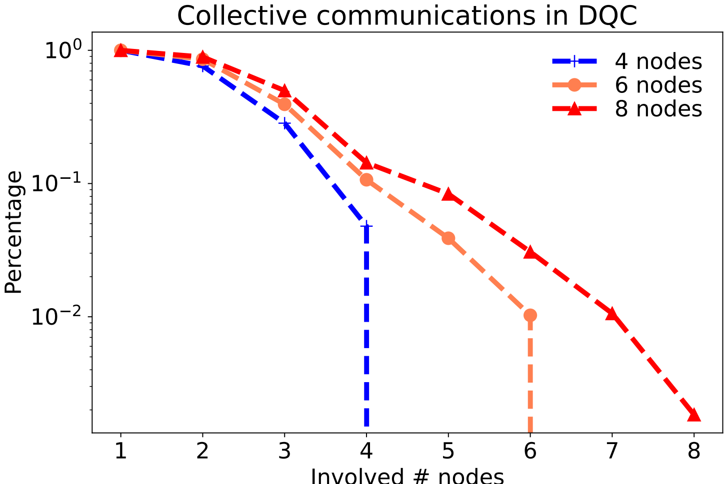


Figure 4.3: Collective communication from quantum circuits in [51] with the OEE qubit-node mapping [71]. Dot (x,y%) on each curve means there are y% multi-qubit gates involving at least x node. Results are averaged over circuits in [51].

be connected since in coherent collective communication, nonlocal gates should depend on each other (in terms of qubits). Our definition of collective communication is more general and flexible than the one in [68], containing but not limited to broadcast, reduce, etc.

We observe that **distributed quantum programs have abundant collective communication**. By collecting statistics on various quantum circuits (arithmetic functions, encoding circuits, etc.) from the widely studied quantum benchmark [51] (having circuits up to 143 qubits), we observe that on average 77.6%, 20.2%, and 10.6% of quantum gates involve more than 3, 6, and 9 qubits, respectively. This demonstrates the potential existence of abundant collective communications when executing these circuits on DQC hardware. Figure 4.3 shows the statistics of collective communication when qubits of various quantum circuits in [51] are mapped to DQC systems with 4, 6, and 8 nodes (blue, orange, and red curves) with each node holding '# circuit qubit/# node', respectively. The qubit-node mapping uses the widely-adopted OEE algorithm which tries to maximally reduce inter-node quantum gates. As shown in Figure 4.3, there are

about 28.4% of the multi-qubit gates require communications between more than 3 nodes, when running circuits on 4 compute nodes. The percentage grows up when we run these circuits on 8 compute nodes, where 49.8% of multi-qubit gates involve computation on more than 3 nodes.

In summary, we observe that the efficient implementation of collective communication in distributed quantum programs is critical to promoting DQC's computational potential.

4.2.2 Opportunities and Challenges

First, we can greatly reduce the amount of inter-node communication through pattern analysis of collective communication. Let us revisit the distributed quantum circuit in Figure 4.1(b). Realizing that the circuit forms a collective communication on three nodes, we can decrease the amount of inter-node communication from 5 (if using existing DQC compilers [9]) to 2 (by placing all three qubits into the same node). This example demonstrates the importance of **implementing inter-node gates collectively**, i.e., analyzing the pattern of a group of inter-node gates as a coherent whole (i.e., collective communication) and finding the most communication-efficient way to implement them from a higher level.

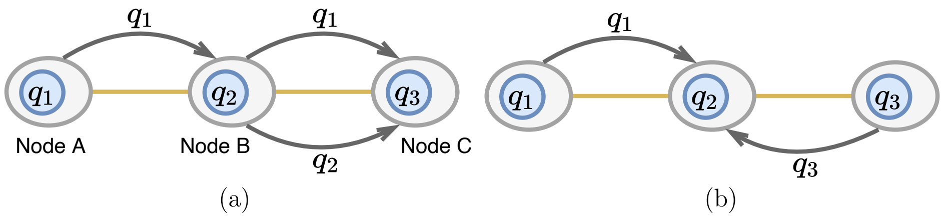


Figure 4.4: Two examples of routing the collective communication block in Figure 4.1(b) onto the nearest-neighbor architecture. (a) Data path: move q_1 and q_2 to node C. (b) Data path: move q_1 and q_3 to node B.

Second, new opportunities emerge when we route collective communication to a DQC system without full connection between nodes. Let us now consider routing the dis-

tributed circuit in Figure 4.1(b) onto the nearest-neighbor DQC architecture [72]. Two routing examples are presented in Figure 4.4. The first example (Figure 4.4(a)) requires 2+1 invocations of inter-node communication, with one of the three being the routing overhead. However, **by redesigning the data paths of the collective communication to better align with the underlying DQC network topology**, we can reduce the routing overhead from one inter-node communication to zero, as shown in Figure 4.4(b). This is a unique optimization opportunity for collective communication. There is no such design space on the data path of communication between two nodes.

Finally, another optimization opportunity emerges from the underlying system design. There are cases where the underlying DQC system may not always have enough communication resources to support collective communication. For example, for the collective communication block in Figure 4.1(b), if each node only has one communication qubit and does not use data qubits to store the generated remote EPR pairs, then each node at most accommodates one EPR pair at any time, making it impossible to simultaneously move both q_1 and q_2 to the node holding q_3. However, if we use two data qubits to buffer the EPR pairs generated by the communication qubit, it becomes possible for one node to have two EPR pairs at the same time, making the collective communication in Figure 4.1(b) directly executable. Overall, an EPR pair buffer is critical to enable collective communication, especially for DQC nodes with a limited number of communication qubits.

While being promising for reducing DQC's communication overhead, the identified optimization opportunities also impose difficulties for the compiler design:

1) Collective communication is usually not directly accessible. For circuits decomposed to basic gates, collective communication is hidden in the details of scattered inter-node gates. Collective communication is also affected by qubit placement, gate ordering, etc.

2) Given a DQC network topology, it is unclear how to efficiently utilize collective com-

munication patterns to route nonlocal operations. Existing DQC compilers lack a higher-level routing that considers both architecture information and communication patterns. **3)** The DQC system design directly affects the efficiency of executing collective communication. This design needs high-level information like the collective communication blocks one node needs to accommodate, which is hard to extract without deep program analysis.

4.3 Collective Communication System Design

In this section, we introduce the compiler designs that tackle the identified challenges and enable efficient utilization of high-level information in collective communication to reduce inter-node communication in distributed quantum programs. The qubit in this section can be a logical qubit or just a physical qubit.

QuComm includes three stages: the *communication fusion* which is used to unveil collective communication from circuit details, the *communication routing* which utilizes identified collective communication patterns to route inter-node gates onto the underlying DQC architecture, and the *communication system design* that improves the efficiency of collective communication by buffering EPR pairs.

4.3.1 Communication Fusion

The availability of collective communication in distributed quantum programs is affected by various factors, e.g., whether the distributed program is decomposed or not, the qubit mapping onto each node, and the gate ordering. The *insight* for identifying collective communication is that inter-node gates forming collective communication should require much less inter-node communication when implemented collectively, compared to implementing each of them independently. Based on the insight, we adopt a greedy

Algorithm 2: Communication fusion stage

```
   Input: Distributed quantum circuit circ
   Output: Collective communication blocks blk_list
 1 blk_list = [];
 2 Initialize a new block b0 with the next nonlocal gate;
 3 while blk_list continues grow do
       /* The aggregation step                                  */
 4     b1 = []; // The block next to b0
 5     for nonlocal gate g in the remaining circuit do
 6         if g has overlapped qubits with b0 then
 7             if possible, move g to b1 by circuit rewriting;
 8         end
 9     end
       /* The fusion step                                       */
10     while b1 is not empty do
           // cost is defined in Equation (4.1)
11         if cost(b0 + b1) < cost(b0) + cost(b1) then
12             merge b1 into b0;
13             b1 = [];
14         else
15             pop the last gate of b1 out;
16         end
17     end
       /* Terminating fusion                                    */
18     while the next gate g to b0 is local do
19         if cost(b0 + g) == cost(b0) then
20             merge g into b0;
21         else
22             break;
23         end
24     end
25     blk_list.append(b0);
26     Initialize a new block b0 with the next nonlocal gate;
27 end
28 output blk_list;
```

strategy to construct collective communication blocks, as shown in Algorithm 2.

To enable node-aware search while being general for any network topology among nodes, this stage only requires information about the maximum number of EPR pairs each node can accommodate at the same time, i.e., the **EPR capacity** of each node. The

EPR capacity of a node also indicates the maximum number of external qubits this node can hold simultaneously. For an ideal DQC architecture where each node has infinite communication qubits, the EPR capacity per node is ∞. The output of Algorithm 2 would contain a series of collective communication blocks. There are two important steps in Algorithm 2:

1) Aggregation: This step is to maximize collective communication opportunities through circuit rewriting (with rules in [53]), regardless of how scattered the input circuit is.

2) Fusion: This step is based on the insight that efficiently constructed collective communication blocks should always lead to less inter-node communication.

Let $E(n_a)$ be the EPR capacity of node n_a; $C(b_0)$ be $\{\#$ qubits involved in $b_0\}$; $C(b_0 - n_a)$ be $\{\#$ qubits involved in b_0 but not in $n_a\}$. Then we define the cost of implementing $b_0 + b_1$ as follows:

$$\min_{n_a \in nodes}\{\max\left(2 * (C(b_0 + b_1 - n_a) - E(n_a)), 0\right) + \min(E(n_a), C(b_0 + b_1 - n_a))\} \tag{4.1}$$

That is to say, if $C(b_0 + b_1) \leq E(n_a)$, we can simply transfer all qubits involved in b_0 and b_1 to n_a; otherwise, for each remaining qubit, we will perform an inter-node SWAP gate (by two TP-Comm or three Cat-Comm) to exchange it into n_a (ref. the first summand in Equation (4.1)). Equation (4.1) is only an estimation of the implementation cost but serves as a good metric for identifying profitable fusion.

To demonstrate Algorithm 2, let us consider the example circuit in Figure 4.5(a). The finalized circuit after the communication fusion stage is shown in Figure 4.5(b). The first collective communication block ① starts from gates between node A and node B. The gates between node B and node C will be merged into block ① since three

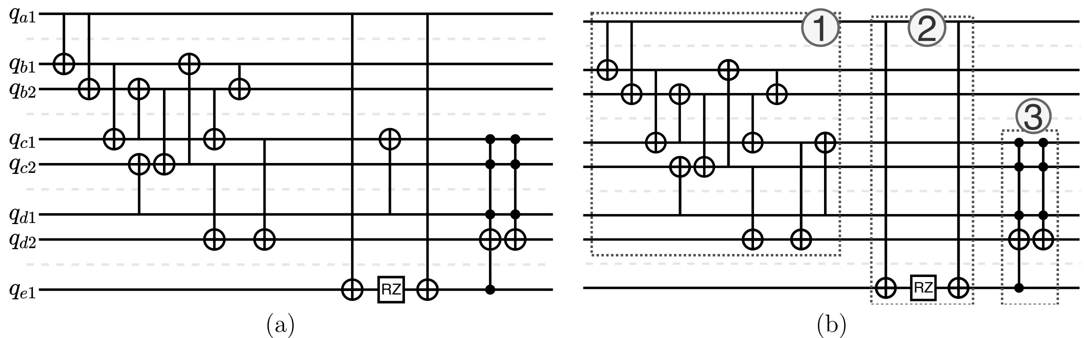


Figure 4.5: (a) An example distributed circuit for illustrating Algorithm 2. q_{a*}, q_{b*}, q_{c*}, q_{d*}, q_{e*} are in node A, B, C, D and E, respectively. We assume each node's EPR capacity is 5. (b) The circuit after communication fusion.

invocations of inter-node communication are reduced when implementing them together, according to Equation (4.1). Block ① is further enlarged by incorporating gates between node C and node D (note that the gate $\mathrm{CX}\, \mathrm{q_{d1}}, \mathrm{q_{c1}}$ is aggregated to block ① by circuit rewriting). Block ① also contains the local gate $\mathrm{CX}\, \mathrm{q_{b1}}, \mathrm{q_{b2}}$ since it does not incur extra communication. Unfortunately, the two gates between node A and node E cannot be merged into block ① as no communication reduction is observed. The gates between node A and node E then form block ②. Block ② is also a collective communication block if node A and node E are not directly connected where we need intermediate nodes to relay quantum data transfer.

4.3.2 Communication Routing

With the identified collective communication blocks, we then need to route these blocks for the underlying DQC network topology. The *insight* of this stage is to match the data transfer path of collective communication with the underlying DQC architecture so that the inter-node communication induced by routing overhead can be reduced. In this stage, we would examine the pattern of collective communication and identify efficient routing optimizations correspondingly.

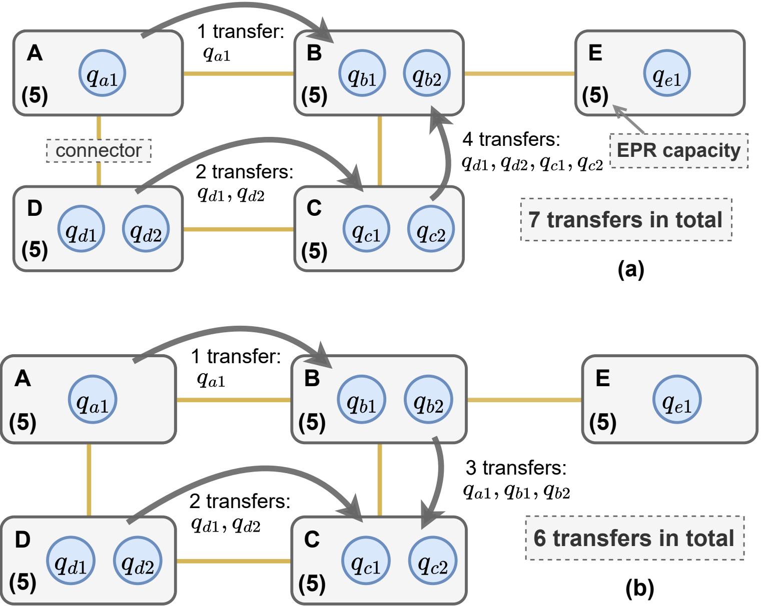


Figure 4.6: Examples of selecting the node for qubit aggregation on a nearest-neighbor DQC architecture, targeting block ① in Figure 4.5(b). Gray curves represent data paths. (a) When node B is selected. (b) When node C is selected.

Routing an individual collective communication block: The main idea is still to move/share all involved qubits to the same node while we can transform the data transfer path to fit into the underlying DQC network. The length of data path $L(pth)$ in the DQC network is computed as

$$L(pth) = \sum_{link \in pth} link.weight \tag{4.2}$$

The weight can be distilled/raw EPR fidelity of each inter-node link or just 1 for the uniform DQC hardware. Without loss of generality, here we assume all link weights are 1. Overall, we identify three routing optimizations for data transfer paths. Firstly, we should select the node for qubit aggregation based on the underlying network topology information along with the EPR capacity for each node. For example, for the collective communication block ① in Figure 4.5(b), if the underlying DQC network is fully-connected, it makes no difference to transfer all qubits to node B or node C. However, for the nearest-neighbor DQC architecture in Figure 4.6, transferring all qubits to node C is

cheaper (ref. Figure 4.6(b)). With the node for qubit aggregation selected, we pick data transfer schemes as suggested by [9]: it is more efficient to use Cat-Comm for read-only data transfer and TP-Comm for writable data transfer. For example, q_{a1} is transferred by Cat-Comm while q_{b1} is transferred by TP-Comm.

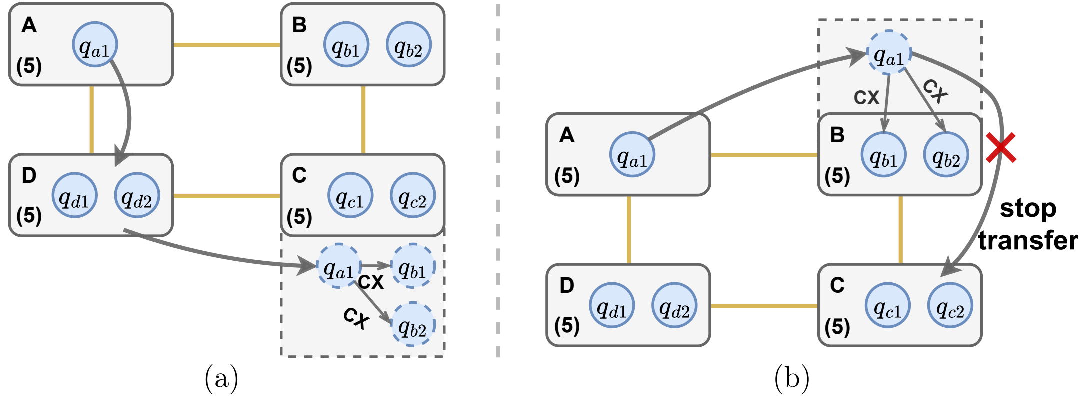


Figure 4.7: Two shortest paths for sharing q_{a1} to node C. The path in (b) enables early execution of CX gates between q_{a1} and node C. Gray arrows between data qubits mean CX gates. Data transfer paths for other qubits are omitted.

Secondly, enabling early execution along the data path can eliminate unnecessary data transfer. With the node for qubit aggregation determined (node C in Figure 4.6), the next step is to determine the data transfer path for each involved qubit. Figure 4.7 shows two different shortest data paths that *share* q_{a1} to node C. Compared to the path in Figure 4.7(a), the path in Figure 4.7(b) enables the execution of the two CX gates between q_{a1}, q_{b1} and q_{b2} in node B. Since q_{a1} is not involved in later inter-node operations, we would stop further transferring of q_{a1} to node C, saving one invocation of inter-node communication. Thus, we should consider not only the data path length but also the early-execution opportunity along the path when scheduling the transfer of a qubit.

Thirdly, the early execution strategy can be further extended to the parity computation process of large multi-controlled blocks to reduce the amount of inter-node communication. An n-qubit generalized Toffoli gate can be seen as computing the parity of the

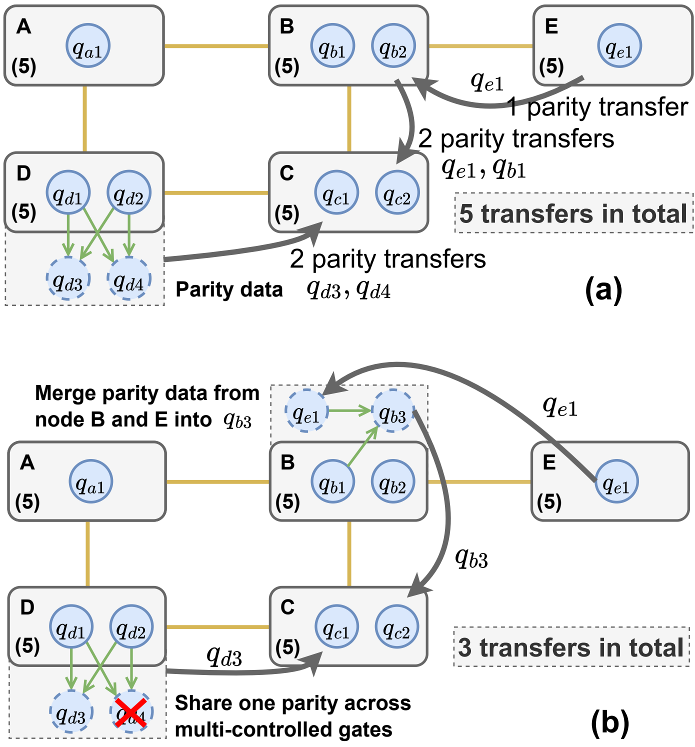


Figure 4.8: Optimizing parity propagation of multi-controlled gates in block ③ in Figure 4.5. Green arrows from $\{q_{d1}, q_{d2}\}$ to q_{d3} means computing parity on the group $\{q_{d1}, q_{d2}\}$ and storing it to q_{d3} (i.e., a Toffoli gate on q_{d1}, q_{d2}, q_{d3}). (a) The parity paths by [73]. (b) Our two optimizations on parity paths.

(n-1) control lines and this parity computing process can be decomposed by separating control lines into different groups [74]. For example, CCCCX $\mathrm{q}_0, \mathrm{q}_1, \mathrm{q}_2, \mathrm{q}_3, \mathrm{q}_4$ can be decomposed into gates CCX $\mathrm{q}_0, \mathrm{q}_1, \mathrm{q}_5$; CCX $\mathrm{q}_2, \mathrm{q}_3, \mathrm{q}_6$; CCX $\mathrm{q}_5, \mathrm{q}_6, \mathrm{q}_4$; CCX $\mathrm{q}_0, \mathrm{q}_1,$ q_5; CCX $\mathrm{q}_2, \mathrm{q}_3, \mathrm{q}_6$, where the parity computing results on groups $\{q_0, q_1\}$ and $\{q_2, q_3\}$ are stored in q_5 and q_6, respectively. Here in this stage, we will implement a group of multi-controlled gates collectively and adaptively on the underlying DQC architecture which may not be fully-connected, for the first time. As an example, given the collective communication block ③ in Figure 4.5(b), we would analyze the overall parity propagation and apply routing optimizations correspondingly:

1) Merge parity along the propagation path (i.e., early execution). As in Figure 4.8(a),

when transferring the parity computed in node E to the node for qubit aggregation (i.e., node C), the parity data would go by node B. Since node B also have parity data, we can combine the parity data from node E and B into one, as shown in Figure 4.8(b). One remote communication is thus reduced due to reduced parity data.

2) Share parity across multi-controlled gates. For example in Figure 4.8(b), the multi-controlled gates in block ③ both depend on the parity from the group $\{q_{d1}, q_{d2}\}$, thus we can avoid recomputing and resending the parity data from the group $\{q_{d1}, q_{d2}\}$, saving one inter-node communication.

After the collective communication block is executed, the node for qubit aggregation may be occupied by external qubits and cannot accommodate more EPR pairs and other external qubits. To release the occupation, we would inspect future collective communication blocks and transfer those external qubits to positions where they are needed for future multi-qubit gates. This process may help reduce the transition overhead between collective communication blocks, as discussed below.

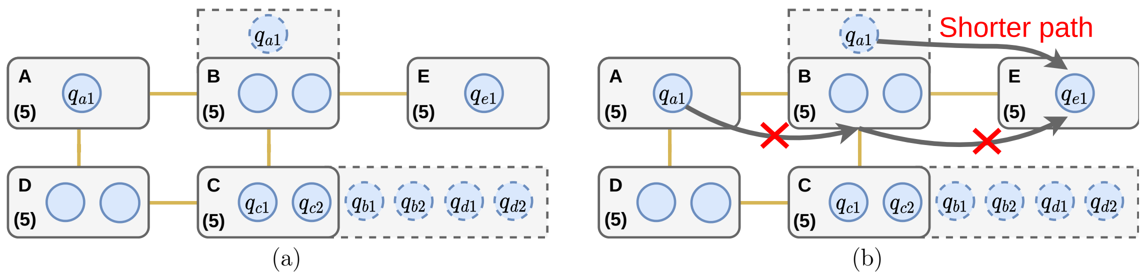


Figure 4.9: (a) The qubit layout after executing collective communication block ①. (b) It is shorter to transfer the read-only copy of q_{a1} (by Cat-Comm) from node B to node E rather than from node A to node E.

Routing transition between collective communication blocks: When transiting the routing from one collective communication block to another one, we can use data transfer that happened in the former one to reduce the routing overhead of the current block. For example in Figure 4.9(a), after routing block ①, q_{a1} is coherently in node A and node B. Therefore, to execute the CX in block ②, we can use Cat-Comm to share q_{a1}

Node C	Qubit list | EPR capacity		
	$q_{c1}, q_{c2}, q_{c3}, q_{c4}$ | 3	q_{c1}, q_{c2}, q_{c3} | 4	q_{c1}, q_{c2} | 5
	# inter-node comm		
Block ①	7	5	5
Block ③	3	3	3
Local Gates: 4 CX on q_{c3}, q_{c1}, 1 CX on q_{c4}, q_{c1}	0	≤ 1	≤ 5
Overall	10	≤ 9	stop

Table 4.1: The table shows the configuration process on node C, assuming each node only has 3 communication qubits, i.e. EPR capacity is 3. Block ① and ③ are from Figure 4.5(b) where each node's EPR capacity is 5. '# inter-node comm' is derived from Sec. 4.3.2.

from node B directly, saving one inter-node communication, as shown in Figure 4.9(b). Thus, when transiting between two collective communication blocks, we should first inspect the qubit layout change (e.g., one qubit may coherently exist in multiple nodes) caused by the former block and then shorten the data transfer path of the next block accordingly.

4.3.3 Communication Buffer Design

As stated in Sec. 4.2.2, # communication qubits on each node may potentially limit the efficient execution of collective communication. *Our insight* to overcome it is to use data qubits to buffer (through using local SWAP gates) remote EPR pairs generated by the communication qubits so that we can use these data qubits to accommodate data of external qubits. We say those data qubits form a communication buffer. This is the first formalization of the communication buffer concept in DQC compilers. The communication buffer essentially provides an abstraction or intermediate layer that is able to approximate the ideal DQC hardware (the one with infinite communication qubits). As

long as the communication buffer is large enough, we can implement collective communication without inter-node qubit swapping.

The size of the communication buffer requires careful design. Using too many data qubits in the communication buffer would cause each node to have fewer qubits to store program information, thus requesting more nodes to support the same program. Intuitively, for a given program, using more nodes would induce more inter-node communication. However, if we use only a few data qubits in the communication buffer, the communication reduction by optimizing collective communication would be small as well. To balance these two effects, we propose a program-adaptive communication buffer design so that the communication buffer in each node is just able to support collective communication in the program.

In this stage, we first perform the communication fusion stage assuming each node has a large number of communication qubits, obtaining collective communication blocks related to each node. We then configure the communication buffer of each node, starting from the node associated with most inter-node gates, with the following steps:

1) For a node n_i, find the qubit q_0 in n_i which incurs the least increment (say, $I(q_0)$) of inter-node communication when it is moved to another node (say, n_j) with idle data qubits. $I(q_0)$ can be easily computed by counting multi-qubit gates that involve q_0, in n_i and n_j.

2) According to Equ (4.1), re-inspect collective communication blocks associated with n_i to compute overall inter-node communication reduction by adding one data qubit in the communication buffer of n_i. Denote the reduction by $R(n_i)$.

3) If $I(q_0) \leq R(n_i)$, we would place q_0 into n_j, and add one data qubit in the communication buffer of n_i. Repeat this process until there is no further improvement.

The proposed communication buffer design only increases the size of a communication buffer if and only if the amount of inter-node communication is reduced. The whole

configuration process is computationally-cheap and iterates over each node linearly. The iteration cost on each node is bounded by the size of related collective communication blocks, which is often a constant factor. Table 4.1 shows an example of configuring the communication buffer on node C. The overall inter-node communication is reduced when moving q_{c4} to another node. However, we may not gain any benefit by moving q_{c3}. The configuration process thus terminates.

4.4 Evaluation

In this section, we compare the performance of QuComm to the baseline [9] and analyze the effect of optimizations proposed in QuComm.

4.4.1 Experiment Setup

DQC hardware model. For evaluation, we adopt the mesh-grid network [8] for DQC: . In the DQC architecture, we assume 8 compute nodes and 40 data qubits per node. We assume each data qubit is a logical qubit protected by QEC codes [69, 20]. Thousands of physical qubits may be required to build one logical qubit. We also assume each compute node has an independent magic state distillation unit [25] to enable local logical T gates. Further, we assume each node can only establish communication with neighboring nodes. We consider configurations of 1 or 3 or 5 logical communication qubits per node to evaluate the performance of QuComm on DQC systems with limited or abundant communication resources. Since our work focuses on communication optimization and only concerns the number of inter-node communication, we do not make any assumption about the logical qubit topology inside each node. We would consider more DQC architecture options in Section 4.4.3 to evaluate the benefit of QuComm.

Benchmark programs. The fault-tolerant benchmark programs used in the evaluation

Name	# logical qubit	# remote logical CX	# logical comm by QuComm (full)		
			1 comm logical qubit/node	3 comm logical qubit/node	5 comm logical qubit/node
XOR	100	16	3	3	3
	200	48	28	14	10
	300	128	33	33	33
RCA	100	22	4	4	4
	200	44	10	8	8
	300	77	14	14	14
XORR	100	-	2	2	2
	200	-	18	4	4
	300	-	7	7	7
QFT	100	6400	81	80	80
	200	32000	562	480	480
	300	78400	1001	780	780
Grover	100	32000	6000	6000	6000
	200	96000	56000	28000	20000
	300	256000	66000	66000	66000

Table 4.2: Fault-tolerant benchmarks and results by QuComm.

are obtained from [51] and summarized in Table 4.2. Programs in Table 4.2 include the quantum XOR gate, the quantum ripple carry adder (RCA), the quantum Fourier transformation (QFT) algorithm, and Grover's algorithm. Those programs For Grover, we consider the secret string with all ones and repeat the iteration by 1000 times. All programs are decomposed into the Clifford+T basis [1], except for the raw XOR gate (XORR), which is specifically decomposed toward the DQC architecture according to [73]. We would further consider near-term applications in Section 4.4.3 to evaluate the impact of QuComm on the NISQ (Noisy Intermediate Scale Quantum) era [4].

Baseline. As the baseline, we implement the DQC compiler AutoComm [9]. AutoComm

groups a series of remote CX gates between one qubit and one node into a *burst* communication block and implements the burst communication block (all remote CX gates in it) with at most two invocations of communication protocols. AutoComm represents the state-of-the-art effort in optimizing quantum communication overhead in distributing quantum programs, as far as we know. However, without the communication buffer, AutoComm cannot optimize more general collective communication that may involve more than two compute nodes. We adopt the same circuit partition algorithm—OEE algorithm [71], which maximally reduces inter-node gates induced by partition, for both QuComm and AutoComm, in order to eliminate the difference caused by the circuit partition.

Metric. We use the number of invocations of inter-node communication protocols (Cat-Comm or TP-Comm), i.e., the amount of inter-node communication based on logical qubits, to characterize the communication overhead of the compiled distributed quantum circuits. The amount of inter-node communication in a quantum program is equivalent to the number of EPR pairs (on logical qubits) required to execute the program on DQC hardware.

Notations. Before diving into results, we first introduce some notations and abbreviations. The communication reduction by QuComm refers to '1- # comm by QuComm/# comm by baseline', with 'comm' meaning communication. '# comm lqb/node' means the number of logical communication qubits per node. For simplicity, we use **L1**, **L2**, and **L3** to denote QuComm's stages: **communication fusion**, **communication routing**, and **communication buffer design**.

Program	Comm reduction by QuComm L1			Comm reduction by QuComm L1+L2			Comm reduction by QuComm L1+L2+L3		
	1 comm lqb/node	3 comm lqb/node	5 comm lqb/node	1 comm lqb/node	3 comm lqb/node	5 comm lqb/node	1 comm lqb/node	3 comm lqb/node	5 comm lqb/node
XOR-100	46.2%	66.7%	66.7%	46.2%	75.0%	75.0%	76.9%	75.0%	75.0%
XOR-200	26.2%	58.8%	70.6%	26.2%	58.8%	70.6%	33.3%	58.8%	70.6%
XOR-300	3.7%	41.8%	54.1%	3.7%	43.9%	56.1%	69.2%	68.4%	68.4%
RCA-100	0.0%	0.0%	0.0%	0.0%	50.0%	50.0%	80.0%	50.0%	50.0%
RCA-200	0.0%	0.0%	0.0%	0.0%	50.0%	50.0%	75.0%	50.0%	50.0%
RCA-300	0.0%	0.0%	0.0%	0.0%	50.0%	50.0%	80.0%	50.0%	50.0%
XORR-100	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	60.0%	0.0%	0.0%
XORR-200	0.0%	0.0%	0.0%	0.0%	50.0%	20.0%	28.0%	50.0%	20.0%
XORR-300	0.0%	0.0%	0.0%	0.0%	74.1%	61.1%	84.4%	74.1%	61.1%
QFT-100	0.0%	0.0%	0.0%	0.0%	20.0%	20.0%	42.1%	20.0%	20.0%
QFT-200	0.0%	0.0%	0.0%	0.0%	29.4%	29.4%	43.8%	29.4%	29.4%
QFT-300	0.0%	0.0%	0.0%	0.0%	61.0%	61.0%	59.6%	61.0%	61.0%
Grover-100	46.2%	66.7%	66.7%	46.2%	75.0%	75.0%	76.9%	75.0%	75.0%
Grover-200	26.2%	58.8%	70.6%	26.2%	58.8%	70.6%	33.3%	58.8%	70.6%
Grover-300	3.7%	41.8%	54.1%	3.7%	43.9%	56.1%	69.2%	68.4%	68.4%

Table 4.3: Communication reduction by QuComm compared to AutoComm [9].

4.4.2 Compared to Baseline

In this section, we analyze the relative communication reduction of QuComm compared to the baseline and discuss the effect of designs in QuComm. Table 4.2 and 4.3 summarize results of QuComm.

Overall, QuComm significantly reduces the amount of inter-node communication across all benchmarks and device configurations tested, compared to the baseline. QuComm on average reduces the amount of inter-node communication by **60.8%**, **52.6%**, and **51.3%** on configurations of 1, 3, and 5 communication qubits per node, respectively.

The effect of program patterns and L1 optimization. QuComm behaves differently on programs of distinguished patterns. Collective communication in programs can be classified according to the extent of qubit correlation. For strongly correlated distributed quantum programs, e.g., XOR and Grover in Table 4.2, each collective communication block has an inter-node interaction graph (on logical qubits) *close* to the complete graph.

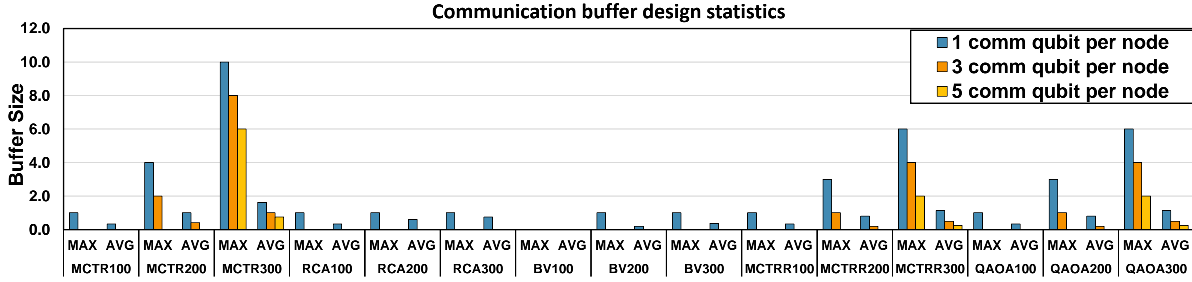


Figure 4.10: The communication buffer design results of QuComm. 'MAX', 'AVG', and 'TOT' denote the maximum size, average size, and total size of communication buffers in the DQC system.

For example, for a decomposed three-qubit XOR gate (i.e., the Toffoli gate) distributed over three nodes, its inter-node interaction graph is a triangle since there are *inter-node gates* on each two of the three involved logical qubits. In contrast, for loosely correlated distributed quantum programs e.g., RCA and QFT in Table 4.2, the interaction graph of each collective communication block has few edges compared to the vertex count.

Strongly correlated distributed quantum programs would gain more benefits from the L1 pass of QuComm. With abundant communication resources (e.g., 3 or 5 communication qubits per node), the communication reduction by 'QuComm L1' on strongly-correlated distributed programs is on average **59.8%** higher than on loosely-correlated distributed programs. The reason for the discrepancy is that strongly correlated collective communication benefits more from the implementation by aggregating qubits to the same node. For loosely correlated distributed programs, the communication block discovered by QuComm L1 is almost similar to the qubit-to-node burst communication [9] and cannot benefit from the L1 stage. Overall, L1 offers QuComm significant communication reduction of 48.3% (averaged over various program size and # comm lqb/node) on strongly correlated distributed programs (which have abundant collective communication), compared to the baseline which can only handle burst communication.

The effect of L2 optimization. The L2 optimization also provides a great reduction in

inter-node communication, especially on loosely correlated distributed programs. With abundant communication resources (3 or 5 logical communication qubits per node), the communication reduction for loosely correlated distributed programs by L2 is on average **43.4%**, far surpassing **3.5%** for strongly correlated distributed programs. On the other hand, compared the baseline which lacks communication-aware routing, L2 offers QuComm on average 28.9% communication reduction on loosely correlated programs, despite the communication fusion of QuComm on loosely correlated programs is similar to the baseline.

Results in Table 4.3 validate the designs of L2 by comparing 'QuComm L1+L2' to 'QuComm L1'. QuComm L2 moves logical qubits to the next inter-node communication position by examining current and future communication patterns. For RCA, this prevents the frequent TP-Comm back and forth caused by the baseline, leading to a **50.0%** communication reduction in the RCA benchmark (see Table 4.3 Column 6-7).

Moreover, for QFT where q_i controls all q_j $(j > i)$, QuComm L2 would utilize existing read-only copies of the control line to shorten the qubit transfer path while the baseline does not have such a capability. This optimization on average leads to a **61.0%** communication reduction on the 300-qubit QFT program (see Table 4.3 Column 6-7). Further, XORR evaluates the effect of early execution in L2. The early execution strategy leads to a **34.2%** communication reduction in the XORR benchmark, on average (see Table 4.3 Column 6-7). Finally, for the XOR and Grover benchmark, the benefit of L2 lay in selecting the proper node for qubit aggregation. For 3 or 5 logical communication qubits per node, the topology-aware node selection in L2 on average leads to **8.3%** communication reduction on the 100-qubit XOR and Grover programs.

The effect of L3 optimization. The communication buffer is more important for DQC systems where limited communication qubits are available. With the buffer, 'QuComm L1+L2+L3' significantly reduces the communication overhead, compared to 'QuComm

L1+L2' (by 21.5% on average) and AutoComm (by 54.9% on average) over all device configurations and programs tested, as shown in Table 4.3. This is because the routing and implementation of both burst and collective communication will be severely affected when lacking communication (buffer) qubits. Further, we show the size of the designed communication buffer in Figure 4.10.

As shown in Table 4.3, the importance of the communication buffer increases as the number of communication qubits decreases. Compared to 'QuComm L1+L2', 'QuComm L1+L2+L3' further reduces the amount of inter-node communication by **1.6%**, **3.3%** and **50.6%** on average, for the configurations of 5, 3, and 1 communication qubits per node. For the DQC system with only one logical communication qubit per node, the communication buffer not only facilitates the direct execution of collective communication blocks but also provides resources to relay quantum data transfer. Further, from Figure 4.10, we can draw two observations: a) the size of the communication buffer grows as the program size increases; b) strongly correlated distributed programs require more help from the communication buffer than loosely correlated distributed programs.

Finally, we claim that our communication design will not hurt the scalability. Firstly, our design features an input-adaptive buffering module which is configured to favor using idle data qubits on each node, instead of requesting extra nodes. Secondly, as shown in Figure 4.10, a small buffer that uses less $< 2\%$ data qubits would generally suffice for reducing communication at a large scale. Lastly, even without the communication buffer, 'QuComm L1+L2' still greatly reduces the communication overhead compared to the baseline, as shown in Table 4.3.

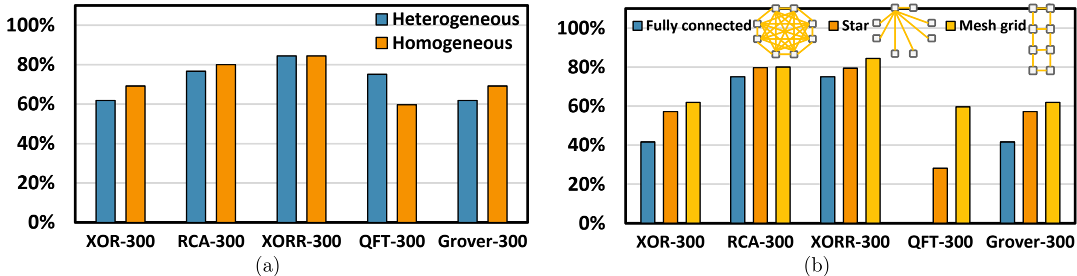


Figure 4.11: Communication reduction by QuComm on heterogeneous DQC systems, compared to the baseline [9]. (a) The effect of heterogeneous node size. (b) The effect of diverse node connectivity.

4.4.3 Additional Studies

In this section, we further evaluate the performance of QuComm on heterogeneous DQC architectures and discuss the impact of QuCommon the NISQ era.

The effect of heterogeneity for QuComm. In Figure 4.11(a), we consider a heterogeneous DQC system consisting of eight compute nodes, with 20, 20, 30, 30, 50, 50, 60, and 60 logical qubits, respectively. For comparison, the homogeneous DQC comes with eight nodes but all with 40 logical qubits. We assume one logical communication qubit per node for both heterogeneous and homogeneous DQC systems. As shown in Figure 4.11(a), compared to the baseline, QuComm can significantly reduce inter-node communication in both heterogeneous and homogeneous DQC systems. This is because collective communication is widely available for distributed programs mapped to DQC systems and the node heterogeneity may not hurt collective communication optimization.

We further evaluate the performance of QuComm on diverse DQC networks where the connectivity of nodes may be heterogeneous. We consider the fully connected and star-like network topology in addition to the mesh-grid topology. For all network topologies, we assume 8 compute nodes, 40 logical qubits per node, and 1 logical communication qubit per node. As shown in Figure 4.11(b), compared to the baseline, QuComm achieves significant communication reduction on all considered DQC architectures. We

can see that, the more sparse a DQC network is, the more benefits QuComm can provide. QuComm achieves the largest communication reduction on the mesh grid topology whose average node-to-node distance is 2.0, longer than 1.86 and 1.0 for the star-like and fully connected topology, respectively.

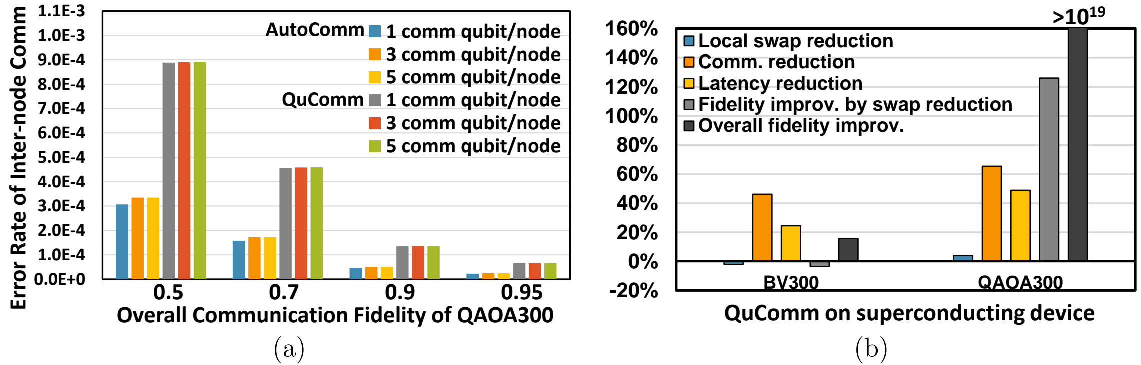


Figure 4.12: (a) The effect of nonlocal communication on the near-term application. (b) The effect of local SWAP on DQC nodes with IBM architecture [28]. Results are by comparing QuComm to the baseline [9].

The effect of QuComm on NISQ. We further evaluate QuComm on near-term applications and devices. In the evaluation, we assume physical qubits for data qubits of compute nodes. Inter-node communication protocols are directly executed on physical qubits. Near-term programs are decomposed into the CX+U3 basis [28]. For both Figure 4.12(a)(b), we assume the mesh grid network topology on 8 compute nodes, 40 data qubits per node, and 1 communication qubit per node.

Figure 4.12(a) shows the required error rate of inter-node communication to achieve specified overall communication fidelity for the 300-qubit QAOA (Quantum Approximate Optimization Algorithm) program. The result in the figure indicates that, ensuring the same level of overall communication fidelity, QuComm can admit on average 174.1% higher inter-node communication error rate, compared to the baseline. Further, with QuComm communication buffer design, we can equip each DQC node with fewer communication qubits. Thus, with QuComm, it may be possible to demonstrate DQC in the

near term.

Figure 4.12(b) demonstrates the result of QuComm on the IBM heavy hexagon architecture [28], compared to the baseline. Gate latency and fidelity are derived from [9]. Other experiment settings follow the one in Figure 4.12(a). For test programs, we consider the 300-qubit BV (Bernstein–Vazirani algorithm) and QAOA. As shown in Figure 4.12(b), QuComm does not necessarily induce more local SWAP gates than the baseline. On the one hand, the communication buffer may increase the SWAP overhead of performing CX between same-node data qubits since qubits in the buffer may hinder the SWAP path (see BV300 in Figure 4.12(b)). On the other hand, when executing inter-node CX, we need to move data qubits closer to the communication buffer. The SWAP overhead between data qubits and buffer qubits is less than that between data qubits and the communication qubit since the buffer spans the device area used for inter-node communication. This reduction of local SWAP overhead outweighs the overhead of swapping EPR pairs from the communication qubit into buffer qubits if many inter-node CX gates are executed (see QAOA300 in Figure 4.12(b)).

Furthermore, as shown in Figure 4.12(b), the local SWAP overhead change induced by QuComm is minor (<0.05%) because of the small average size of communication buffers (<2 per node as shown in Figure 4.10). QuComm always tries to use a small communication buffer since a large communication buffer may instead lead to more inter-node communication, as discussed in Section 4.3.3. This observation, on the other hand, indicates that the program latency reduction and fidelity improvement (in Figure 4.12(b)) mainly stem from inter-node communication reduction.

Comparing QuComm to more DQC compilers. We further compare QuComm to two more recent DQC compilers, called GP-CAT [45, 5] and GP-SWAP [42, 45] for simplicity. GP-CAT executes each inter-node CX gate by solely using Cat-Comm. GP-SWAP executes remote CX by swapping qubits between node and making the remote

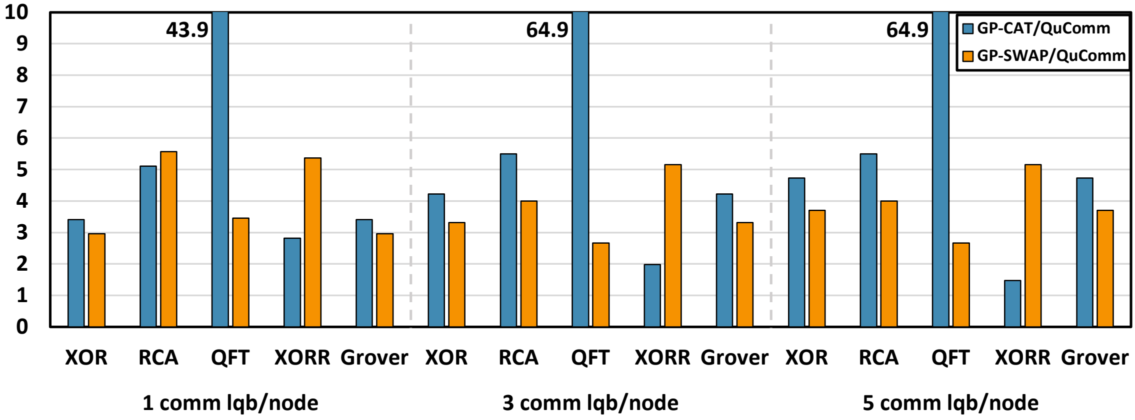


Figure 4.13: The ratio of '# comm' by GP-CAT/GP-SWAP to that by QuComm. Results are averaged over 100, 200, and 300 qubits.

CX local. The experiment setting is the same as the one for Table 4.3. As shown in Figure 4.13, compared to GP-CAT and GP-SWAP, QuComm significantly reduces inter-node communication, on average by 4.88x. Specifically, compared to GP-CAT, the communication reduction by QuComm scales with the inter-node gate count in each program. For programs with denser inter-node communication, e.g., QFT, QuComm can build larger collective communication blocks for more aggressive communication reduction. Compared to GP-SWAP, the benefit of QuComm comes from its communication-aware routing, which avoids repeated and unnecessary quantum data movement between quantum nodes by utilizing the higher-level program information provided by uncovered collective communication blocks.

Sensitivity analysis of of QuComm. Finally, we study the sensitivity of QuComm to # node and # data qubits per node, with results shown in Figure 4.14. In the figure, we consider XOR and QFT as test programs, which represents strongly-correlated and loosely-correlated distributed programs, respectively. We also consider the star DQC architecture as it has a deterministic shape when node size changes and has similar performance as the mesh-grid architecture, as shown in Figure 4.11(b). One logical com-

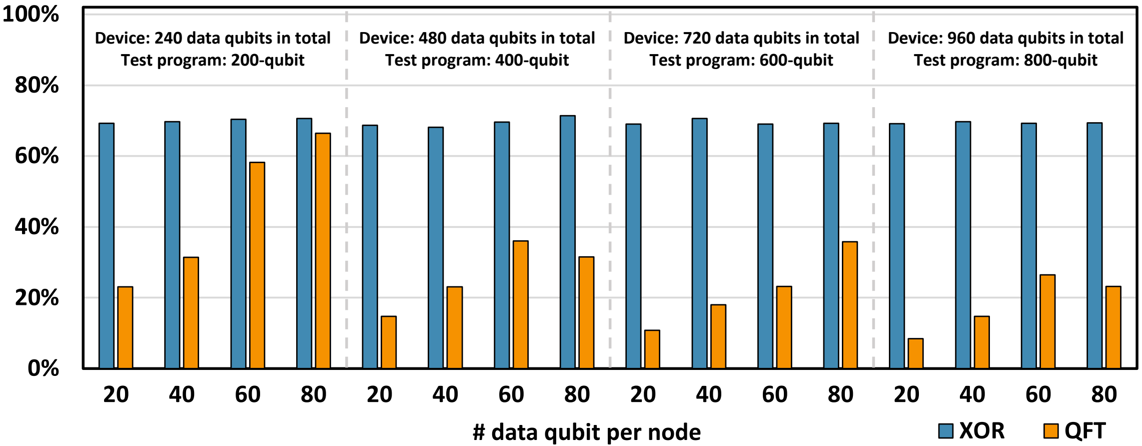


Figure 4.14: The sensivity of QuComm to '# node' and '# data qubit/node', in terms of communication reduction compared to [9].

munication qubit per node is assumed while other settings follows the one for Table 4.3.

As shown in Figure 4.14, the communication reduction by QuComm is stable for the strongly-correlated distributed program. This is because the collective communication uncovered by QuComm is consistently better than the burst communication by [9]. For example, for the decomposed CCZ gate in Figure 4.1, by our collective communication optimization, 60% fewer communication invocations are need, compared to by that burst communication optimization [9], no matter how # node and # data qubits per node change. In contrast, for loosely correlated programs, the advantage of QuComm mainly comes from its communication-aware routing. For QFT, the amount of inter-node communication increases quadratically with respect to # node, surpassing the communication reduction by QuComm's routing. Thus, as shown in Figure 4.14, the benfit of QuComm decreases as # node increases. On the other hand, by keeping # node fixed, the benefit of QuComm is stable as # data qubit per node increases, as shown in Figure 4.14. The is as expected as '# data qubit per node' does not affect QuComm's routing. Overall, for various combinations of '# node' and '# data qubit per node', QuComm is always better than [9].

4.4.4 Complexity Analysis and Impact

In this section, we will discuss the scalability of our framework, its impact on QEC requirement, as well as the trade-offs of our framework design. We then introduce potential future works based on these discussions.

Scalability Analysis of QuComm

We assume there is M multi-qubit gates, N nodes involved, E network links involved in the distributed program. We further assume K data qubits per node. For the communication fusion stage (L1) of QuComm, it performs a linear scan of the circuit as shown in Algorithm 2. Thus, the computational complexity of QuComm L1 is $O(M)$. Its space complexity is $O(M)$, so as to store the program and qubit-to-node mapping.

As for the communication routing stage (L2) of QuComm, the complexity for finding the node for aggregation and scheduling data propagation is $O(MN)$, as at most M collective communication blocks and N nodes for each specific block are involved. The shortest paths between nodes can be computed in advanced and is of complexity $O(E \log N)$. Thus, the computational complexity of QuComm L2 is $O(MN + E \log N)$. Its space complexity is $O(N^2)$ as it needs to store the shortest paths ($O(N^2)$), the distance matrix ($O(N^2)$) and the available EPR pair count per node ($O(N)$)).

Finally, as for the communication buffer design (L3) of QuComm, it first tries to find the least communication-involved data qubit q_c and the optimal relocation of this qubit for each node. The computation will keep track of the count of multi-qubit gates on each node that involves q_c, and is thus of complexity $O(M + N)$. QuComm L3 then estimates the communication reduction by relocating q_c according to Equ (4.1), and is thus of complexity $O(M)$. In the worst case, the buffer design process may go through N nodes and K data qubits for each node. Thus, the computational complexity of QuComm L3

is $O(NKM)$ in the worst case, and $O(NM)$ in the best case (the buffer design process for each node often ends in 1-2 iterations). QuComm L3's space complexity is $O(NK)$ as L3 only needs to keep track of the assignment of communication buffer qubits and remote gates for each node.

Overall, the computational complexity of QuComm is $O(M^2 + E \log N)$ in the worst case, and $O(NM + E \log N)$ in the best case. Note that $O(NK)$ is $O(M)$ as the qubit count of a program is often far fewer than its gate count. On the other hand, the total space overhead of QuComm is $O(M + N^2)$.

Impact of QuComm on QEC

Further, we study the effect of communication reduction by QuComm on the QEC requirement. As a case study, we use surface code as the underlying QEC facility, where $p_L \approx 0.03(p/p_{th})^{d/2}$. Here p_L is the error rate of the logical qubit; p_{th} is the error threshold of surface code and can be set to be 0.01; p is the physical error rate; and d is the code distance. In the following reasoning, we assume L local gates, R remote gates, and C invocations of inter-node communication protocols by QuComm for the distributed program.

First, QuComm enables smaller code distance or tolerates higher physical error rate for remote communication, while preserving the overall program fidelity. We also assume the **overall communication fidelity** we want to achieve is $1 - p_C$. Then QuComm reduces the code distance for remote communication by '$(1 - \frac{\log(p_C/(0.03C))}{\log(p_C/(0.03R))})$*100%'. For $p_C = 0.01$ and programs in Table 4.2, the code distance reduction by QuComm is on average 29.5%, up to 47.1%. On the other hand, QuComm can tolerate '$((R/C)^{2/d} - 1)$*100%' higher physical error rate for remote communication than the unoptimzed case. For $d = 5$ and programs in Table 4.2, the (tolerated) physical error rate upper bound of communication by QuComm is on average 238.6% higher, up to 1488.1%.

Second, we can also use the fidelity gain from communication reduction (by QuComm) to compensate for looser QEC for local gates. We assume that the overall program fidelity we want to achieve is $1 - p_P$. Likewise, compared to the unoptimized case, QuComm can approximately reduce the code distance required for local logical qubits by '$(1 - \frac{\log((p_P - p_C * C/R)/(0.03L))}{\log((p_P - p_C)/(0.03L))})*100\%$', or tolerate error rate of local physical qubits by '$(((p_P - p_C * C/R)/(p_P - p_C))^{2/d} - 1)*100\%$' higher. For $p_P = 0.02, p_C = 0.01$, and programs in Table 4.2, the code distance reduction by QuComm is on average 5.3%, up to 7.0%. For $p_P = 0.02, p_C = 0.01$, $d = 5$ for local logical qubits, and programs in Table 4.2, the (tolerated) error rate upper bound of local physical qubits by QuComm is on average 27.5% higher, up to 31.9%.

Finally, QuComm reduces the need of EPR pair generation by at least $R - C$. For the same overall EPR pair fidelity 0.99, the same 2-to-1 entanglement purification protocol [75] and programs in Table 4.2, QuComm can tolerate on average 419.5% (up to 3059.4%) higher error rate for EPR generation.

Chapter 5

Synthesizing an Error-Corrected Qubit

Starting from this chapter, we will explore enabling fault tolerance over noisy quantum hardware. In this chapter, our focus will be on the first problem: efficiently constructing an error-corrected 'logical' qubit by encoding redundant physical qubits of superconducting quantum hardware. The proposed framework presents the first automated solution to this problem.

5.1 Introduction

Quantum hardware has made significant progress over the past decade, with the first demonstration of *quantum supremacy* in 2020 [58]. Among various quantum hardware technologies [76, 77, 78, 79], the superconducting (SC) qubit is currently one of the most promising candidates for building quantum processors [80, 81] due to its low error rate, single qubit addressability, manufacturing scalability, etc. Many of the latest quantum computers adopt SC technology, such as IBM's 65-qubit heavy-hexagon-architecture chip [82], Rigetti's 32-qubit octagonal-architecture device [83], and Google's 54-qubit square-architecture processor [58].

The low error rate of SC quantum processors makes them ideal platforms for quantum error correction (QEC) [15, 16, 17, 18, 19, 20], thus enabling fault-tolerant (FT) quantum computation. Among various QEC codes, the surface code [20] is a popular choice due to its high tolerance of physical error rates (up to 1%). This makes surface codes one of the most viable QEC options for demonstrating near-term FT quantum computation.

With off-the-shelf surface code arrays, many recent research efforts have been devoted to improving the efficiency of FT quantum computation, ranging from compilation [84, 85], communication scheduling [86, 87], to micro-controller design [88]. All these studies are based on a nontrivial assumption: we have found a scalable way to build logical qubits with the surface code family on existing quantum devices, in particular SC devices.

However, implementing surface codes on SC devices is complex in itself as error detection relies on sophisticated circuits. Surface codes divide physical qubits into *data qubits* and *syndrome qubits*, with syndrome qubits detecting the errors of neighboring data qubits through *measurement circuits* [20]. In surface code, the implementation of measurement circuits requires a qubit structure of 2D-lattice, with each qubit coupled to four neighbors [20]. Such an architecture is not readily available on many latest SC quantum processors [82, 83]. This is because dense architectures like the 2D-lattice would lead to a high physical error rate and low yield rate [89].

Previous works attempt to address the connectivity gap between surface codes and sparsely connected SC devices either by adapting architectures with tunable couplings [79] or by designing device-dedicated QEC codes [30]. Nevertheless, the former method is expensive and may introduce additional device noise, while the latter method is not automated. A third attempt is to treat the measurement circuits of the surface code as ordinary quantum circuits and compile them to sparse SC architectures with existing quantum compilers [90, 91, 92, 93, 94, 95, 96, 97]. Unfortunately, generic compilers are not suitable for compiling surface codes. Firstly, they don't distinguish data qubits

from other qubits. Those compilers may move data qubits frequently, making it hard to apply logical operations which assume a fixed data qubit layout [20]. Secondly, the SWAP gates they use to overcome the connectivity gap make the compiled measurement circuits more error-prone, compared to specialized measurement circuits [30, 31] which do not use SWAP gates. Finally, they only focus on gate-level optimization and do not account for the parallelism between measurement circuits enabled by a specific execution order [20].

To address problems of existing methods, we propose the first automatic synthesis framework *Surf-Stitch* which specializes in stitching the surface code family to various SC quantum devices. With specialized measurement circuits [31, 30] as the backend, our framework overcomes three key challenges of the surface code synthesis, which remain unexplored by existing works. The first is *the allocation of data qubits.* If the data qubits of a measurement circuit are far apart from each other, we would need many ancillary qubits to help detect their errors. Conversely, if they are too close, there will not be enough room for the syndrome qubit. The second is *the construction of measurement circuits.* Measurement circuits should be small as large circuits are error-prone and hurt the error detection accuracy of the surface code. Besides, large measurement circuits may contend for ancillary qubits. Such resource conflicts would destroy the parallelism of error detection. The third is *the execution order of measurement circuits.* We should exploit parallelism between measurement circuits as much as possible, to shorten the error detection cycle and reduce the decoherence error.

Our framework decouples the solution space of the identified key challenges with a modular optimization scheme that includes three stages. Firstly, we optimize the allocation of data qubits as they are the key to gluing measurement circuits together. We search for data qubits over rectangular device blocks since the measurement circuit is exactly shaped by a rectangle [20]. We require each rectangular block to be the smallest

possible, for a compact data qubit layout and potential small measurement circuits. Secondly, we optimize measurement circuits for the allocated data qubits. The goal is to keep them small and minimize the conflict between them if possible. To achieve the goals, we constrict each measurement circuit within rectangular blocks of zero overlapping areas and then adopt two heuristics to find small circuits. Finally, we optimize the execution order of measurement circuits as the conflict between them is sometimes inevitable. We observe that the error detection cycle can be reduced by executing large measurement circuits together. Therefore we propose a procedure to find and execute large circuits that do not have resource conflicts in parallel.

We evaluate the proposed synthesis framework by comparing it with manually-designed QEC codes [30]. The results show that the surface codes synthesized by our framework can achieve equivalent or even better error correction capability. This result is inspiring as it unveils the possibility that automated synthesis can surpass manual QEC code design by experienced theorists. We also investigate our framework on various mainstream SC quantum architectures to demonstrate its wide applicability. Surf-Stitch would be of great interest to both QEC researchers and quantum hardware designers. Theorists will have a baseline to compare with when designing novel QEC codes. Hardware researchers can identify inefficient architecture designs for the surface code with Surf-Stitch.

Our contributions are summarized as follows:

- We systematically formulate the surface code synthesis problem on SC quantum devices for the first time and identify three key challenges: data qubit allocation, measurement circuit construction, and syndrome measurement schedule.
- We propose the first automatic synthesis framework that addresses the identified challenges step by step, with insights extracted from surface codes and SC quantum architectures.

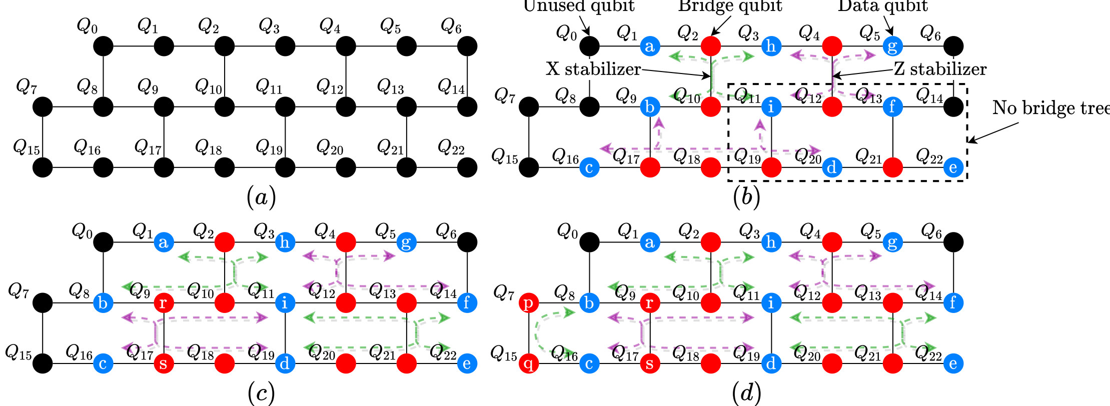


Figure 5.1: A motivating example for synthesizing a (rotated) distance-3 surface code. (a) An SC device based on the hexagon structure. (b) A bad data qubit layout where the stabilizer X_{idfe} cannot be measured. (c) A promising data qubit layout that ensures all stabilizer measurements. (d) An example of resolving the bridge tree conflict.

- Our evaluation demonstrates the effectiveness of the proposed framework by the comparison to manually designed QEC codes and a comprehensive investigation of Surf-Stitch on various mainstream SC quantum architectures.

5.2 Problem Formulation

In this section, we first formulate the problem of synthesizing surface codes onto SC quantum processors and then introduce the optimization opportunities.

We consider implementing the (rotated) surface code in Figure 2.4(b) on a quantum device with the hexagon architecture (Figure 5.1(a)) [30]. In this hexagon device, each qubit connects to at most three other qubits. This imposes a challenge to synthesizing stabilizer measurement circuits of the surface code since a syndrome qubit in either an X- or Z- type stabilizer measurement should connect to four data qubits (see Figure 2.3(b)(c)). As in Figure 2.5, we can overcome the connectivity limitation of this SC device with specialized measurement circuits [30, 31] as long as the data qubits and the bridge tree for each stabilizer measurement are determined. While deciding the

data qubits and the bridge tree for one stabilizer measurement is easy, deciding them for all stabilizer measurements and making the implemented measurement circuits work together are challenging tasks in the overall surface code synthesis.

In this section, we formulate the surface code synthesis problem into three key stages: data qubit allocation, measurement circuit construction, and stabilizer measurement schedule. Since the measurement circuit is determined once the bridge tree is selected, we will refer to the second stage as bridge tree construction.

We briefly introduce the objectives and the design considerations of each stage as follows.

Data qubit allocation: We choose to allocate and fix the position of data qubits first as data qubits are the key to gluing stabilizer measurement circuits together. Once allocated, the location of data qubits should not be changed, otherwise, the logical operations designed for a fixed data qubit layout [20] would be invalidated. The layout of data qubits affects the execution of stabilizer measurements. As an example, we synthesize the distance-3 surface code in Figure 2.4(b) with two data qubit layouts in Figure 5.1(b) and Figure 5.1(c). In Figure 5.1(b), the stabilizer X_{idfe} cannot be measured without moving data qubits and inserting SWAP gates, which are not allowed to avoid error proliferation. In contrast, all stabilizer measurements $(X_{abhi}, X_{idfe}, X_{fg}, X_{bc}, Z_{bcid}, Z_{higf}, Z_{ah}, Z_{de})$ can be executed on Figure 5.1(c) by using the depicted bridge trees.

Bridge tree construction: After the data qubits are placed, the next step is to select bridge qubits and construct bridge trees for stabilizer measurements. The first constraint in this stage is that we should minimize the number of bridge qubits for each stabilizer measurement since using more physical qubits would result in larger measurement circuits which are naturally more error-prone. Besides, the construction of bridge trees affects the efficiency of error detection because two stabilizers can be simultaneously measured only if their bridge trees do not share qubits (i.e., no resource conflict). For instance, referring

to Figure 5.1(c), if we measure X_{bc} with bridge qubits $\{r,\ s\}$, these two qubits then cannot be used in the measurement circuit of X_{abhi} at the same time because the bridge qubits need to be reset at the beginning of any measurement circuit. However, if we measure X_{bc} with bridge qubits $\{p,\ q\}$ in Figure 5.1(d), we can measure X_{bc} and X_{abhi} in parallel. An efficient bridge tree construction should enable the concurrent measurement of as many stabilizers as possible.

Stabilizer measurement scheduling: The third stage is to schedule the execution of stabilizer measurements. It would be desirable to execute stabilizer measurements in parallel as much as possible since it can reduce the error detection latency and mitigate the decoherence error. However, stabilizer measurement circuits with overlapped bridge qubits cannot be executed simultaneously. For example in Figure 5.1(c), the measurement circuit of X_{abhi} and Z_{bcid} cannot be measured together since they share bridge qubits $\{q_9, q_{10}\}$. One possibility is to measure X_{abhi} and X_{idhe} first, then measures Z_{hgif} and Z_{bcid}. This schedule may seem promising, but it is not optimal as these two groups of stabilizer measurements take 20 operation steps in total, using the flag-bridge circuit [31](see Figure 2.5) as the backend. As a comparison, if we measure X_{abhi} and Z_{hgif} first and measure X_{idfe} and Z_{bcid} second, the total number of operation steps is only 18. Our objective is to identify the potential parallelism in stabilizer measurements and generate efficient scheduling to shorten the overall error detection latency.

5.3 Synthesis Algorithm Design

In this section, we introduce the surface code synthesis flow of Surf-Stitch. As discussed above, we will introduce three key stages: data qubit allocation, bridge tree construction and stabilizer measurement scheduling.

5.3.1 Data qubit allocator

We start by allocating data qubits. Once allocated, the positions of data qubits should not be changed. This is because the logical operations on surface codes [20] assume a fixed data qubit layout, and moving data qubits would invalidate those high-level operations. Moreover, moving data qubits would involve a series of SWAP gates which are noisy and could destroy the logical information stored in data qubits.

A fundamental requirement for data qubit allocation is to ensure that a bridge tree exists for each stabilizer. The following proposition about the degree of nodes (qubits) provides a necessary condition to guarantee this.

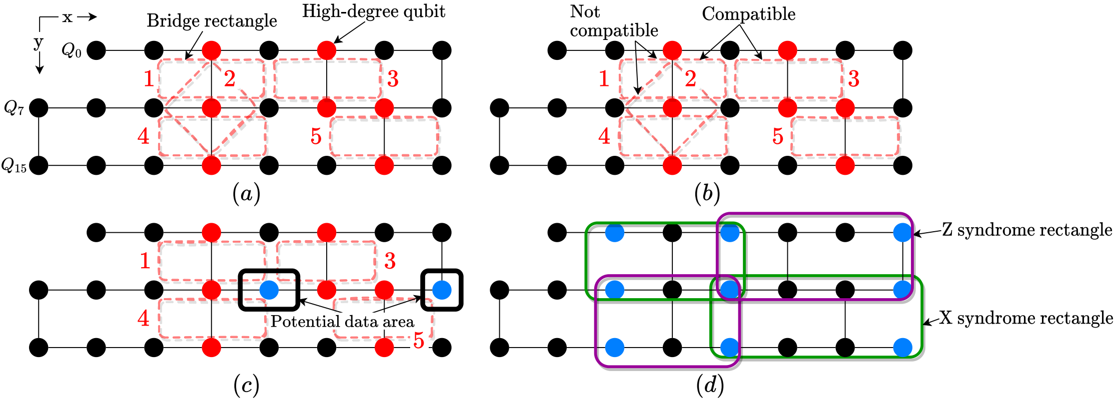


Figure 5.2: Data qubit allocation example. (a) A modified device from Figure 5.1(a). Red circles indicate physical qubits with a high degree of connectivity (i.e. with 3 or more edges). (b) Finding compatible bridge rectangles. (c) Locating data qubits. (d) The final data qubit layout and syndrome rectangles.

Proposition 5.3.1. *Any bridge tree for a stabilizer with support on four data qubits must have at least one four-degree node or two three-degree nodes.*

Proof. For any graph $G(V, E)$ where V is the vertex set and E is the edge set, we have $\sum_{v \in V} deg(v) = 2|E|$. An n-vertex tree always has $n - 1$ edges. A bridge tree with four data qubits has four 1-degree leaf nodes and all other nodes should have degrees of at least 2. Therefore we have $4 + \sum_{v \in V \backslash \text{data qubits}} deg(v) = 2n - 2$ and $\sum_{v \in V \backslash \text{data qubits}} deg(v) =$

Algorithm 3: Data qubit allocation

```
   Input: Device architecture graph G.
   Output: Data qubit layout data_layout.
 1 L_h = all three- and four-degree nodes in G;
 2 bridge_rects = [] ; // the set of bridge rectangles
 3 for n_a in L_h do
 4 |   if deg(n_a) == 3 then
 5 |   |   n_b = the nearest high-degree node of n_a;
 6 |   |   rect = the minimal rectangle containing n_a, n_b and their neighboring qubits;
 7 |   else
 8 |   |   rect = the minimal rectangle containing n_a and its neighboring qubits;
 9 |   end
10 |   bridge_rects.append(rect);
11 end
12 r_0 = the bridge rectangle at the top left corner of G;
13 bridge_rect_tuple = []; // compatible bridge rects;
14 repeat
15 |   for (r_1, r_2, r_3) ∈ ⊗^3 bridge_rects do
16 |   |   if r_0, r_1, r_2, r_3 are mutually compatible then
17 |   |   |   potent_dqbits = qubits enclosed by r_0, r_1, r_2, r_3; // potential data area;
18 |   |   |   if potent_dqbits ≠ ∅ then
19 |   |   |   |   bridge_rect_tuple.append((r_0, r_1, r_2, r_3));
20 |   |   |   |   break;
21 |   end
22 |   set r_0 to r_1, r_2, r_3 in turn to find new tuples of r_0, r_1, r_2, r_3 that has non-empty
   |     potent_dqbits;
23 until bridge_rect_tuple converges;
24 data_layout = [];
25 for r_0, r_1, r_2, r_3 in bridge_rect_tuple do
26 |   dqb = the qubit at the center of potent_dqbits of r_0, r_1, r_2, r_3;
27 |   data_layout.append(dqb);
28 end
```

$2n-6 = 2(n-4)+2$. We only have $n-4$ vertices after removing the four leaf nodes. So we must have at least one four-degree node or two three-degree nodes. □

From Proposition 5.3.1, we see that a feasible layout should ensure that each data qubit has enough three-degree or four-degree qubits around it so that it can form a stabilizer with nearby data qubits. To achieve that, we introduce a data qubit layout that ensures the existence of *local bridge trees*, whose bridge qubits lie within the region

bounded by the corresponding stabilizer's data qubits. The benefit of this strategy comes from the fact that local bridge trees often lead to shallow measurement circuits. For illustration, we adopt the SC quantum architecture shown in Figure 5.2(a). We embed the coupling graph of that architecture into a 2D grid so that all qubits can be referred to by the spatial coordinates on the grid. Such an embedding is always possible for the latest SC processors as they are usually designed in a modular structure.

Now we state the data qubit allocation algorithm, as shown in Algorithm 3. We keep a list (denoted as L_h) for all the three- and four-degree nodes in the grid and record their coordinates since high-degree nodes are critical to constructing the data qubit layout. In Figure 5.2(a), $L_h = \{Q_2, Q_4, Q_{10}, Q_{12}, Q_{13}, Q_{18}, Q_{21}\}$. Then we process the nodes in the list sequentially. For each node n_a in L_h, if it is a three-degree node, we search for its nearest high-degree node n_b and create a minimal rectangle containing n_a, n_b, and their neighbors. If n_a is of degree ≥ 4, then we create a rectangle containing n_a and its neighbors. Such a rectangle is called a *bridge rectangle*. Figure 5.2(a) depicts five bridge rectangles resulted from $\{Q_2, Q_{10}\}$, $\{Q_{10}\}$, $\{Q_4, Q_{12}\}$, $\{Q_{13}, Q_{21}\}$ and $\{Q_{18}, Q_{10}\}$, indexed from 1 to 5. We omit other bridge rectangles here for simplicity.

Based on those bridge rectangles, we can then determine the positions of data qubits. As shown in Figure 2.3(a), each data qubit of the surface code should be shared by four stabilizers. Likewise, we can determine the position of a data qubit by four bridge rectangles. We search for *compatible* bridge rectangles starting from rectangle 1. (We can also start from rectangle 2 which is created from a four-degree qubit. We will discuss this possibility in Section 5.4.) Two bridge rectangles are said to be compatible if their overlapping area is zero. For example in Figure 5.2(b), rectangle 2 is not compatible with rectangle 1 and rectangle 4, while rectangles 1, 3, 4, and 5 are mutually compatible. We avoid using incompatible rectangles as they may not allow a feasible data qubit layout. When four compatible bridge rectangles are found, we search for the data qubit in the

potential data area (the black rectangle in the center of Figure 5.2(c)), which is enclosed by those compatible bridge rectangles, as shown in Figure 5.2(c). If the potential data area is empty, we select another four compatible bridge rectangles. Otherwise, we select the qubit at the center of the potential data area as a data qubit.

On the boundary, we may not have enough bridge rectangles to locate the data qubits. For example, the bottom right corner of rectangle 3 is only neighbored by rectangle 5. In this case, we have to locate the data qubit based on only those two bridge rectangles. Specifically, a potential data qubit should satisfy: A) its x coordinate $\geq$ the largest x coordinate in rectangles 3 and 5; B) its y coordinate should lie between the largest y coordinate of rectangle 3 and the smallest y coordinate of rectangle 5. With these spatial constraints, the only qubit we can find is Q_{14}, as shown in the black rectangle on the right of Figure 5.2(c). Positions of other data qubits are determined in a similar way.

The final layout of data qubits and their associated syndrome rectangles are shown in Figure 5.2(d). A syndrome rectangle is an extension of the bridge rectangle which includes the allocated data qubits. We can assign a stabilizer to each syndrome rectangle and synthesize the corresponding measurement circuits locally (using qubits inside each syndrome rectangle). In the next section, we will discuss how to find a short bridge tree for each stabilizer.

5.3.2 Bridge tree finder

Based on allocated data qubits and syndrome rectangles, we search for bridge trees that satisfy two requirements: small in size and local in position. The reason why they are required to be small is that large bridge trees would compromise the fidelity of stabilizer measurements. This is because the error correction capability of the synthesized surface codes is sensitive to the length of bridge trees, as each additional edge in a bridge tree

Algorithm 4: Bridge tree construction

Input: A syndrome rectangle R with data qubits $\{a, b, c, d\}$.
Output: Candidate bridge trees.

```
// bridge trees by the star-tree method;
```
1 $star_trees = []$;
```
// bridge trees by the branching-tree method;
```
2 $branching_trees = []$;
3 **for** qb *in* R **do**
4 $\quad T$ = the bridge tree by connecting qubit qb to data qubits $\{a, b, c, d\}$ with shortest paths;
5 $\quad$ insert T to $star_trees$ and remove trees larger than T from $star_trees$;
6 **end**
7 let $\{a', b', c', d'\}$ be an arrangement of $\{a, b, c, d\}$ s.t. $l_{a'b'} + l_{c'd'} = \min\{l_{ab} + l_{cd}, l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$;
```
// l_ab is the distance of a → b;
```
8 connect a' and b', c' and d' with shortest paths;
9 **for** qb_1 *in* $a' \rightarrow b'$, qb_2 *in* $c' \rightarrow d'$ **do**
10 $\quad T$ = the resulting bridge tree by connecting qb_1 and qb_2 with shortest paths;
11 $\quad$ insert T to $branching_trees$ and remove trees larger than T from $branching_trees$;
12 **end**
13 merge $star_trees$ and $branching_trees$ to find a list of small local bridge trees;

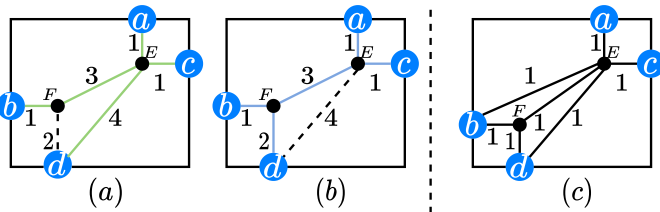


Figure 5.3: Finding bridge trees in a syndrome rectangle with data qubits $\{a, b, c, d\}$. (a)(b) shows the case where path merge is efficient, while (c) shows the case when path merging incurs extra overhead. (a) Green edges denote the shortest paths from qubit E to data qubits and they form a bridge tree with length 10. (b) Blue edges form a bridge tree with length 8. (c) An example where data qubits are close to each other.

results in two more CNOT gates in the measurement circuit, increasing the probability of correlated errors which are hard to detect and correct [20]. The reason why bridge trees should be local is to guarantee the parallelism of bridge trees, which also affects the fidelity of stabilizer measurements. For bridge trees that share bridge qubits, i.e., incompatible bridge trees, their corresponding stabilizers must be measured sequentially,

resulting in a longer error detection cycle, which means more decoherence errors. To reduce potential conflicts between bridge trees, we only search for bridge trees inside each syndrome rectangle. Such *local bridge trees*, whose qubits lie completely within the syndrome rectangles, naturally facilitate the concurrent measurement of stabilizers.

A natural way to find small local bridge trees is to first locate the bridge tree root within the syndrome rectangle, and then connect the tree root to data qubits by the shortest paths. We denote this method as the *star-tree method*. A disadvantage of this method is that it may miss opportunities for path merging. For example, in the syndrome rectangle in Figure 5.3(a), the length of the bridge tree produced by the star-tree method is 10 (green edges). In contrast, by merging paths $E \to F \to b$ and $E \to d$, we can get a bridge tree of length 8 (blue edges in Figure 5.3(b)), which reduces the number of CNOT gates in the resulting stabilizer measurement circuit by at least 4.

To remedy the above shortcoming, we propose the *branching-tree method*, which first connects close data qubit pairs by shortest paths, and then connects those shortest paths to build a complete bridge tree. As an example, suppose we are constructing a bridge tree for the syndrome rectangle in Figure 5.3(a). We first find the shortest paths $a \to c$ and $b \to d$, since $l_{ac} + l_{bd}$ (l_{ac} is the length of the shortest path from a to c) is smaller than $l_{ab} + l_{cd}$ and $l_{ad} + l_{bc}$. Then by connecting paths $a \to c$ and $b \to d$ with path $E \to F$, we immediately obtain the small bridge tree (blue edges) in Figure 5.3(b). The following proposition bounds the length of the bridge tree generated by the branching-tree method:

Proposition 5.3.2. *Let the total edge length of the bridge tree T generated by the branching-tree method be $E(T)$, then,*

$$E(T) \leq \frac{1}{2}(l_{ab} + l_{ac} + l_{ad} + l_{bc} + l_{bd} + l_{cd}).$$

Proof. W.l.o.g., we assume $l_{ab} + l_{cd} \leq \min\{l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$. Then in T, we first connect

a and b, c and d, respectively. On the other hand, the distance between shortest paths $a \to b$ and $c \to d$ is smaller than $\min\{l_{ac}, l_{ad}, l_{bc}, l_{bd}\}$. This proposition then can be proved by combining these two inequalities. □

Generally, the branching-tree method is more efficient if $\min\{l_{ab}+l_{cd}, l_{ac}+l_{bd}, l_{ad}+l_{bc}\}$ is small, as shown in Figure 5.3(a)(b). In this case, the length of the resulting bridge tree is very close to $\frac{1}{2}(l_{ad} + l_{bc})$. In contrast, the length of the bridge tree by the star-tree method is at least $\max\{l_{ad}, l_{bc}\} + 2$, which is larger than that by the branching-tree method. However, if $\max\{l_{ab} + l_{cd}, l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$ is small too, the benefit of path merging may not outweigh the overhead of not using shortest paths. Figure 5.3(c) shows an example where the star-tree method produces a shorter bridge tree. In practice, we will run both the star-tree method and the branching-tree method, then find small bridge trees by merging their results, as shown in Algorithm 4. Once the bridge tree is determined, we can assign the syndrome qubit to the center node of the bridge tree.

In general, our bridge tree finder can generate small bridge trees that approximate optimal bridge trees as long as the distances between data qubits are small (by Proposition 5.3.2).

5.3.3 Stabilizer measurement scheduler

With all stabilizers and their measurement circuits allocated to the physical device, the next goal is to minimize the runtime of stabilizer measurements by maximizing parallelism, which naturally reduces the effect of decoherence. Two stabilizers can be measured in parallel if and only if their measurement circuits do not share bridge qubits. Such stabilizers are said to be compatible with each other. To exploit the parallelism of compatible stabilizers while not allowing incompatible stabilizers to be measured concurrently, we propose a heuristic scheduling approach in Algorithm 5, which consists of two steps:

Algorithm 5: Iterative stabilizer measurement scheduling

```
   Input: Binary tuples of stabilizer and syndrome rectangle: {(s, R)}.
   Output: A schedule of binary tuples of stabilizer and bridge trees.
   // Schedule initialization;
1  S1 = tuples of X-stabilizers and syndrome rectangles;
2  S2 = tuples of Z-stabilizers and syndrome rectangles;
   // Iterative refinement;
3  repeat
4      if exec_time(S1) < exec_time(S2) then
5          swap(S1, S2);
6      r2 = (s, R) in S2 that has longest execution time;
7      swap_list = [r2]; for i in [0 : k] do
8          S = S_{i%2+1};
9          for r in swap_list do
10             swap_list.remove(r);
11             for r1 in S in descending order do
12                 if r1 and r does not have compatible bridge trees then
13                     if exec_time(r1) > exec_time(r) then
14                         terminate the refinement loop;
15                     swap_list.append(r1);
16                     S.remove(r1);
17             end
18             if swap_list == ∅ then
19                 break;
20         end
21     end
22     if swap_list ≠ ∅ then
23         recover S1 and S2 to the values before this iteration;
24         break;
25 until S1 converges;
26 generate the finalized stabilizer measurement schedule from S1 and S2;
```

schedule initialization and refinement loop.

Schedule initialization: The proposed data qubit allocation ensures that syndrome rectangles of the same type do not have bridge tree conflicts, i.e., the stabilizer measurements of the same type are compatible with each other. With this guarantee, we initialize the stabilizer measurement schedule with two sets, S_1 and S_2 which contain X- and Z-type stabilizers, respectively.

Refinement loop: The core idea of the refinement loop is to move stabilizers with

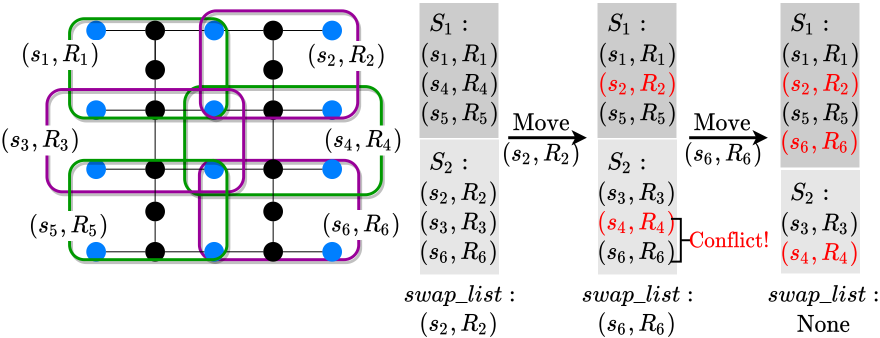


Figure 5.4: An example of stabilizer measurement scheduling.

large measurement circuits into one set. The motivation for this refinement is that the execution time of a set of stabilizer measurements is determined by the stabilizer with the deepest measurement circuit. After the refinement loop, we end up with one set containing the stabilizers with large measurement circuits, and the other set containing the remaining stabilizers which have small measurement circuits and can be measured in a short time.

To illustrate how the refinement loop works, suppose we are given stabilizers and syndrome rectangles shown in Figure 5.4. Initially, we have $S_1 = \{(s_1, R_1), (s_4, R_4), (s_5, R_5)\}$ and $S_2 = \{(s_2, R_2), (s_3, R_3), (s_6, R_6)\}$. We then send the largest element in S_2, which is (s_2, R_2) in this case, to the *swap_list* and swap it into S_1. Since (s_4, R_4) and (s_2, R_2) do not have compatible bridge trees, we will move (s_4, R_4) to S_2. In S_2, (s_6, R_6) is not compatible with (s_4, R_4), so it will be swapped into S_1. After this swap, the refinement loop will stop since the *swap_list* is empty and every stabilizer in S_1 has a larger bridge tree than the stabilizer in S_2. The finalized stabilizer measurement schedule is shown in Figure 5.4(b). Compared to the initial schedule, the refined schedule in Figure 5.4(b) reduces the error detection cycle by one time step, and reduces the CNOT gate number by two.

5.4 Evaluation

In this section, we first evaluate the proposed synthesis framework *Surf-Stitch* by comparing its generated surface codes with state-of-the-art manually designed QEC codes. We then demonstrate the effectiveness of Surf-Stitch on mainstream SC architectures by analyzing the error correction capability and resource overhead of the synthesized codes.

5.4.1 Experiment Setup

Evaluation setting: We use the flag-bridge circuit [31] as the backend for instantiating stabilizer measurement circuits as it provides the extra feature of fault-tolerant error detection [98]. We implement all numerical simulations with stim v1.5.0, a fast stabilizer circuit simulator [99]. We use PyMatching v0.4.0 [100] for error decoding with measurement signals from bridge qubits. The PyMatching decoder is the implementation of the well-studied *Minimum Weight Perfect Matching* (MWPM) algorithm [30, 20]. Error rates are computed by performing 10^5 simulations with $3d$ (d is the code distance) error detection rounds, on a Ubuntu 18.04 server with a 6-core Intel E5-2603v4 CPU and 32GB RAM.

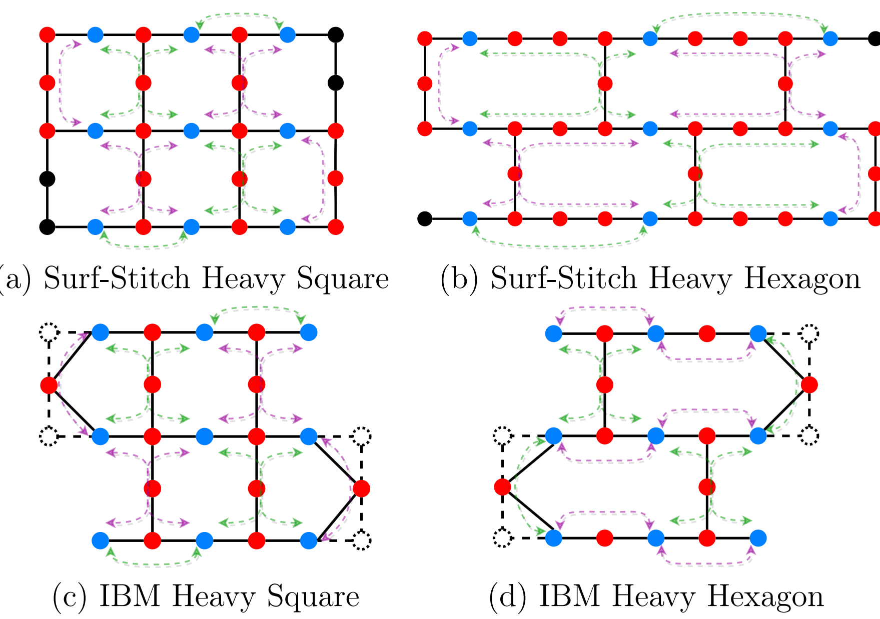
Metrics: We evaluate the *error threshold* of the synthesized surface codes to demonstrate their error correction capability. Error threshold indicates the error rate below which hardware errors can be tolerated [20]. Hence, a higher error threshold is preferred. The time-step count determines the execution speed of the surface code and its logical operations. A large time-step count would also introduce more decoherence errors. Thus, a small time-step count is preferred. Finally, We evaluate the resource overhead of the synthesized surface codes with the *CNOT count* and the *qubit count*. A resource-efficient synthesis should use fewer CNOT gates and bridge qubits.

Type	Name	Building blocks	Tilling Example	Remark
Polygon Architectures	Square			Each square can have at most four neighboring squares for tiling.
	Hexagon			Each hexagon can have at most six neighboring hexagons for tiling.
	Octagon			Each octagon can have at most four neighboring octagons for tiling.
Heavy Architectures	Heavy Square			Heavy squares are tiled like squares.
	Heavy Hexagon			Heavy hexagons are tiled like hexagons.

Table 5.1: Overview of device architectures.

Device architectures: we use two categories of device architectures, as shown in Table 5.1. The first category of architectures built from tiled polygons is the basic structure of many SC quantum devices, e.g. Google's Sycamore [58] and IBM's latest machines [101]. The second category of architectures is mainly used by IBM devices [101]. It consists of *heavy architectures* with an extra qubit inserted into each edge of the polygons. Edges with the extra qubit in the middle are called *heavy edges*. Compared to polygon architectures, the average qubit connectivity of heavy architectures is lower due to the inserted two-degree qubits. All architectures in Table 5.1 can be easily embedded into a 2D grid.

Error model: In all simulations, we assume a similar circuit-level error model as in [20, 30]. For the gate error, we assume an error probability p_e for the single-qubit depolarizing error channel on single-qubit gates, the two-qubit depolarizing error channel on two-qubit gates, and the Pauli-X error channel on measurement and reset operations. For the idle error induced by decoherence, we assume each idle qubit is followed by a single-qubit depolarizing error channel per gate duration with the error probability



(a) Surf-Stitch Heavy Square (b) Surf-Stitch Heavy Hexagon

(c) IBM Heavy Square (d) IBM Heavy Hexagon

Figure 5.5: The synthesized distance-3 surface codes by Surf-Stitch and the two manually designed QEC codes by IBM [30].

0.0002, which is estimated by the decoherence error formula $1 - e^{-\frac{t}{T}} \approx 0.0002$, with the gate duration $t = 20\,ns$ and the relaxation or dephasing time $T = 100\,\mu s$ [28]. These errors happen on all qubits, including data qubits and bridge qubits.

5.4.2 Compared to manually designed QEC codes

We first compare the synthesized surface codes by Surf-Stitch to the two manually designed QEC codes by Chamberland et al. [30] on heavy architectures. Figure 5.5(a)(b) show the qubit layouts and stabilizer measurements of our synthesized surface codes on the heavy square architecture ('Surf-Stitch Heavy Square') and the heavy hexagon architecture ('Surf-Stitch Heavy Hexagon'). Figure 5.5(c)(d) show the manually designed QEC codes on the heavy square architecture ('IBM Heavy Square') and the heavy hexagon architecture ('IBM Heavy Hexagon'). The error thresholds of these codes are in Figure 5.6. The error thresholds are computed with respect to Pauli X errors.

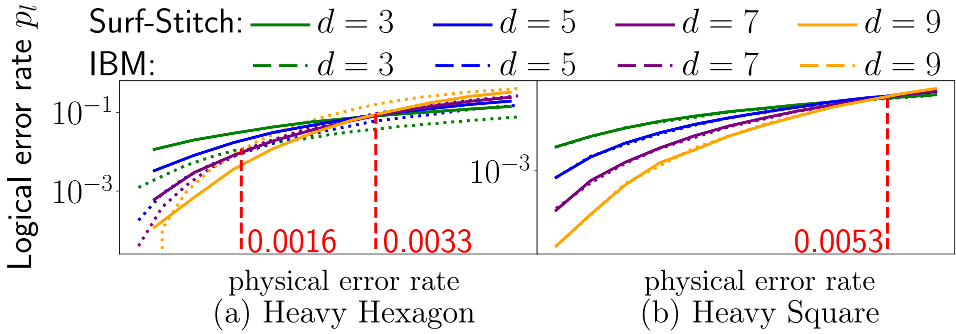


Figure 5.6: The error threshold is the physical error rate where code curves of different distances meet. (a) The error thresholds are 0.16% and 0.33% for codes by IBM and Surf-Stitch, respectively. (b) The error threshold is 0.53% for both codes.

Overall, compared with the manually designed codes on the heavy architectures, the surface codes synthesized by Surf-Stitch can have comparable or even better error correction capability. On the heavy hexagon architecture, the error threshold of 'Surf-Stitch Heavy Hexagon' is 0.33% which is 106% higher than that of 'IBM Heavy Hexagon' (0.16%), as shown in Figure 5.6(a). This significant discrepancy comes from the fact that 'IBM Heavy Hexagon' measures gauge operators instead of the stabilizers for the Pauli-X error detection. Besides, 'IBM Heavy Hexagon' does not guarantee the fault tolerance of the Pauli-X error detection procedure. On the heavy square architecture, the error threshold of 'Surf-Stitch Heavy Square' is the same as that of 'IBM Heavy Square', as shown in Figure 5.6(b). This is because the code synthesized by Surf-Stitch is almost identical to 'IBM Heavy Square' except for stabilizers on boundaries, as shown in Figure 5.5(a)(c). The only difference is that 'IBM Heavy Square' removes some boundary nodes (dotted) and edges (dotted) for better efficiency of stabilizer measurements on the borderline.

In summary, Surf-Stitch can automatically generate surface codes that have similar or even better error correction capability compared with manually designed QEC codes on the two studied architectures.

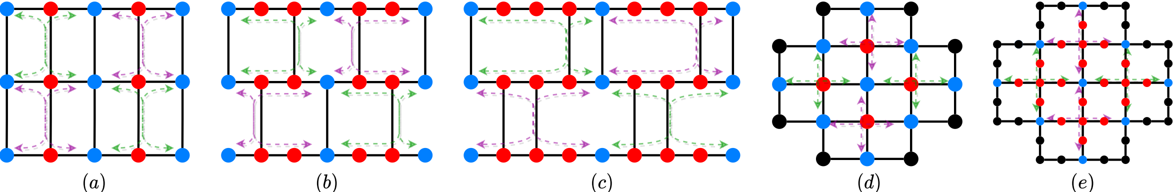


Figure 5.7: First four stabilizers of the synthesized surface codes by Surf-Stitch. (a)(b)(c) syntheses on the square, hexagon, and octagon architectures. (d)(e) syntheses on the square and heavy square architectures by using syndrome rectangles induced by 4-degree qubits.

Code	Avg. bridge qubit #	Avg. CNOT #	Avg. time-step #	Tot. time-step #	Error thresh-old
Surf-Stitch Heavy Sqaure	3	8	12	24	0.53%
Surf-Stitch Heavy Hexagon	7	19	20	40	0.33%
Surf-Stitch Sqaure	2	6	10	20	0.63%
Surf-Stitch Hexagon	4	10	13	26	0.47%
Surf-Stitch Octagon	6	14	14	28	0.38%
Surf-Stitch Sqaure-4	1	4	8	8	0.70%
Surf-Stitch Heavy Sqaure-4	5	12	13	13	0.45%
IBM Heavy Sqaure	3	8	12	24	0.53%
IBM Heavy Hexagon	3	8	12	24	0.16%

Table 5.2: Metrics of the synthesized surface codes by Surf-Stitch. The average numbers of bridge qubits, CNOT gates, and time steps are computed over all X-type stabilizers.

5.4.3 Synthesis on various SC architectures

We further apply Surf-Stitch to other architectures in Table 5.1 to demonstrate the general applicability of Surf-Stitch. Figure 5.7(a)-(c) presents the synthesis results of Surf-Stitch on the square, hexagon, and octagon architectures. Besides syntheses enabled by a pair of three-degree bridge qubits as in Figure 5.7(a)-(c), we also include another two surface codes generated by using syndrome rectangles centering around four-degree qubits, as shown in Figure 5.7(d)(e). These two codes have the suffix '-4' in the code name in Table 5.2 and Table 5.3. Table 5.2 summarizes the characteristics of stabilizer measurements in the synthesized surface codes by Surf-Stitch. Table 5.3 shows the resource requirements of these synthesized surface codes and is obtained by finding the smallest

Code	data qubit %	bridge qubit %	unused qubit %	Tot. qubit #
Surf-Stitch Heavy Sqaure	31.7%	45.6%	22.8%	79
Surf-Stitch Heavy Hexagon	18.8%	59.4%	21.8%	133
Surf-Stitch Square	55.6%	44.4%	0.0%	45
Surf-Stitch Hexagon	30.5%	48.8%	20.7%	82
Surf-Stitch Octagon	13.8%	75.8%	10.4%	116
Surf-Stitch Square-4	43.9%	56.1%	0.0%	57
Surf-Stitch Heavy Sqaure-4	16.3%	83.7%	0.0%	153
IBM Heavy Sqaure	31.7%	45.6%	22.8%	79
IBM Heavy Hexagon	17.4%	63.0%	19.6%	92

Table 5.3: Qubit utilization of the distance-5 surface codes synthesized by Surf-Stitch on different architectures.

tiling of building blocks in Table 5.1 that is able to support the distance-5 surface code and then computing the ratios of different types of qubits.

The effect of architectures: High-degree architectures are more effective for surface code synthesis than low-degree architectures. Compared to polygon architectures, heavy architectures increase the bridge qubit number by 114% on average, up to 400%. Heavy architectures also increase the average time-step number by 40.7% on average. Fortunately, in Surf-Stitch, such significant resource differences do not lead to great error correction capability degradation. Compared to polygon architectures, heavy architectures reduce the error threshold by 26.7% on average. Besides, low-degree devices have a much lower hardware error rate and are easier to fabricate than high-degree devices [89].

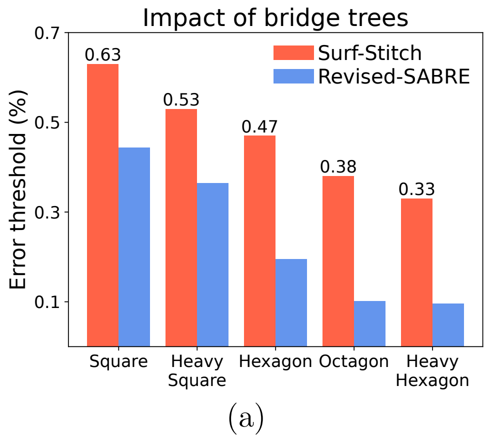
The effect of the synthesis design: The synthesized codes centering on four-degree qubits have higher resource requirements than the synthesized codes induced by a pair of three-degree qubits. In Table 5.3, 26.7% and 93.7% more qubits are required for 'Surf-Stitch Square-4' and 'Surf-Stitch Heavy Square-4' than 'Surf-Stitch Square' and 'Surf-Stitch Heavy Square', respectively. The synthesis by four-degree qubits may also have a lower error threshold. Compared to 'Surf-Stitch Heavy Square', 'Surf-Stitch Heavy Square-4' decreases the error threshold by 15.1%.

Code	bridge qubit #	bridge /data	2-qubit gate #	1-qubit gate #
Surf-Stitch Heavy Sqaure	$2(d^2-1)$	2	$8d(d-1)$	$2(d-1)*(3d+1)$
Surf-Stitch Heavy Hexagon	$2(2d+1)*(d-1)$	4	$4(4d-1)*(d-1)$	$2(d-1)*(7d-1)$
Surf-Stitch Square	$(d-1)*(d+2)$	1	$6d(d-1)$	$2(d-1)*(2d+1)$
Surf-Stitch Hexagon	$(2d+3)*(d-1)$	2	$10d(d-1)$	$2(5d-1)*(d-1)$
Surf-Stitch Octagon	$(d-1)*(3d+7)$	3	$14d(d-1)$	$2(d-1)*(6d+1)$
Ideal	d^2-1	1	$4(d^2-1)$	$2(d^2-1)$

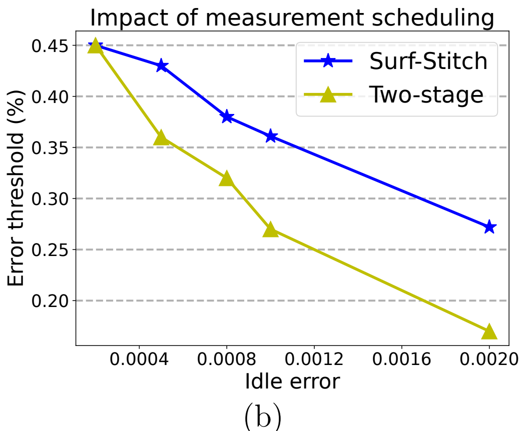
Table 5.4: Resource scalability of Surf-Stitch on different architectures. d is the code distance. The 'Ideal' rows denotes the ideal surface code on a 2D lattice [20].

Overall, not only the architectural design but also the synthesis design have a critical impact on the resource overhead and error correction capability of the synthesized codes. By optimizing the three key stages in the surface code synthesis, *Surf-Stitch* achieves reasonable error thresholds on various mainstream SC quantum architectures. In fact, IBM recently demonstrates a CNOT gate with a fidelity of 99.77% [102], whose physical error rate of 0.23% is lower than the worst error threshold by Surf-Stitch in Table 5.2 (0.33% on the heavy hexagon architecture).

Being effective in synthesizing surface codes, Surf-Stitch also has good scalability on quantum resources. Table 5.4 reports the bridge qubit number and quantum gate number by Surf-Stitch, with respect to the code distance d. These data are obtained by analyzing the patterns of the synthesized surface codes. As shown in Table 5.4, the required number of bridge qubits (also, two-qubit gates and single-qubit gates) in Surf-Stitch scales almost linearly with the number of data qubits, i.e., d^2. This indicates that no matter how large the code distance is, for a given architecture, Surf-Stitch only needs a constant number of bridge qubits to measure one stabilizer. Such good scalability comes from the fact that Surf-Stitch always uses small syndrome rectangles and only considers *local* bridge trees that have limited sizes and do not grow as the code distance increases.



(a)



(b)

Figure 5.8: Sub-component analysis. (a) The impact of different bridge trees. (b) The impact of different measurement scheduling.

5.4.4 Analysis on sub-components

In this section, we study the effect of Surf-Stitch's three optimization stages.

Data qubit allocation: To demonstrate the necessity of a specialized data qubit allocation algorithm, we compare the data qubit allocation pass of Surf-Stitch to Qiskit [28] and the random sampling method on device architectures in Table 5.1. For Qiskit, we try two different qubit layouts: SABRE [96] and NoiseAdaptive [90]. The random sampling method tries to find data qubits by sampling the device nodes uniformly. During 100000 trials, Qiskit and the random sampling method do not produce any *valid* data qubit layout that is able to execute all stabilizer measurement circuits without moving data qubits. The reason for the failure of these methods is that they do not consider the constraints of the surface code synthesis: (a) once allocated, data qubits should not be moved; (b) there should be high-degree qubits between data qubits. In contrast, Surf-Stitch always produces valid data qubit layouts.

Bridge tree construction: Keeping other optimization steps fixed, we compare the bridge tree construction algorithm of Surf-Stitch to a *revised SABRE* routing algorithm. We use a revised SABRE here because the original SABRE cannot distinguish bridge

qubits and data qubits. To make a CNOT gate executable, SABRE may move data qubits towards syndrome qubits. Such behavior is prohibited in the surface code because moving data qubits will invalidate the logical operations of the surface code. Besides, SABRE cannot guarantee the correct execution of stabilizer measurements. When we measure an X- and Z-stabilizer together, the order of the CNOT gates between syndrome qubits (or bridge qubits) and data qubits must follow the zig-zag ordering [20], as shown in Figure 2.5. Unfortunately, SABRE does not obey this constraint. Therefore, we revise the SABRE algorithm to make it applicable for implementing stabilizer measurements. As in Figure 5.8(a), the SABRE method is also significantly worse than Surf-Stitch due to the large number of extra CNOT gates induced by using SWAP gates. This result illustrates the necessity of a specialized bridge tree optimization pass.

Stabilizer measurement scheduling: Keeping the first two optimization stages fixed, we compare the stabilizer measurement scheduling of Surf-Stitch to the two-stage measurement scheme [31] which first measures all X-stabilizers and then measures Z-stabilizers. For the comparison, we consider the synthesized surface code in Figure 5.7(e). As shown in Figure 5.8(b), the stabilizer measurement schedule of Surf-Stitch achieves a higher error threshold than the two-stage schedule, and the advantage of Surf-Stitch increases as the idle error grows. Such a result demonstrates the effectiveness of Surf-Stitch's stabilizer measurement scheduling, especially for SC quantum devices in the near future.

Chapter 6

Synthesizing a Reliable Computing Platform

This chapter will discuss about how to build a reliable computing platform based on quantum error correction theory. This chapter will explore architecture-compiler co-designs so as to enable an efficient fault-tolerant quantum computing platform.

6.1 Introduction

Quantum computing suffers from device noise which greatly limits the problem size a quantum device can address with a low failure rate [103]. To address this challenge, quantum error correction (QEC) codes are invented to mitigate quantum noises and enable fault-tolerant quantum computing (FTQC) [1]. The QEC code encodes a high-fidelity logical qubit with a group of unreliable physical qubits (named data qubits) and corrects potential errors on the logical qubit based on error information extracted from data qubits. Experiments demonstrate the reliability of quantum architectures based on QEC and the enablement of QEC on realistic hardware has witnessed a series of

breakthroughs recently [11, 34, 31, 30, 104, 105, 106, 107, 108].

To execute quantum programs fault-tolerantly, the logical operation that can directly manipulate the logical qubit protected by the QEC code is indispensable. Like on physical qubits, it is critical to have a universal logical gate basis, such as the logical version of the Clifford+T gate basis, which can be used to achieve arbitrary unitary transformation directly on logical qubits. Unfortunately, not all logical operations can be implemented with a low cost, no matter what logical gate basis we adopted, according to [24]. Taking the most widely-used Clifford+T gate basis for example, for well-known QEC codes like the Steane code [1], the implementation of the logical T gate is much more complex than other logical gates and requires the support of extra QEC protocols.

To reduce the overhead of the difficult-to-implement logical operation like the logical T, it is critical to optimize QEC protocols required to implement such logical operation. Among these protocols, magic state distillation and code switching are two most promising methods. Magic state distillation consumes multiple logical qubits to implement the logical T gate. Its advantages as well as optimization opportunities have been extensively studied in literature [1, 25, 84]. Distinctly, code switching [27] allows a cheap implementation of the logical T by encoding the logical qubit in different QEC codes along the time dimension. While being complicate on the Steane code, the logical T can be easily implemented on the Reed-Muller (RM) code, by simply applying physical $T^\dagger$ gates on each data qubit. To implement the logical T on a logical qubit of the Steane code, we can first use one code switching to transform the logical qubit into the RM code mode, implement the logical T, and then switch the logical qubit back into the Steane code. As an emerging method, the advantage of code switching has not been well investigated yet. There are large optimization spaces left for exploration.

Especially, existing efforts of code switching [27] only consider the theoretical protocol for implementing the logical T, totally missing optimization opportunities from the

architecture and compiler aspects that can further enlarge the benefit of code switching. From the architecture perspective, it is unclear how we can implement QEC with code switching (in short, QEC-CS) on quantum hardware. For example, what is the optimal data qubit layout for a QEC-CS logical qubit? How can we place multiple QEC-CS logical qubits to achieve optimal computational efficiency? These two critical questions need to be addressed before we reach an effective design of the QEC-CS architecture. From the compiler perspective, the unique optimization opportunity imposed by code switching remains unexplored. Code switching is an expensive QEC operation and requires a series of noisy and time-consuming physical operations between data qubits in a logical qubit [27]. Reducing the utilization of code switching is critical for minimizing the overhead of programs on the QEC-CS architecture.

A better understanding of the effectiveness of code switching in the architecture and program context is critical towards FTQC. Without comprehensive exploration, it is hard to determine what architectural designs for logical qubits and gates can lead to more efficient FTQC. Furthermore, studying code switching in the program context can reveal the advantages and optimization chances of code switching for FTQC on quantum applications. In its early stage, FTQC urges extensive exploration of various design spaces to unveil the promising next step.

To this end, we propose the first full-stack framework, named FLEX, to provide architecture and compiler support for FTQC based on QEC-CS. Firstly, we present a comprehensive architecture design for the data qubit layout of one QEC-CS logical qubit. We observe that three types of QEC operations (error detection, code switching, and logical gates) have different impacts on the reliability and space overhead of the logical qubit. Their hardware implementation may even conflict with each other. To address those problems and find the optimal data qubit layout, we propose using the profiling data (e.g., impact on logical operation error rates, consumed physical qubits) and structural

information (e.g., data qubit locations in an error detection circuit) of those QEC operations to guide the search priority and circumvent implementation conflicts. Experiments demonstrate the effectiveness of our architectural design over the large search space.

Secondly, we present a compiler design that supports the efficient execution of quantum programs on the QEC-CS architecture. On the one hand, we observe that a pair of code switching operations can be used to execute more than the logical T gate depending on the program context, saving program latency and improving fidelity. On the other hand, we observe that executing too many logical gates between a pair of code switching operations may instead hurt the program fidelity considering the different error correction capabilities of the two QEC codes being switched. Our compiler achieves a good balance for this unique trade-off in the QEC-CS architecture and far surpasses existing QEC compilers in reducing the space-time overhead of quantum programs, as demonstrated by our experiments.

Thirdly, we present a co-design of the QEC-CS architecture and compiler to further promote the computational potential of QEC-CS. We observe that, for compiled quantum programs of different features, regarding different optimization metrics, it is better to use different connectivity between logical qubits and place QEC-CS logical qubits accordingly. This is the first co-design study of FTQC since existing works [11, 34, 31, 30] only consider the simulation of one logical qubit. Our evaluation shows that the co-design can further improve the performance of specific quantum programs in particular metrics, e.g., the space-time overhead and the program fidelity.

Finally, we compare our QEC-CS architecture to the QEC architecture based on magic distillation. Results show that code switching is more promising than magic state distillation in the program context, especially in the space-time overhead.

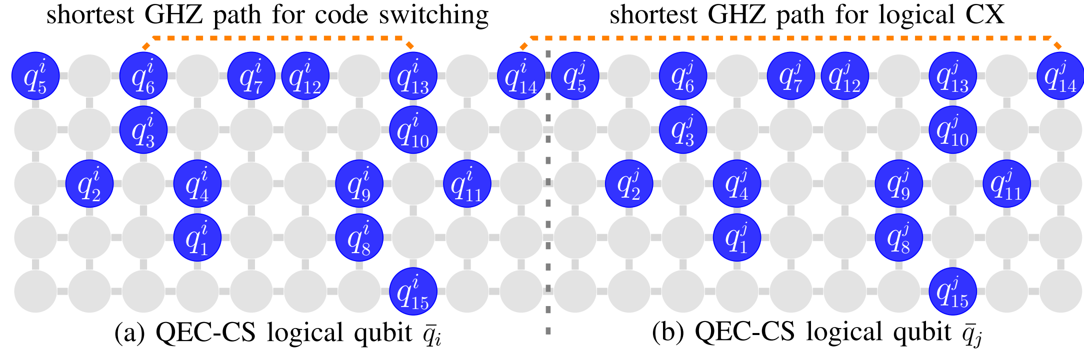


Figure 6.1: An example layout of data qubits (q_*^i, q_*^j) for QEC-CS logical qubits $\bar{q}_i$ and $\bar{q}_j$, which only minimizes the error detection cost.

6.2 Design Considerations

Our objective is to improve the overall fidelity as well as reduce the space-time overhead of running quantum programs on the QEC-CS architecture by orchestrating the architecture and software design. Here, we highlight the critical considerations and observations to achieve our design objectives.

6.2.1 Architectural Design for QEC-CS

1) Challenges in logical qubit design

Generally, to enforce QEC support on quantum hardware, we need to map data qubits and error detection circuits of logical qubits to hardware [11]. However, for the QEC-CS logical qubit design, we should further support dynamic code switching between two QEC codes. Overall, when placing data qubits of a QEC-CS logical qubit on hardware, we expect the overhead (e.g., involved qubit/gate count, latency) of executing error detection, logical CX, and code switching on data qubits to be as small as possible. A QEC-CS logical qubit with lower overhead is more reliable because of its fewer physical gates and qubits (i.e., fewer error locations [1]).

Unfortunately, the overhead optimization of error detection differs from the optimiza-

tion of code switching, and logical gates. The error detection circuit requires data qubits to be close to the parity qubit so that we can use fewer qubit resources (e.g., flag qubits) to gather error information from data qubits that are not in the neighborhood of the parity qubit. However, placing data qubits too close will instead hinder the implementation of logical CX and code switching. Figure 6.1 shows an example data qubit layout that minimizes error detection cost but the code switching and logical CX require using long GHZ paths. Indeed, for code switching, the shortest GHZ path connecting q_6^i and q_{13}^i is blocked by q_7^i. While for logical CX, the shortest GHZ path between q_{14}^i and q_{14}^j is blocked by q_5^j.

Similarly, the overhead optimization of code switching differs from the optimization of logical CX. Code switching requires logical CX inside the QEC-CS logical qubit and we should make data qubits involved in this interior logical CX close to reduce the length of required GHZ paths. However, this optimization may hinder the logical CX between two QEC-CS logical qubits. For example, in Figure 6.1, the shortest GHZ path between q_2^i and q_2^j required by $CX_L\,\bar{q}_i\,\bar{q}_j$ is blocked by q_9^i, which is involved in CX gate within the code switching protocol.

Overall, it remains unclear how to design the layout of the QEC-CS logical qubit to coordinate error detection, code switching, and logical CX. To address this challenge, we argue it is critical to gather quantitative data about the impacts of these three types of QEC operations on the reliability of the QEC-CS architecture.

2) Observations for architecture design

Table 6.1 shows the simulation data for the Steane logical CX gate, based on different design options of # flag qubit and GHZ paths (see Sec. 6.4 for simulation details). We observe that it is more critical to reduce error detection overhead (i.e., # flag qubit). For example, for the length-0 GHZ path and one flag qubit option, adding one more flag qubit will decrease the pseudo-threshold of logical CX by 0.0017 while adding one

# flag qubit \ GHZ path length	0	1	2	3	4
1	0.0034	0.0022	0.0016	0.0012	0.0009
2	0.0017	0.0013	0.0010	0.0008	0.0006

Table 6.1: Pseudo-threshold of the Steane logical CX for varied # flag qubit, and GHZ path length (averaged over data-qubit pairs).

edge in the GHZ path only lead to a decrease of 0.0012. This is because error correction following the logical CX can mitigate the negative effect of longer GHZ paths. Further, larger error detection overhead will also hurt the fidelity of other logical operations, e.g., code switching and single-logical-qubit gates.

Further, we observe that logical CX should be optimized before code switching as it appears more frequently in quantum programs. For example, quantum arithmetic circuits are based on the Toffoli gate. The decomposed Toffoli gate has seven T gates [1], meaning that a Toffoli gate requires at most 14 code switching operations with two for each T_L. However, on a connectivity-limited qubit layout, the overall CX count may surpass the amount of code switching required, as each CX will generally induce 4 SWAP gates for routing [109] and we have 6 CX gates in the Toffoli gate (see Sec. 6.4 for more program details).

Observation 1: a good data qubit layout for a QEC-CS logical qubit should first reduce the overhead of error detection, then the logical CX, and finally the code switching.

After the design for an individual QEC-CS logical qubit, the next step is to design the layout of multiple QEC-CS logical qubits to support quantum programs. The placement of multiple QEC-CS logical qubits affects the 'logical connectivity' between logical qubits. On the one hand, enabling direct logical CX between remote logical qubits by the GHZ path may not feasible due to its low reliability. Also, a long GHZ path may hinder logical CX gates on logical qubits near the path since two GHZ path should not intersect for functional correctness. On the other hand, higher logical connectivity (i.e., more direct

logical CX) will reduce the number of SWAP gates for routing thus improving program fidelity and reducing latency. Whether increasing the logical connectivity or not should consider the feature of input programs. For example, chemistry simulation programs like UCCSD [110] requires many remote CX and prefer higher connectivity. In contrast, arithmetic circuits focus on local interactions between two or three qubits [110], for which a small connectivity is enough.

Observation 2: The placement of logical qubits affects logical CX and the 'logical connectivity'. By orchestrating the logical qubit layout design and the compiler design, we may further improve program fidelity or reduce latency.

6.2.2 Compiler Design for QEC-CS

Different implementations of non-transverse gates (e.g., T_L) often induce different compiler designs. Unfortunately, existing QEC compilers do not support the optimization of code switching, which is the critical difference between QEC-CS to other QEC architectures. We observe two challenges for optimizing the code switching usage in quantum programs. The first is to determine the non-transverse logical gate we expect to execute with code switching. We can adopt code switching to implement T_L is for the Steane code or H_L for the RM code. The second is to determine the number of logical gates we expect to execute after one invocation of code switching. The Steane and RM code share many transverse logical gates, e.g., CX_L, X_L. If we perform code switching from Steane code to RM code for T_L, it is unclear whether we should switch back to the Steane code directly after T_L executed, or perform more transverse gates in the RM code before switching back.

Firstly, we argue that most logical gates should be executed in the Steane code mode of QEC-CS logical qubits and use code switching for T_L. The insight is that logical gates

of Steane code is more reliable than the ones of RM code since Steane code has fewer error locations [111]. We should avoid executing too many logical gates on the RM code mode. Similar conclusion holds for other code pairs for switching [27].

Secondly, we observe that it may be more advantageous to execute more than one T_L gate after one code switching operation. Executing more logical gates between two code switching may increase infidelity, but it also reduces the count of code switching, which is error-prone and time-consuming. To give an example, let us consider the gate sequence:

$$T_L\ \bar{q}_0; T_L\ \bar{q}_1; CX_L\ \bar{q}_0\,\bar{q}_1; T_L\ \bar{q}_1, \tag{6.1}$$

which frequently appears in quantum programs [112] (e.g, arithmetic circuits, Grover). Two logical qubits ($\bar{q}_0$ and $\bar{q}_1$) are first switched from Steane code to RM code to perform the first two T_L. If we switch the two logical qubits back to Steane code after the two T_L are executed, we will need another two code switching to execute the last T_L on $\bar{q}_1$. Instead, if we delay the code switching of $\bar{q}_0$ and $\bar{q}_1$ after the last $T_L\ \bar{q}_1$ executed, we can save two code switching on $\bar{q}_1$. Such a delay of code switching is advantageous, as long as the infidelity/latency induced by two code switching operations is larger than the infidelity/latency difference between the RM CX and the Steane CX. Indeed, this condition always holds as shown in Section 6.4.

Overall, we are not going to delay all code switching since executing too many logical gates in the less reliable code mode may hurt the program fidelity as discussed before. We can achieve a good balance by inspecting the program context (e.g., the one in Equation 6.1) and determining whether to delay code switching or not according to fidelity/latency gain.

Observation 3: The code switching for T_L should be applied in a context-aware way for more computational benefits.

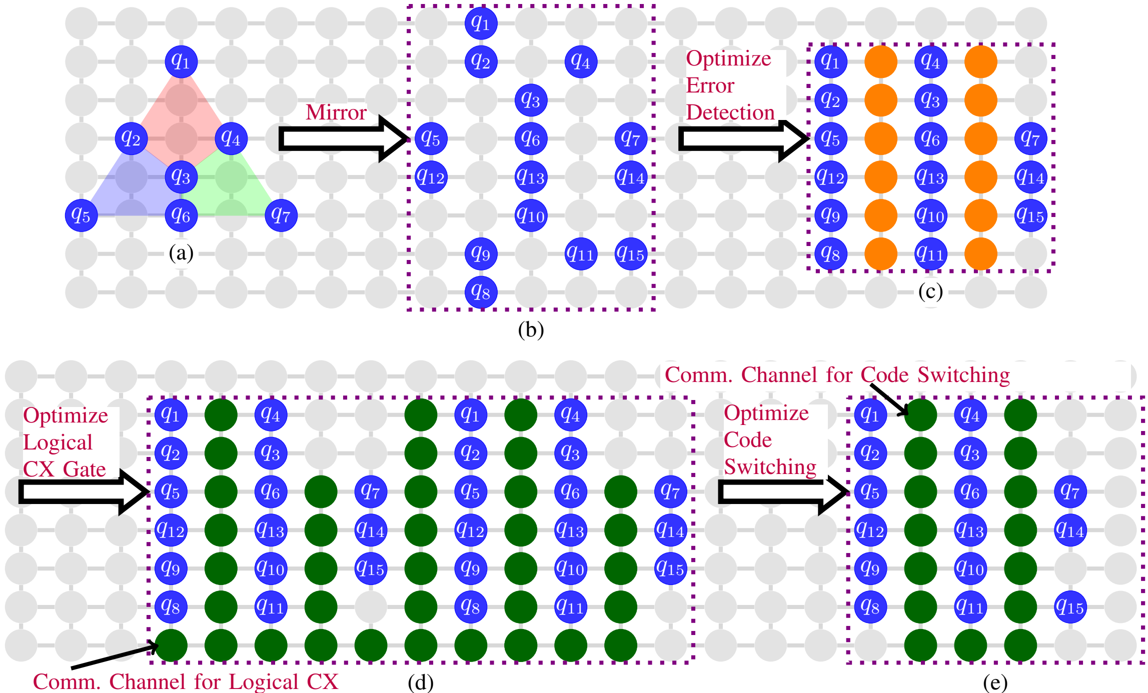


Figure 6.2: Optimizing the data qubit layout of one QEC-CS logical qubit with knowledge from error detection, code switching, and logical CX.

6.3 QEC-based Computing Platform Design

In this section, we implement the QEC-CS architecture and compiler based on observations outlined in Section 6.2.

6.3.1 The QEC-CS Architecture

1) Implementing one QEC-CS logical qubit

We first search for the data qubit layout of one QEC-CS logical qubit. Assume the (color) code pairs for code switching is C_1 and C_2 with $C_1 \subsetneq C_2$ according to [27]. We first determine the initial layout of the QEC-CS logical qubit. The insight is to use the geometrical shape of stabilizer operators. For example, for $C_1 = \textit{Steane code}$, there are spatial relations between data qubits of the Steane code and we can use this relation to determine the initial layout of C_1, as shown in Figure 6.2(a). The initial layout of C_2

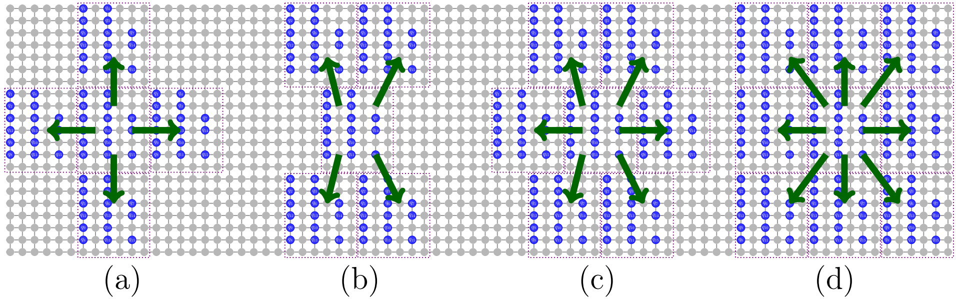


Figure 6.3: Examples of placing multiple logical qubits. Green arrows denote logical CX directions. (a) connectivity 4. (b) connectivity 4 rotated. (c) connectivity 6. (d) connectivity 8.

can be determined based on the relation of C_1 and C_2. For instance, for $C_2 = RM\ code$, C_2 is almost doubling the C_1 and its initial layout can be found by mirroring the Steane code layout, as shown in Figure 6.2(b). Such a relation of C_1 and C_2 widely exists in code pairs for code switching [27].

After the initial layout of C_2 determined, the first step is to guarantee the minimal overhead of error detection circuits of C_1 and C_2, according to **Observation 1**. We fine-tune the data qubit layout of C_1 and C_2 for smaller overhead of error detection circuits according to Equation 6.2:

$$\forall s \in S, \min_{ts \in \{\uparrow,\downarrow,\leftarrow,\rightarrow,nop\}} TOT_CX(\{ts_i(q_i), \cdots\}), \qquad \{q_i, \cdots\}\text{: qb of } s \tag{6.2}$$

The function TOT_CX is computed as the total edge count of the smallest tree of flag qubits that connects all data qubits in an error detection circuit s. The overall set of error detection circuits S of C_1 and C_2 is organized by the weight of stabilizer operators. The optimization by Equation 6.2 will be repeated (< 3 times for $C_2 = RM\ code$) for a smaller overhead of error detection. Figure 6.2(c) shows the optimized layout of the QEC-CS logical qubit for $C_1 = Steane\ code$ and $C_2 = RM\ code$, where ancillary qubits (parity and flag qubits) used for error detection circuits are orange dots.

Further, we should reduce the cost of the logical CX and the code switching op-

eration. For those two operations, the source of overhead is the GHZ path between non-neighboring data qubits. The goal of this step is thus to reduce the total lengths of GHZ paths while still guaranteeing a small overhead of error detection. The insight for this step is that to support remote CX between vertically distant data qubits, there should be a vertical line of unoccupied physical qubits as the communication channel (see Figure 6.2(d) green dots). Likewise, for horizontally distant data qubits, there should be a horizontal line of unoccupied physical qubits as the communication channel. Such communication channels can be guaranteed by moving each data qubit along a horizontal or vertical direction by at most **one step**, according to Equation 6.3 ('EC': error correction):

$$\min_{ts \in \{\uparrow,\downarrow,\leftarrow,\rightarrow,nop\}} \sum_i GHZ_LEN(ts_i(q_i), ts_i(q_i^h)) + GHZ_LEN(ts_i(q_i), ts_i(q_i^v)) + TOT_CX(Steane\ EC + RM\ EC) \tag{6.3}$$

The function GHZ_LEN is computed as the length of the shortest uninterrupted GHZ path between data qubits. q_i^h and q_i^v are data qubits of the horizontal and vertical logical qubit neighbors, respectively. Figure 6.2(d) shows the logical CX-optimized layout of the QEC-CS logical qubit for $C_1 = Steane\ code$ and $C_2 = RM\ code$. This small tweaking of the data qubit layout will not greatly influence the overhead of error detection. The optimization for the logical CX will also provide sufficient communication channels for the 'interior' logical CX of code switching. Thus, to reduce the overhead of remote physical CX in code switching, we will optimize the data qubit layout according to Equation 6.4 ('CS': code switching):

$$\min_{loc(q_i),\, q_i \in C_2} TOT_GHZ_LEN(CS) + TOT_GHZ_LEN(CX) + TOT_CX(Steane\ EC + RM\ EC) \tag{6.4}$$

This optimization for code switching determines the location of data qubits in an ascending order, based on the count of error detection circuits each data qubit involves. The

optimization in Equ 6.4 also keeps low overhead of error detection and logical CX.

Overall, for the example where C_1 = *Steane code* and C_2 = *RM code*, the tuned layout of the QEC-CS logical qubit is shown in Figure 6.2(e). In the figure, ancillas in the communication channel (i.e., GHZ paths) are time-multiplexed and will also be used for error detection or code switching (see Figure 6.2(c)(d)(e)). The preparation latency for GHZ states is not significant, as it only involves two layers of physical CX gates and one layer of measurement-conditioned Pauli gates for any-length GHZ path. The GHZ path connects data qubits of the same numbering (i.e., q_1 to q_1 in Figure 6.2(d)) but in different logical qubits. Our design in Figure 6.2(e) enables at most 4 simultaneous GHZ state preparation. The GHZ state preparation latency is a part of the logical CX latency.

2) Placing multiple QEC-CS logical qubits

There is a large amount of freedom when placing multiple logical qubits. We can tile logical qubits along horizontal, vertical, or diagonal directions, as shown in Figure 6.3, where logical qubit layouts with different connectivity are demonstrated. Table 6.2 shows the latency and fidelity of logical CX gates on different layouts (see Sec. 6.4 for more simulation details).

Layouts with different connectivity have their own advantages. Higher-connectivity layouts usually induce fewer SWAP gates than lower-connectivity layouts, thus can produce more reliable program outcomes. For example, for the connectivity-4 layout, if we perform a (Steane) logical CX along the diagonal direction, we need one logical SWAP gate along the vertical direction plus one horizontal logical CX gate, leading to an error rate of $1.4 * 10^{-8}$ when the device error rate is 10^{-6} (see Table 6.2). In contrast, on the connectivity-8 layout, the diagonal logical CX is directly executable, with a lower error rate at $5.3 * 10^{-9}$ (see Table 6.2). Thus, establishing direct logical CX between distant qubits to increase connectivity may boost the reliability of the resulting QEC-CS layout.

However, higher connectivity of the layout may instead hurt the parallelism of logical

CX. For two logical CX, if their communication channels (green dots in Figure 6.2) intersect with each other, these two logical CX cannot be executed simultaneously. For example, on the connectivity-8 layout, for a logical qubit $\bar{q}_i$, denoting its upper left, upper, left logical qubits as $\bar{q}_{i+1}$, $\bar{q}_{i+2}$, $\bar{q}_{i+3}$, CX_L $\bar{q}_i$ $\bar{q}_{i+1}$ and CX_L $\bar{q}_{i+2}$ $\bar{q}_{i+3}$ cannot be concurrently executed as their GHZ paths intersect. Generally, the more distant two logical qubits of one logical CX is, the more logical CX may lag behind. Moreover, GHZ paths for each data qubit in the same logical qubit may also interfere with each other. As shown in Table 6.2, on the connectivity-8 layout, the latency of diagonal logical CX is far longer than the vertical logical CX.

Different layouts may fit programs of different features, providing opportunities for architecture-compiler co-design. For example, if one compiled program does not have much parallelism between logical CX, e.g., the UCCSD benchmark, using the connectivity-8 layout may be better than other layouts as for the fidelity and latency of executed logical CX. On the other hand, for the compiled program without many long-distance logical CX, using the connectivity-4 (rotated) layout may instead improve CX parallelism while still maintaining the same level of fidelity. Therefore, we can pick the best layout for a program according to its logical CX feature. This co-design can further promote FTQC based on code switching.

Overall, in Section 6.4, we have provided more quantified performance data for qubit layouts in Figure 6.3 and demonstrated the computational benefits of architecture-compiler co-design.

6.3.2 Compiler for QEC-CS

To provide program compilation support for the QEC-CS architecture, we need to address two major tasks: QEC-CS architecture abstraction and code switching optimiza-

Logical CX gates		Connectivity-4	Connectivity-4 rotated	Connectivity-6	Connectivity-8
Horizontal Steane CX	Latency	72.1	—	72.1	72.1
	Error Rate	$3.9 * 10^{-9}$	—	$3.9 * 10^{-9}$	$3.9 * 10^{-9}$
Vertical Steane CX	Latency	20.6	—	—	20.6
	Error Rate	$3.3 * 10^{-9}$	—	—	$3.3 * 10^{-9}$
Diagonal Steane CX	Latency	—	30.9	30.9	72.1
	Error Rate	—	$3.8 * 10^{-9}$	$3.8 * 10^{-9}$	$5.3 * 10^{-9}$
Horizontal RM CX	Latency	82.4	—	82.4	82.4
	Error Rate	$3.7 * 10^{-8}$	—	$3.7 * 10^{-8}$	$3.7 * 10^{-8}$
Vertical RM CX	Latency	41.2	—	—	41.2
	Error Rate	$2.7 * 10^{-8}$	—	—	$2.7 * 10^{-8}$
Diagonal RM CX	Latency	—	51.5	51.5	154.5
	Error Rate	—	$3.0 * 10^{-8}$	$3.0 * 10^{-8}$	$3.9 * 10^{-8}$

Table 6.2: Performance of logical CX for different logical qubit placement. The error rate of logical CX is computed when the device error level is 10^{-6}. The latency is normalized to physical CX counts.

tion.

QEC-CS architecture abstraction. The first task is to abstract the QEC-CS architecture, including the logical qubit topology and basic instructions. The coupling graph of logical qubits can be directly extracted from the given QEC-CS architecture. For the instruction set, we will expose two more instructions besides Clifford+T logical gates. The first one is **EC** which means to perform error correction. The second one is **CS**, meaning to perform code switching. The **CS** instruction provides the compiler with the ability to control code switching. Further, the fidelity and latency data of the instruction set can be simulated and computed according to their specific implementations (see Sec.6.4 for more details).

Optimization of code switching. The second task is to reduce the usage of code switching which is necessary but is error-prone and time-consuming. Conventionally, we will use two code switching operations for a logical T gate with one before it and one after it. Here, we propose reducing the amount of code switching required by a quantum program in a context- and fidelity-aware way (**Observation 3**), with following steps:

Step 1: Blocking. For a given circuit, we search for the next (logical) T gate. Assuming qubit q_0 is associated with this T gate, we create a new block blk containing

the T gate. We denote the qubit list of *blk* by *qlist*. For q_i in *qlist*, we will also add other gates that are close to *blk* and are applied on q_i to *blk* until the new gate on q_i is H gate. We will update *qlist* in this process since CX gates may have been added to *blk*. Overall, *blk* represents the largest scope of **CS** on q_0.

Step 2: Gate reordering. Then we will move CX, X, and Z gates of *blk* outside this block by circuit writing [53] so that we can keep most gates executed in the Steane code mode of QEC-CS logical qubits, and only execute necessary gates in the RM code mode, according to Observation 3.

Step 3: Block refining. For each CX gate g in *blk*, computing the fidelity of *blk* if we mark g to be executed by the Steane code mode. If the fidelity is improved, we will split *blk* into two parts, each part representing a new scope of **CS** on q_0. This step is repeated until no splitting is possible. The goal of this step is to ensure no fidelity loss is caused by executing logical gates in the RM code mode.

We will repeat the above three steps until no new blocks are found. Finally, we will add CS instructions at the start and end of each refined block. The start and end of each refined block in Step 3 represent the optimized timing of applying code switching on related logical qubits.

As an example of the benefit of CS optimization, let us consider the circuit in Figure 6.4. For the logical qubit in Figure 6.2(e) and device error rate $< 1e^{-4}$, the normalized infidelity of RM CX, RM 1q, Steane 1q and CS are about 8.8, 2.6, 0.2, and 4.1 Steane CX, respectively. Also, the normalized latency of RM CX, RM 1q, Steane CX, Steane 1q and CS are about 5.5, 3.0, 2.9, 1.0 and 9.1 Steane EC rounds, respectively. Without CS optimization, the CS count, the normalized infidelity, and the normalized latency for the example circuit are 14, 82.0, and 167.8, respectively. With the proposed CS optimization, the block found is highlighted with dotted lines in Figure 6.4. The CS count, normalized infidelity, and normalized latency for the example circuit are reduced to 6,

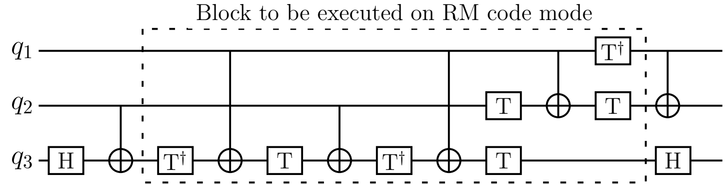


Figure 6.4: An circuit (Toffoli gate) for illustrating CS optimization.

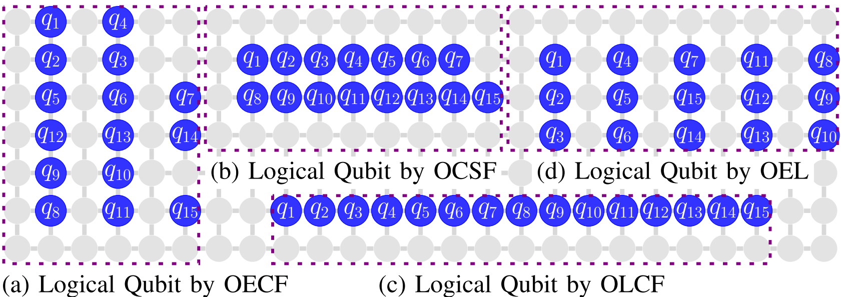


Figure 6.5: QEC-CS logical qubits by four different schemes. Blue circles denote data qubits, with other qubits within the same dotted rectangle serving as ancillary qubits for QEC operations.

80.4, and 105.4, respectively.

6.4 Evaluation

6.4.1 Experiment Setup

Benchmark programs We consider two categories of benchmark obtained from [110], as shown in Table 6.4. The first category focuses on implementing arithmetic functions, e.g., multi-qubit XOR, ripple-carry adder and ripple-carry comparator (in short, XOR, Adder and Comparator). These quantum programs are subroutines of large quantum applications. The second category aims to solve practical problems, e.g., Grover's algorithm, quantum walking, and Unitary Coupled Cluster ansatzes (UCCSD). For quantum walking, we specifically select the Binary Welded Tree (BWT) algorithm. For UCCSD, we simulate the CH_4 molecule. All programs are decomposed into circuits with the

Arch Layouts	Resource Overhead						Operation Latency					Pseudo-threshold				
	# CX for Steane EC	# CX for RM EC	Avg. ghz length for CS	Avg. ghz length for Steane CX	Avg. ghz length for RM CX	# physical qubit	Steane EC	RM EC	Code Switching	Avg. Steane CX	Avg. RM CX	Steane 1q gate	Avg. Steane CX	RM 1q gate	Avg. RM CX	Code switching
OECF	40	152	2.7	8.64	9.73	42	24.8	74.4	225.1	46.35	61.80	$1.4e^{-3}$	$2.8e^{-4}$	$1.1e^{-4}$	$3.1e^{-5}$	$6.8e^{-5}$
OCSF	68	268	0.7	10.14	9.83	36	69.2	187.8	487.2	41.20	82.40	$5.5e^{-4}$	$1.5e^{-4}$	$3.8e^{-5}$	$1.6e^{-5}$	$2.9e^{-5}$
OLCF	68	236	6.5	8.50	8.50	30	69.2	190.2	529.8	30.90	56.65	$5.5e^{-4}$	$1.7e^{-4}$	$4.8e^{-5}$	$2.1e^{-5}$	$2.5e^{-5}$
OEL	40	212	5.7	8.57	8.67	40	24.8	115.4	277.4	36.05	56.65	$1.4e^{-3}$	$2.8e^{-4}$	$5.9e^{-5}$	$2.4e^{-5}$	$3.8e^{-5}$

Table 6.3: Performance of QEC-CS logical qubit layouts. 'OECF' is our proposed one. EC: error detection circuits, CS: code switching, latency data: normalized to physical CX counts, ghz length: GHZ path length for physical CX between distant qubits.

Program	# qubit	# gate	# CX	OECF+AgnosticCS					**OECF+AwareCS**				
				Norm. infidelity	Fidelity point	Latency	Space *Time	# CS	Norm. infidelity	Fidelity point	Latency	Space *Time	# CS
XOR	20000	$2.19e^{6}$	$1.79e^{6}$	$4.76e^{6}$	98.3%	$1.47e^{8}$	$1.24e^{14}$	$5.60e^{5}$	$4.72e^{6}$	98.3%	$1.01e^{8}$	$8.51e^{13}$	$3.20e^{5}$
Adder	30000	$2.27e^{6}$	$1.97e^{6}$	$4.20e^{6}$	98.5%	$9.85e^{7}$	$1.24e^{14}$	$4.20e^{5}$	$4.17e^{6}$	98.6%	$6.39e^{7}$	$8.35e^{13}$	$2.40e^{5}$
Comparator	30000	$2.39e^{6}$	$2.06e^{6}$	$4.29e^{6}$	98.4%	$1.00e^{8}$	$1.27e^{14}$	$4.20e^{5}$	$4.26e^{6}$	98.5%	$6.63e^{7}$	$8.35e^{13}$	$2.40e^{5}$
Grover	38	$1.05e^{6}$	$6.30e^{5}$	$3.71e^{6}$	98.7%	$1.33e^{8}$	$2.12e^{11}$	$5.79e^{5}$	$3.66e^{6}$	98.7%	$8.56e^{7}$	$1.37e^{11}$	$3.31e^{5}$
BWT	28	$1.03e^{6}$	$5.98e^{5}$	$3.81e^{6}$	98.6%	$1.38e^{8}$	$1.62e^{11}$	$6.06e^{5}$	$3.77e^{6}$	98.6%	$8.80e^{7}$	$1.03e^{11}$	$3.46e^{5}$
UCCSD	16	$1.07e^{6}$	$1.79e^{5}$	$4.70e^{6}$	98.3%	$2.36e^{8}$	$1.58e^{11}$	$8.23e^{5}$	$4.70e^{6}$	98.3%	$2.36e^{8}$	$1.58e^{11}$	$8.23e^{5}$

Table 6.4: Compilation results on a square grid of OECF logical qubits. 'Norm. infidelity': program infidelity normalized to Steane logical CX counts (for $p_e < 10^{-4}$). 'Fidelity point': program fidelity when $p_e = 10^{-6}$. OECF+AwareCS: our optimizations.

Clifford+T gate basis.

Baseline We are the first full-stack framework for QEC-CS. Different baselines are considered to unveil the huge space of architecture/compiler optimization and provide an in-depth analysis of our design. For architecture design, four schemes for generating QEC-CS logical qubits are evaluated. The first one, named **OECF**, optimizes the resource overhead of error detection circuits as the first priority (then logical CX, code switching). **OECF is the proposed layout design**. The second one, named **OCSF**, optimizes the resource overhead of the code switching operation as the first priority (then error detection, logical CX). The third one, named **OLCF**, optimizes the resource overhead of the logical CX gate as the first priority (then error detection, code switching). The fourth one, named **OEL**, evenly places data qubits of the logical qubit respecting the underlying architecture. It first allocates the location of data qubits and then permutes the location of these data qubits for the best error correction capability, cheapest logical

CX gates, and code switching in turn. Figure 6.5 shows logical qubit layouts generated by the four schemes. for the Steane code and RM code.

For compiler optimizations for code switching, we evaluate two schemes. The first one, named *AgnosticCS*, is context-agnostic which always uses two code switching operations for a logical T gate (Steane mode → RM mode → Steane mode). AgnosticCS is the common strategy of using code switching in existing works [27]. The second one, named *AwareCS*, is context-aware which executes usually more than one logical gate during two code switching. AwareCS always tries to reduce the usage of code switching as long as no fidelity loss is observed. **AwareCS is the proposed optimization**. For compiler optimizations except for code switching, they are not our focus and we adopt existing toolkits for them. Specifically, we use PyZX 0.7.3 [113] to optimize T count and Qiskit 0.35.0 [28] to optimize CX count (e.g., the SWAP gates induced by routing).

Hardware setup. We perform all experiments with Python 3.10.6 on a Ubuntu 22.04 server with two 28-core Intel Platinum 8280 Processor and 1TB RAM. For the architecture design, our framework finishes in less than one minute. Our compiler finishes in less than 10 minute for each program.

Metrics. For the architecture design, the main metrics are the resource overhead, latency and fidelity of error detection, code switching, and logical CX gates. These metrics quantify the performance of one QEC-CS logical qubit. Besides, we also consider the space-time overhead ('# physical qubit used * latency') and SWAP cost required to execute benchmark programs, in order to evaluate the performance of the placement of multiple logical qubits. For the compiler design, the first metric is the code switching count required for a quantum program. We expect to reduce code switching operations. Another two are conventional compiler metrics: the overall program fidelity and latency. We estimate these two metrics based on logical operation statistics.

Device model and noise simulation. We consider the widely-adopted square-grid

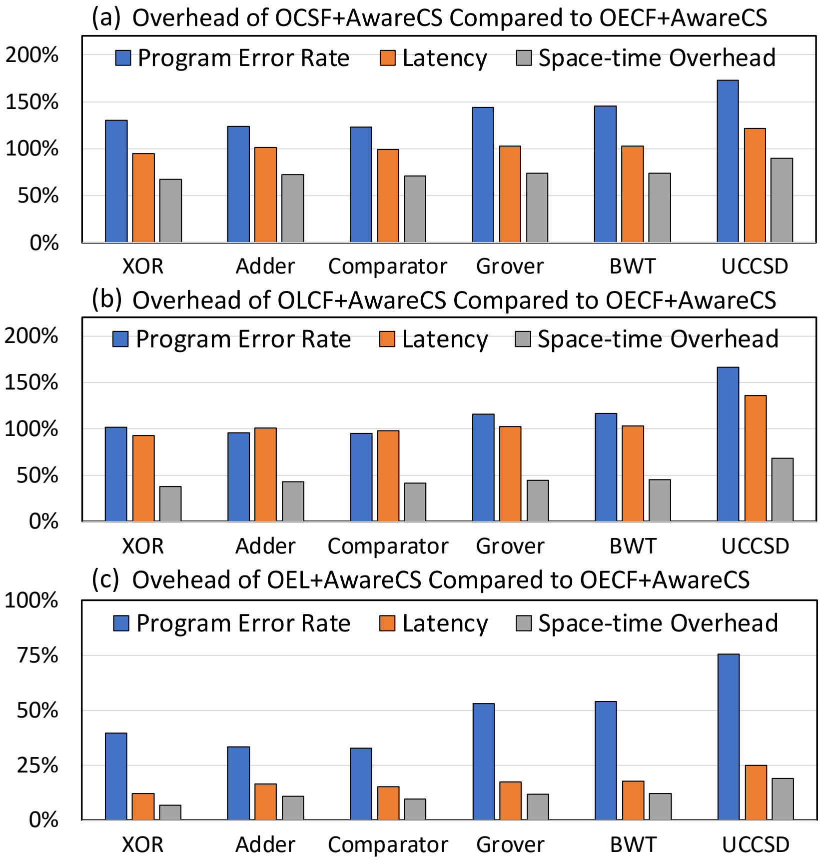


Figure 6.6: 'Overhead of the tested logical qubit design/overhead of OECF'*100%-1. Logical qubits are arranged in a square grid.

connectivity [105] for physical qubits. Further for the error feature of them, we consider the commonly-used circuit noise model [114, 99], with a physical error probability p_e for the single-qubit depolarizing error channel on single-qubit gates, the two-qubit depolarizing error channel on two-qubit gates, and the Pauli-X error channel on measurement and reset operations. For error decoding of QEC-CS, two different lookup decoders are designed specifically toward the code pairs considered. Lookup decoders are widely-used in literature [114]. The decoding latency has a tiny impact on the error detection latency in our experiments. One error detection round is added behind each logical gate.

6.4.2 Experiment Results

Table 6.3 shows the performance of different logical qubit designs in terms of resource overhead, latency, and fidelity. Figure 6.6 shows the performance of different logical qubit designs in the quantum program context. Table 6.4 and Figure 6.7 illustrate the performance of proposed code switching optimizations. Figure 6.8 demonstrates the effect of ‘logical connectivity’ when placing multiple QEC-CS logical qubits. ‘Connectivity-4/6/8’ means to let each QEC-CS logical qubit have 4/6/8 neighboring logical qubits. The ‘Connectivity-4 rotated’ architecture is achieved by rotating the ‘Connectivity-4’ architecture by $\frac{\pi}{4}$, as shown in Figure 6.3.

Overall, the infidelity, space-time overhead, and latency of test programs on other logical qubit layouts (OCSF, OLCF, and OEL) is on average 99.3%, 43.9% and 75.1% higher than that on the OECF layout, respectively. Moreover, compared to AgnosticCS, the proposed AwareCS on average reduces the space-time overhead and code switching count of test programs by 43.8% and 62.5%, respectively. Finally, the results show that increasing ‘logical connectivity’ does not necessarily induce a better FTQC platform and different programs favor different placement of logical qubits. It is thus important to adopt architecture-compiler codesigns to further improve the efficiency of FTQC. We elaborate on these conclusions in the following analysis.

1) The effect of logical qubit design

Firstly, it is critical to reduce the resource overhead of error detection circuits as the first priority when designing a logical qubit. As shown in Table 6.3, compared to OCSF and OLCF, though OECF has higher resource overhead for code switching and logical CX gates, it still has the highest error threshold for all logical operations. This indicates that error detection has the largest impact on the fidelity of logical operations and OECF’s low overhead for error detection guarantees the reliability of logical operations. The OEL

logical qubit has the second-best error threshold, also due to its specific optimizations for error detection circuits. Moreover, in the program context, OECF logical qubits are still the most reliable. As shown in Figure 6.6, the error rate of (six) test programs on OCSF, OLCF, and OEL layouts is on average 138.8%, 113.1%, and 46.0% higher than that on the OECF layout, respectively.

Secondly, error detection circuits latency is also the most important factor in the space-time overhead of quantum programs. Though the OECF logical qubit has the most physical qubits per logical qubit (see Table 6.3 Column 7), the smallest latency overhead of OECF's error detection (see Table 6.3 Column 8,9) still guarantees the smallest space-time overhead for quantum programs. As shown in Figure 6.6, the space-time overhead of test programs on OCSF, OLCF, and OEL layouts is on average 74.4%, 46.0% and 11.2% higher than that on the OECF layout, respectively. This is because each logical gate of a quantum program is followed by an error detection operation. This amplifies the effect of error detection latency on the whole program latency.

Thirdly, the optimization of logical CX is more important than the optimization of code switching. With similar error detection overhead, OCSF and OLCF separately show better code switching and logical CX than each other, due to their specific optimizations toward code switching and logical CX, respectively. Compared to OCSF, OLCF on average achieves 9.2% lower error rate and 15.8% lower space-time overhead for test programs (see Figure 6.6(a)(b)). This is because logical CX appears more frequently than code switching, making optimizing logical CX more advantageous. In quantum programs, each logical CX may induce some SWAP gates for routing and each SWAP gate will be decomposed into three logical CX, increasing the overall logical CX count. On the other hand, our compiler optimization AwareCS greatly reduces the usage of code switching in quantum programs. These two factors together result in 338% more logical CX than code switching (see Table 6.4 Column 4,14), averaged over test programs.

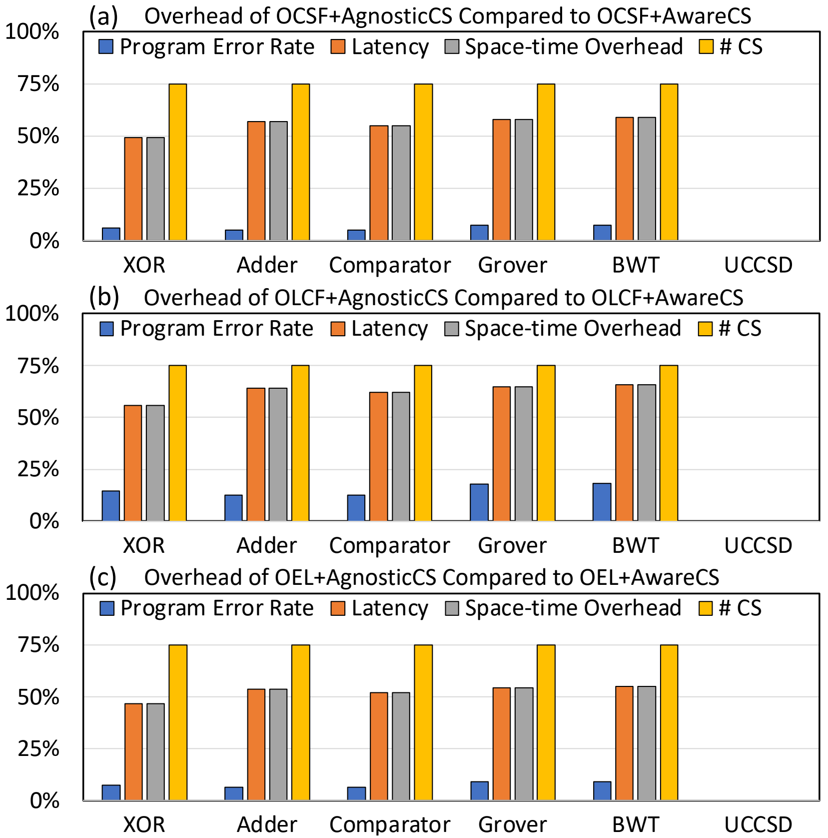


Figure 6.7: ('Overhead of AgnosticCS/overhead of AwareCS'*100%-1) on logical qubits generated by OCSF, OLCF and OEL.

Finally, incorporating QEC code information is critical for logical qubit design. Compared to OECF, OEL determines the data qubit layout according to the underlying hardware topology. Though we have tried to improve the logical qubit by OEL for better error detection, logical CX and code switching, the logical operations by OECF, especially the ones related to the RM code mode, are more reliable than those by OEL (see pseudo-thresholds in Table 6.3 Column 13-17). Moreover, as shown in Figure 6.6, compared to OECF, OEL on average increases the program error rate, latency, and space-time overhead of test programs by 46.0%, 17.0% and 11.2%, respectively. This indicates the importance of incorporating QEC code information into the logical qubit design.

2) The effect of code switching optimization

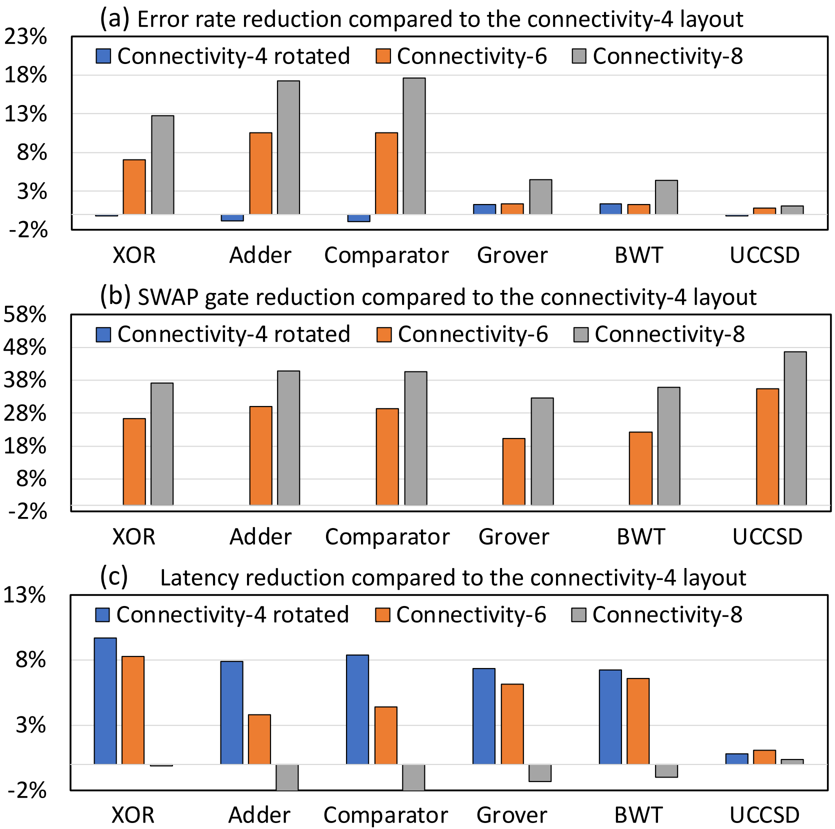


Figure 6.8: The effect of connectivity when placing multiple logical qubits generated by OECF.

Code switching count reduction will lead to space-time overhead reduction as code switching widely exists in test programs and is far more time-consuming than other logical operations (see Table 6.3 Column 8-12). Compared to 'OECF +AgnosticCS', 'OECF+AwareCS' on average reduces the space-time overhead and code switching count of test programs by 43.8% and 62.5%, respectively, as shown in Table 6.4. As for the fact that AwareCS does not show benefits on the UCCSD benchmark, it is because each T gate in UCCSD is followed by a H gate, providing no opportunities for code switching optimization. Fortunately, this is not a common pattern, especially in Toffoli-gate-based programs (e.g., Grover).

Further, the benefit of AwareCS is significant even when the logical qubit design

changes. As shown in Figure 6.7, compared to AgnosticCS, AwareCS on average reduces the space-time overhead of test programs on OCSF, OLCF, and OEL logical qubits by 46.4%, 52.1%, and 43.6%, respectively. This is because the logical qubit design will not change the occurrence of code switching in quantum programs and the latency of code switching is always far longer than other logical operations in different logical qubit designs (see Table 6.3 Column 8-12). The benefit of AwareCS will become even more remarkable when the 'logical connectivity' is higher. This is because the logical CX count of test programs on highly-connected architectures will be smaller, amplifying the effect of code switching.

3) Architecture-Compiler co-design

Firstly, we demonstrate that no layout is universally better. For hardware with constrained connectivity between physical qubits, enforcing higher connectivity between logical qubits does not necessarily induce a better computing platform. As shown in Figure 6.8(a), compared to the connectivity-4 setting (where logical qubits form a square grid), the connectivity-6 and connectivity-8 setting does improve the fidelity of quantum programs because they greatly reduce the SWAP gate count as shown in Figure 6.8(b). However, the connectivity-8 setting induces higher space-time overhead than the connectivity-4 setting, as shown in Figure 6.8(c). This is because logical CX along the diagonal direction hurts the parallelism of other logical CX gates and induces extra latency (see Table 6.2). Also, as shown in Figure 6.8(c), the connectivity-6 setting induces higher space-time overhead than the rotated connectivity-4 setting. This is because in the connectivity-6 setting, the horizontal logical CX takes longer time than the diagonal logical CX (see Table 6.2). Moreover, the lengthened GHZ paths of 'connectivity-4-rotated' may induces higher error rates than the connectivity-4 setting, for example on Xor, Adder, Comparator and UCCSD benchmarks, as shown in Figure 6.8(a). Overall, it is reasonable to match the connectivity of logical qubits with the connectivity of physical

qubits. Enforcing higher connectivity between logical qubits may not provide benefits to both fidelity and space-time overhead.

More importantly, we demonstrate more computational efficiency can be achieved by architecture-compiler co-designs. While it is not possible to achieve better performance on all programs by simply enforcing higher connectivity between logical qubits, it is possible to promote the performance of some specific quantum programs with the compiler output in Figure 6.8. For example, to achieve the smallest space-time overhead (i.e., latency overhead in this context) while allowing slightly ($< 2\%$) higher error rate of test programs than on the connectivity-4 layout, we can use the connectivity-6 layout for the UCCSD benchmark and 'Connectivity-4 rotated' layout for other test programs. Likewise, to achieve the lowest error rate while allowing slightly ($< 4\%$) higher space-time overhead of test programs than on the connectivity-4 layout, we can use the connectivity-6 layout for ripple-carry adder/comparator and connectivity-8 layout for other test programs. The ability of adjusting the QEC-CS architecture provides us the opportunity to co-design with the compiler to improve the performance of specific quantum programs.

4) Compared to distillation-based QEC scheme

We further compare 'OECF+AwareCS' to 'OECF+Distill'. 'OECF+Distill' uses our design for the Steane logical qubit (i.e., the half of the OECF logical qubit in Figure 6.5) but now exploits 15-to-1 magic state distillation [27] for implementing the logical T gate. We design the magic state distillation factory to be a 5*7 rectangle block of Steane logical qubits. Logical qubits along the boundaries of the factory can use the factory to implement the logical T. For the number of ancilla used in the factory, Jones et al. [115] indicated a 10:1 factory-to-data footprint to hide the latency of magic state distillation. In our work, we keep a 3:1 ancilla-to-data ratio for a good space-time balance. We place factories in a mesh grid with lines of logical qubits separating factories. For both

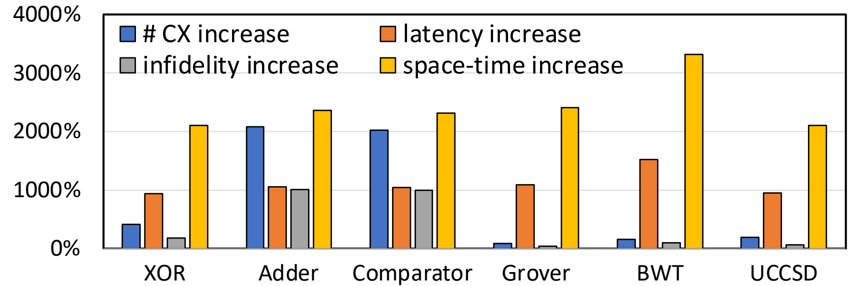


Figure 6.9: The increasing overhead of 'OECF+Distill' compared to 'OECF+AwareCS'.

'OECF+AwareCS' and 'OECF+Distill', we use the connectivity-4 setting.

As shown in Figure 6.9, 'OECF+AwareCS' is significantly better than 'OECF+Distill' over various benchmarks. The space-time overhead of 'OECF+Distill' is on average 24.3x higher than that of 'OECF+AwareCS'. This is because the 15-to-1 magic state distillation requires at least 31 ancillary Steane logical qubits and its time overhead is more than 25x longer than code switching, making the logical T by distillation less resource- and time-efficient than by code switching. Moreover, while the distilled magic state is very reliable, 'OECF+Distill' instead induces 4.2x higher overall error rate on test programs. This is because distillation factories occupy large device area and hinder the routing of logical CX, inducing 8.3x more logical CX gates on average.

6.4.3 Complexity Analysis

Scalability analysis: We first analyze the time/space complexity of our framework. For the architecture design in Sec. 6.3.1, the search process will repeatedly iterate over data qubits. For a distance-d color code, the data qubit count for code switching is at worst $O(d^3)$ [27]. Thus, the time complexity of the architecture search is $O(Kd^3)$ (K is the iteration count) and can be controlled by the user. Also, the space complexity of the search is $O(d^3)$ so as to store the optimized data qubit layout. For compiler designs, the code switching optimization in Sec. 6.3.2 performs a linear scan of gates of input

circuits. The time complexity of our compiler optimization is thus $O(N)$, assuming N is the gate count. The space complexity is also $O(N)$ so as to store the circuit. Overall, our framework is scalable to support higher-distance (color) code pairs.

Overhead scaling: We then analyze how the overhead of our QEC architecture design scales with the code distance. First, the latency of logical CX is at worst $O(d^2)$ (or $O(d^3)$ after code switching) when we need to do physical CX gates on data qubit pairs sequentially. By designing 3D connections [116] over data qubits, the latency of logical CX can be reduced to $O(1)$, by maximizing the parallelism of physical CX gates on data qubits. The error detection overhead of the color code is $O(1)$ as our stabilizer-shape-respected architecture design ensures the parallelism of the same-type (X-type or Z-type) stabilizer measurement. The code switching can be implemented within $O(d)$ QEC cycles [27] and its latency overhead is thus $O(d)$. On the other hand, the space overhead of each logical qubit in our framework is $O(d^3)$ as discussed above. Thus, the space-time overhead of logical single-qubit gates, error detection, and code switching is $O(d^3)$, $O(d^3)$ and $O(d^4)$, respectively. The space-time overhead of logical CX is dependent on the hardware support, and scales with $O(d^6)$ in the worst case, and $O(d^3)$ in the best case.

Chapter 7

Synthesizing Verified Quantum Operations

This chapter will dicuss about how to verify implementations of quantum operations protected by quantum error correction codes from quantum noise. The proposed verification framework can be used to guarantee the correctness of the synthesized error-corrected quantum operations.

7.1 Introduction

Quantum error correction (QEC) is a key enabler of fault-tolerant quantum computation (FTQC) on error-prone quantum hardware [1, 117, 118]. In particular, to achieve fault tolerance, QEC codes (QECC) will encode *logical qubits* — i.e., the quantum information to be protected — into many redundant physical qubits known as *data qubits* [20, 30]. QECC then utilizes *logical operations* — typically involving complex quantum circuits applied to data qubits — to manipulate the encoded information in logical qubits. These logical operations are crucial, as they enable the change of logical

qubit states (in short, logical states) without the need of the high-overhead steps of explicit decoding and re-encoding. This capability is essential for the pursuit of FTQC [1].

The critical role of logical operations in QECC further underscores the necessity for thorough correctness verification. In its absence, there is a risk that these operations could distort the logical state unexpectedly. Formally, in QECC, a logical operation F is considered correct if it reliably transforms any given input logical state into the targeted output state [1]. Building on this concept, we define "QECC correctness" wherein every logical operation within a universal logical gate set, such as the Clifford+T gate basis [1], adheres to this standard of correctness. This sufficiency stems from the property that a universal gate set, by definition, can construct any quantum operation [1, 20]. Hence, ensuring correctness for each logical operation in this set effectively guarantees the correctness of any logical operation in QECC.

Unfortunately, to our best knowledge, state-of-the-art quantum verifiers [119, 120, 121, 122, 123, 124, 125] are not able to effectively prove the correctness of QECC. The primary challenges in this context involves: 1) the encoding of a logical qubit requires a large number of data qubits, often in the thousands, according to the gap between the logical qubit error required by practical quantum applications [126] and the real physical qubit error of NISQ quantum hardware [20], and 2) the encoded logical qubit state is highly-entangled [1], which consists of exponentially many terms of quantum states over data qubits. For example, simulation-based techniques [124, 125] fail, as the sheer scale of data qubit count in the logical qubit state induces intractable exponential computational overhead. While other verification techniques [119, 120, 121, 122, 123] adopt symbolic reasoning to mitigate this overhead, they still cannot scale to practical QECCs due to the complexity of tracing logical states (which incur exponential memory overhead). Additionally, the complexity of logical operations exacerbates the situation, as their implementation demands a substantial number of low-level physical quantum

instructions. These instructions further result in an extensive sequence of verification steps.

We propose the first automated approach that can efficiently verify QECC correctness. Our *key insight* is to abstract both logical states and operations based on *stabilizer operators*. Traditionally, stabilizer operators in QECC are used to identify error detection mechanisms [1]: they define a valid quantum state subspace, flagging any logical state outside this subspace as an anomaly for detection. Instead, in our approach, we leverage a few stabilizer groups (each with n stabilizer operators) to uniquely determine any valid logical state in QECC that has $O(2^{n-1})$ terms [1], effectively circumventing the exponential memory overhead typically associated with representing logical qubit states. Moreover, we abstract a sequence of physical quantum gates, each altering some local physical state terms of a logical state, into a global transformation of stabilizer operators (which characterize the whole logical state). This abstraction considerably streamlines the complexity of QECC circuits for implementing logical operation and facilitates the scaling of the QECC verification.

While conceptually stabilizer operators provide a mechanism that simplifies the representation of logical states and operations, it is non-trivial to adapt them for efficient QECC verification. In what follows, we highlight a few key contributions of our work that bridge this gap.

Key contribution 1: a stabilizer-based language for QECC circuits. Given a QECC circuit C, we first transform C to a program P in a *stabilizer-based* language in a *semantics-preserving* fashion. Intuitively, this transformation process abbreviates multiple quantum gates from C to one operator in P, which by construction preserves the semantics. The key advantage of using higher-level operators, however, is to expose global program information that involves all data qubits, and focus the reasoning on the global transformation between two logical states, bypassing all intermediate transition

of the logical state induced by quantum gates local to individual data qubits. More specifically, our language supports abstracting error detection subroutines from C (typically corresponding to a few dozens of quantum gates) to stabilizer-based variables in P. Another class of language abstractions can compress a series of quantum gates from C (that sequentially manipulate local physical state terms of the logical state) to a global transformation of stabilizer operators. These new abstractions will be used to transform the pre/post-conditions as well, as explained next.

Key contribution 2: a stabilizer-based language for program states. Given a quantum circuit C and a pair of pre/post-conditions to be verified against, in addition to converting C to P in our stabilizer-based programming language, we also transform the pre/post to a stabilizer-based format. In general, we have a stabilizer-based language to express program states in P, where we utilize stabilizer operators to symbolize logical states. Our language considers stabilizer operators (defined by templates) whose subspaces contain the logical states from pre/post, as well as those from P. In other words, our language is defined by P and the pre/post. This induces a language that is both succinct and expressive for verification purposes in our experience.

Key contribution 3: an automated stabilizer-based verification algorithm. Now, given a program P and its pre/post, all in a stabilizer-based representation, we compute the weakest precondition for the given post and check its implication for the given pre. While overall standard, our algorithm is the first application of the Hoare-style reasoning to the *stabilizer-based* QECC verification. This novel combination enables us to verify large QEC codes, which prior work cannot reach.

Importance of automated verification. Our approach is push-button and fully automated — we believe this is necessary for QECC correctness verification, especially given the abundance of QEC codes that have been created in the past [127, 1, 30, 20] as well as to be invented in the future (e.g., by advanced artificial intelligence [128] or search

techniques [129]). It is thus critical to develop automated techniques to scale verification to the escalating volume of QEC codes.

Evaluation. We have implemented our proposed technique in a tool, called VERITA, and use it to verify commonly used QEC codes. Our experimental results show that, VERITA can successfully verify significantly more benchmarks than state-of-the-art quantum verifiers, within much less time. Our approach also scales better: for example, while prior work can verify the surface code with up to 25 data qubits within 3 hours, VERITA can scale to more than 5,000 data qubits using the same amount of time. We have also performed extensive ablation studies that demonstrate the importance of each component of our technique.

In summary, we makes the following contributions.

- We propose a new stabilizer-based programming language for QECC circuits, which allows a higher-level reasoning with our concise stabilizer-based representations for groups of quantum gates.
- We propose a new stabilizer-based language for QECC logical states, which avoids exponential memory overhead and allows direct utilization of high-level program information.
- We propose the first automated stabilizer-based verification algorithm for QEC codes.
- We implement our proposed techniques in a new tool, called VERITA. Our evaluation results highlight the stabilizer-based abstraction yields a significantly faster verifier.

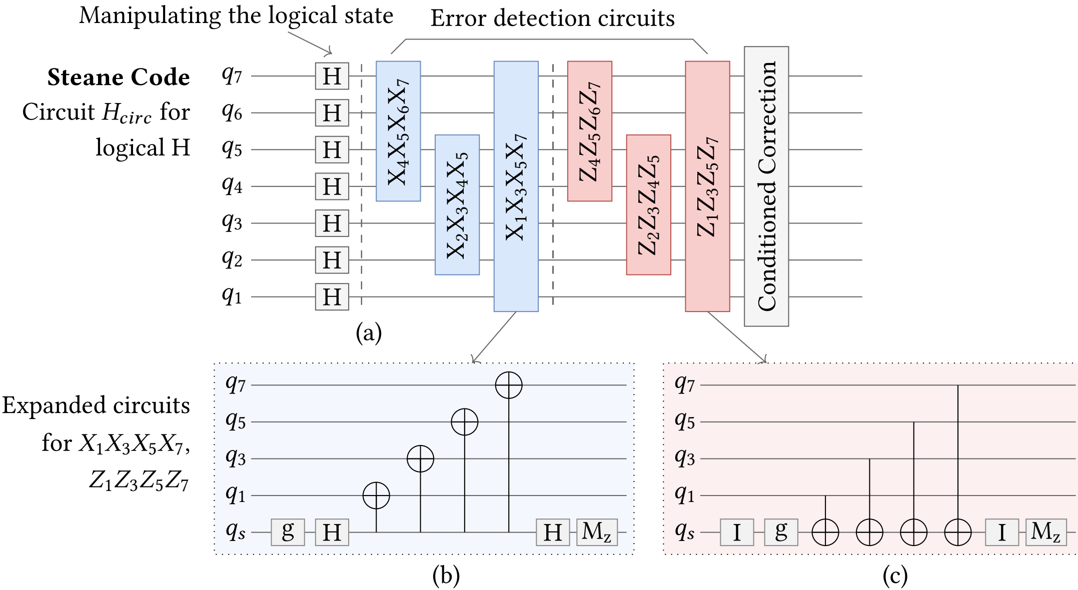


Figure 7.1: An example program of the Steane code which implements the logical H on the encoded logical qubit. Pauli strings in (a), e.g., $X_1X_3X_5X_7$ and $Z_1Z_3Z_5Z_7$, represents error detection circuits shown in (b)(c). Physical operations g, H, I, M_z refers to reset, Hadamard gate, identity gate, and Z-type measurement, respectively.

7.2 Motivating Example

Logical Operation on QECC. The quantum diagram in Figure 7.1(a) illustrates the circuit implementation of a logical H gate on a logical qubit, which is encoded using the Steane code [1] across seven data qubits, $q_1, \ldots, q_7$. For ease of reference, we refer to the circuit implementing a logical operation as a *QECC circuit*.

A QECC circuit typically consists of two distinct types of subroutines. The first type of subroutine is responsible for error detection. For the Steane code in Figure 7.1(a), the six *stabilizer operators*, which are a string/tensor product of Pauli operators like $X_1X_3X_5X_7$, each represents an error detection circuit (aka *stabilizer measurement circuit* [1]). For example, Figure 7.1(b) depicts the error detection circuit denoted by $X_1X_3X_5X_7$ and on q_1, q_3, q_5, q_7. These circuits are indispensable in any QECC circuit as they detect errors that may occur during the execution. For example, considering a Pauli

Z error on q_1 before the measurement circuit of $X_1X_3X_5X_7$, this Pauli Z error, through the gate $CX\ q_s, q_1$, will flip the measurement outcome of q_s, indicating the presence of certain errors. Afterward, depending on the error detection outcome, error correction operations will be applied to eliminate the detected error, e.g., a quantum Z gate will be applied on q_1 for a detected Pauli Z error on q_1.

The second type of subroutine is tasked with manipulating the logical state. For example, the initial seven qubit-wise H gates in Figure 7.1(a) perform the Hadamard transformation on the logical qubit encoded by the Steane code, i.e., they will transform the logical zero state $|0_L\rangle$ to the logical magic state $|+_L\rangle$. We remark this qubit-wise implementation of the logical H is specific to the Steane code. For other QECCs, the implementation varies. For example, for the Reed-Muller code [126], the implementation of the logical H relies on the complicated magic state distillation [126], making it even harder to show the correctness of the circuit. Here for the demonstration purpose, we focus on the verification of the logical H of the Steane code.

QECC correctness. Establishing QECC correctness can be achieved by verifying the correctness of logical operations in the logical version of the Clifford+T gate set, which can be used to achieve any unitary transformation and encompasses logical X, Z, H, S, T, and CX gates, along with logical initialization. To prove the correctness of a logical operation, we need to show that this operation does transform an arbitrary logical state to the desired logical state, depending on the specific logical operation examined. For example, concerning the logical H circuit H_{circ} in Figure 7.1(a), we need to prove it implements the Hadamard transformation, i.e., transforms $|+_L\rangle$ to $|0_L\rangle$ and $|0_L\rangle$ to $|+_L\rangle$, where braket notations with subscript L denote logical states. Formally, we are to prove

$$\models \{|0_L\rangle\}H_{circ}\{|+_L\rangle\}, \qquad \models \{|+_L\rangle\}H_{circ}\{|0_L\rangle\} \tag{7.1}$$

This desired input-output behaviors is similar to the physical H gate, which requires $\models \{|+\rangle\}H\{|0\rangle\}$, $\models \{|-\rangle\}H\{|1\rangle\}$.

Intuitively, we may think, since one H gate can be easily shown to achieve the desired Hadamard transformation on one data qubit, H gates on all data qubits should be obviously proved to achieve the Hadamard transformation on the logical qubit. Unfortunately, this intuition does not hold for the complex QECC circuit, as explained next.

Challenges. The challenges of proving QEC correctness originates from the complexity of logical states and error detection. First, the logical state encoded by QECC is increasingly complex as the QECC size grows. For a QECC circuit with n data qubits, the logical state may consist of $O(\sqrt{2}^{n-1})$ terms of n-qubit physical quantum states. For example, the $|0_L\rangle$ encoded by the Steane code involves 8 seven-qubit physical states:

$$\begin{aligned} |0_L\rangle := &\frac{1}{4}|0000000\rangle + \frac{1}{4}|0001111\rangle + \frac{1}{4}|0111100\rangle + \frac{1}{4}|0110011\rangle \\ &\frac{1}{4}|1010101\rangle + \frac{1}{4}|1011010\rangle + \frac{1}{4}|1101001\rangle + \frac{1}{4}|1100110\rangle \end{aligned} \tag{7.2}$$

While we can still verify the small QECC circuit in Figure 7.1(a) with existing quantum verification frameworks, the exponentially complex logical state for a QECC with more than 100 data qubits will cause out of memory error with those approaches since they directly reason over quantum states and there is no efficient way in the literature to reduce the the memory overhead.

In addition, the error detection part of QECC circuits will significantly expand the reasoning flow. As shown in Figure 7.1(a), the size of error detection circuits (48 physical operations induced by 6 stabilizer operators) is far greater than the size of codes for manipulating the logical qubit (7 physical H gates). The discrepancy holds as QECC scales up. For example, considering the surface code with n data qubits, the error detection cir-

```
0  H_prog :=
1  q0q1q2q3q4q5q6q7 := ⊗_{i=1}^{7} H_i q0q1q2q3q4q5q6q7;
2  ∫1 := X4X5X6X7;
3  ∫2 := X2X3X4X5;
4  ∫3 := X1X3X5X7;
5  ∫4 := Z4Z5Z6Z7;
6  ∫5 := Z2Z3Z4Z5;
7  ∫6 := Z1Z3Z5Z7;
8  correct(∫1, ∫2, ..., ∫6);
```

(a)

```
0  correct(∫1, ∫2, ..., ∫6) :=
1  if M[∫3, q0q1q2q3q4q5q6q7]
   then
2      skip;
   else
3      q1 = Z1 q1;
   end
```

(b)

Figure 7.2: (a) The logical H program in *Veri-Lang*, translated from Figure 7.1(a). (b) The simplified implementation of the *correct* function, which can only correct the Pauli Z error on q_1.

cuit consists of $8(n-1)$ physical quantum operations while fulfilling the Hadamard transformation on the logical qubit only requires n quantum operations. Failing to simplify the reasoning related to these error detection circuits will induce substantial reasoning overhead.

Now let us explain how VERITA can efficiently verify the QECC circuit in Figure 7.1(a).

Step 1: Convert circuit to program. We translate QECC circuit in Figure 7.1(a) to the program shown in Figure 7.2(a). In the translated program, we utilize three types of abstractions to reduce the program size. The first type is the stabilizer-based assignment statement (see line 2-7), which is used to represent error detection circuits. For example, we use $\int_3 = X_1X_3X_5X_7$ to model the behavior of the circuit shown in Figure 7.1(b). This translation is straightforward, since error detection circuits like those shown in Figure 7.1(b)(c) have clear patterns: they have a parity qubit q_s which use CX gates to interact with data qubits and will be later measured to report the existence of errors. We can use template matching to convert error detection circuits into stabilizer-based assignment statements shown in Figure 7.2(a). In the semantics of our language,

$\int_3 = X_1X_3X_5X_7$ has exactly the same meaning as the circuit in Figure 7.1(b). The only difference is that our statement explicitly points out the error detection circuit in Figure 7.1(b) will project the quantum state over data qubits into the quantum state subspace of the stabilizer operator $X_1X_3X_5X_7$.

The second type is for aggregating quantum gates. We translate the individual H gates in Figure 7.1(a) into an aggregated instruction (line 1) in Figure 7.2(a). This translation can also be directly completed by template matching: we will create an aggregated instruction if we see a group of quantum gates that manipulate all data qubits in a sequential way. There is no difference in semantics whether we write the seven H gates in seven lines or write them in one line and in an aggregated style. The benefit of this abstraction is that, it reveals the global transformation on the logical qubit, which will shorten our reasoning (since fewer sentences are considered).

The third type of abstraction is to use the statement $correct(\int_1, \int_2, \ldots, \int_6)$ to abstract the error correction operation, e.g. the simple one shown in Figure 7.2(b). The semantics of the error correction operation has a clear pattern: the illegal logical state will be recovered to be a legal logical state, i.e., to be into the intersected subspace of all error detection-related stabilizer operators. Our abstraction $correct(\int_1, \int_2, \ldots, \int_6)$ is exactly designed to convey the same semantics. Since our work is focused on demonstrating the correctness of logical operations, using this abstraction can rescue us from diving into the details of the error correction operation.

Step 2: Convert pre and postconditions. While our programming language reduces the code size, we still need an efficient representation for assertions to avoid the exponential memory overhead induced by the highly-entangled logical state. In general, we can use n stabilizer operators to uniquely symbolize the logical state of QECC with n data qubits, thus avoiding exponential terms of physical states. Particularly, for QECC with n data qubits, we propose to translate the state-based assertions to the conjunction

of a set of n stabilizer operators, which means the logical state in the assertion should be in the mutual quantum state subspace of those stabilizer operators.

We determine the set of stabilizer operators with two steps. The first step is to extract error detection-related stabilizer operators from the QEC program translated by Step 1. For example, from Figure 7.2(a), we obtain six stabilizer operators that are assigned to $\int_1, \ldots, \int_6$. These stabilizer operators will serve as the major part of the translated assertion. According to the definition of QECC, the logical state must be in the intersection of subspaces represented by those error detection-related stabilizer operators [1].

In the second step, we will search for additional stabilizer operators from predefined templates. For example, for the $|0_L\rangle$ of the Steane code, since it belongs the subspace represented by the stabilizer operator $\otimes_{i=1}^{7} Z_i$ (i.e., $Z_1 Z_2 \cdots Z_7$), we will pick this stabilizer operator as the remaining part of the translated assertion.

Thus, we can simplify the assertion based on $|0_L\rangle$ of the Steane code with seven stabilizer operators as follows:

$$
\begin{aligned}
|0_L\rangle \quad \Longrightarrow \quad & Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7 \\
& \wedge X_4 X_5 X_6 X_7 \quad \wedge X_2 X_3 X_4 X_5 \quad \wedge X_1 X_3 X_5 X_7 \\
& \wedge Z_4 Z_5 Z_6 Z_7 \quad \wedge Z_2 Z_3 Z_4 Z_5 \quad \wedge Z_1 Z_3 Z_5 Z_7
\end{aligned}
\tag{7.3}
$$

Note that the conjunction means that, the $|0_L\rangle$ is in the intersection of subspaces represented by stabilizer operations on the right hand size of Equ 7.3. Overall, the translation from state-based assertion to the stabilizer-based assertion is cheap since the error detection specification of QECC already provide enough information for the translation. For the demonstration purpose, the translated assertion in Equ 7.3 only consists of singular stabilizer operators. In Section 7.3.2, we will discuss about more complicated cases, where

the element of the conjunction can be an arithmetic expression over stabilizer operators.

Step 3: Weakest precondition generation. With our stabilizer-based language and assertion, we can now use existing machinery to prove the correctness of QEC programs. We specifically design new inference rules based on our language and assertion (see Figure 7.5 in Section 7.4). These inference rules explicitly utilize the high-level program information exposed by stabilizer-based representations. As an example, with the weakest precondition computation (WPC), we can verify the goal $|0_L\rangle \vdash H_{prog} : |+_L\rangle$ for H_{prog} in Figure 7.2, where $|0_L\rangle$ and $|+_L\rangle$ are the post and pre and are already translated by **Step 2**, like in Equ 7.3. For simplicity, we let $Z_L := \otimes_{i=1}^{7} Z_i$, $X_L := \otimes_{i=1}^{7} X_i$, $H_L := \otimes_{i=1}^{7} H_i$, A_S be the conjunction of all the six error detection-related stabilizer operators except $X_1X_3X_5X_7$ in Figure 7.2(a). Then we provide the sketch of the WPC as follows:

$$Z_L \wedge X_1X_3X_5X_7 \wedge A_S \vdash \ correct(\int_1, \int_2, \dots, \int_6) \ : Z_L \cdot X_1X_3X_5X_7 \wedge A_S \tag{7.4}$$

$$Z_L \cdot X_1X_3X_5X_7 \wedge A_S \vdash \ \int_6 = Z_1Z_3Z_5Z_7 \ : Z_L \cdot X_1X_3X_5X_7 \wedge A_S$$

... // omit some steps for simplicity

$$Z_L \cdot X_1X_3X_5X_7 \wedge A_S \vdash \ \int_1 = X_4X_5X_6X_7 \ : Z_L \cdot X_1X_3X_5X_7 \wedge A_S \tag{7.5}$$

$$Z_L \cdot X_1X_3X_5X_7 \wedge A_S \vdash \ q_1q_2q_3q_4q_5q_6q_7 = H_L q_1 \dots q_7 : X_L \cdot Z_1Z_3Z_5Z_7 \wedge (\wedge_{i \neq 6} \int_i) \tag{7.6}$$

Note that, $X_L \cdot Z_1Z_3Z_5Z_7$ means the symbolic multiplication of the (tensor-product) matrices represented by X_L and $Z_1Z_3Z_5Z_7$. Since $|+_L\rangle$ is in state subspaces represented by X_L (according to **Step 2**) and $Z_1Z_3Z_5Z_7$, it must be in the subspace (i.e., $+1$ eigenspace) of $X_L \cdot Z_1Z_3Z_5Z_7$ (see the inference rule in Section 7.3.2 Lemma 7.3.4), this completes the proof. This inference process involves stabilizer-based rules, which do not have counterparts in existing quantum verifiers. For example, Equ 7.4 and Equ 7.5 require Rule 7.18,

7.19 and 7.15 in Section 7.4 Figure 7.5. Notice that the **correct** in Equ 7.4 refers to the simplified one in Figure 7.2(b), rather than the general one in Rule 7.20 in Figure 7.5.

Overall, for the QECC circuit in Figure 7.1(a), with our approach, 8 statements and at most 7 terms of stabilizer operators are involved in the verification process. As a comparison, 106 statements and up to 128 terms of physical states over data qubits will be involved when reasoning with existing approaches. This demonstrates the great efficiency of our framework in verifying QECC circuits.

7.3 Programming Language Designs for QEC

In this section, we introduce our stabilizer-based language designs for QECC-based programs and corresponding assertions, which enables us to reason over stabilizer-based high-level structural information of QECC.

7.3.1 *Veri-Lang*: The QECC Programming

We define the notation for quantum variables as follows: Define qVar as the set of quantum variables, q as a metavariable ranging over quantum variables, and $\bar{q}$ to be a quantum register associated with a finite set of distinct quantum variables. We denote the state space of q by $\mathcal{H}_q$ which is a two-dimensional Hilbert space spanned by the computational basis states $\{|0\rangle, |1\rangle\}$. The state space of $\bar{q}$ is the tensor product of Hilbert spaces $\mathcal{H}_{\bar{q}} = \otimes_{q\in\bar{q}}\mathcal{H}_q$.

Logical operations of QECC are often associated with error detection circuits. For example, the surface code [20] frequently turns on and turns off specific stabilizer measurement circuits to implement logical operations. Besides, the outcomes of stabilizer measurements act as signals for error correction. By introducing a stabilizer variable, which represents a stabilizer measurement circuit without the need of specifying its

physical implementation, we can greatly simplify the description of QEC operations. Specifically, we use the stabilizer operator to define the value of stabilizer variables.

We define the stabilizer operator as follows: a stabilizer operator refers to a tensor product of Pauli operators on data qubits and is used to describe to the error detection circuit in the QEC context. For example, $X_1X_3X_5X_7$ in Figure 7.1(a) is a stabilizer operator and can be used to describe the X-type error detection circuit on data qubits q_1, q_3, q_5, q_7 (see Figure 7.1(b)). The stabilizer operator can also be thought as a symbolic Hermitian matrix, whose $+1$ eigenspace defines a quantum state subspace. For example, $X_1X_3X_5X_7$ denotes the matrix $X_1 \otimes I_2 \otimes X_3 \otimes I_4 \otimes X_5 \otimes I_6 \otimes X_7$, and the 'Steane code' logical state $|0_L\rangle$ is in the $+1$ eigenspace of this matrix/stabilizer operator. One more intuition is that, for the error detection circuit Figure 7.1(b), if the measurement outcome of q_s is $+1$, the resulting physical state over data qubits will be in the $+1$ eigenspace of $X_1X_3X_5X_7$.

Then, we define the notations for the stabilizer variable as follows: define S as the set of stabilizer operators on qVar, s as an individual stabilizer operator in S, sVar as the set of stabilizer variables, and $\int$ as a metavariable ranging over sVar. To make S countable, we assume that every $s \in S$ only involves a finite number of qubits. The range of values for the stabilizer variable $\int$ is $S \cup -S \cup iS \cup -iS$, where i is the imaginary unit.

We define the syntax of *Veri-Lang* as follows:

$$\begin{aligned}
\text{Prog} ::= \ & \textbf{skip} \mid \bar{q} := \otimes_i|0\rangle \mid \bar{q} := U[\bar{q}] \mid \int := s_e^u \\
& \mid \text{Prog}_1; \text{Prog}_2 \\
& \mid \textbf{if } M[\int, \bar{q}] \textbf{ then } \text{Prog}_1 \textbf{ else } \text{Prog}_0 \textbf{ end} \\
s_e^u ::= \ & \pm s \mid \ \pm i\, s
\end{aligned} \tag{7.7}$$

The proposed language constructs consisting of instructions as follows: (1) **skip** does

nothing; (2) $\bar{q} := \otimes_i|0\rangle$ resets quantum register $\bar{q}$ to ground state $\otimes_i|0\rangle$; (3) $\bar{q} := U[\bar{q}]$ perform unitary operation U on quantum register $\bar{q}$; (4) $\int := s_e^u$ assigns a unary stabilizer expression s_e^u to the stabilizer variable $\int$; (5) $\text{Prog}_1; \text{Prog}_2$ is the sequencing of programs; (6) **if** $M[\int, \bar{q}]$ **then** Prog_1 **else** Prog_0 **end** perform the error detection circuit represented by $\int$ on qubits $\bar{q}$ (or in short, measures $\int$ on qubits $\bar{q}$) and executes program Prog_1 if the measurement outcome is $+1$. Otherwise Prog_0 is executed.

The language constructs above are similar to those of the quantum **while**-language [120], except the part associated with stabilizer variables. In our design, we propose the stabilizer variable to expose the high-level stabilizer measurement information. For example in Figure 7.1, we use the stabilizer variable $\int_3 = X_1X_3X_5X_7$ to represent the stabilizer measurement circuit shown in Figure 7.1(b), which consists of a series of 1- and 2-qubit gates and measurements. This abstraction not only reduces the program size but also lay a foundation for easy application of Hoare logic discussed in Section 7.3.2. Further, with stabilizer variables, *Veri-Lang* avoids the implementation details of stabilizer measurement circuits. This makes *Veri-Lang* programs very flexible and independent from the specific implementation of stabilizer measurements. The latter may, for example, depend on architectural properties like the underlying hardware connectivity [30], while the correctness of the QEC operations should not be affected the implementation of stabilizer measurement circuits.

For programming, stabilizer variables can be used to describe operations on stabilizers. For example, to turn off one stabilizer measurement circuit in a QEC program, we can simply set $\int = \text{I}$. The stabilizer variable can also serve to inform the error correction procedure. Every time we detect one or more stabilizer variables with a negative sign (this may happen after a stabilizer measurement), the error decoder knows that at least one error affected the physical circuit. It proceeds to identify the specific error and applies the corresponding correction. Further, we abstract the error correction protocol

as a function over stabilizer variables as follows:

Definition 7.3.1 (Error correction protocol)**.** *Define* ***correct***$(\int_0, \int_1, \cdots, \int_n)$ *as an error decoding and correction protocol by measuring* $\int_0, \int_1, \cdots, \int_n$*.*

The **correct** function can essentially be implemented by a series of condition statement, with measurement results from stabilizer variables.

7.3.2 *Veri-Assn*: The Assertion Language

A stabilizer operator is a Hermitian matrix and can be used as predicate for QEC programs. This observation is particularly important for QEC programs in which the majority of logical operations can be described with a few stabilizer operators. Using stabilizer operators as predicates can circumvent the exponential memory overhead induced by quantum state-based predicate, which has a exponential number of physical state terms, regarding the data qubit count.

However, as predicates, stabilizers are not universal. There are infinitely many quantum states that are not eigenstates of any non-identity stabilizer operator, e.g., $|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$. Such limitation will cause difficulty in the verification of QEC programs. For example, if we are given some logical state that is not the $+1$ eigenstate of any stabilizer operator, we cannot find any predicate except I to abstract such logical state. One well-studied way in the quantum information community to address this problem is to use the Pauli expansion of quantum Observable (Hermitian matrices) [1, 130]:

Lemma 7.3.2 (Pauli expansion)**.** *The quantum observable O of a n-qubit system can be expressed as a linear combination of Pauli strings: $O = \sum_i w^i \sigma_n^i$, where $\sigma_n^i \in \{I, X, Y, Z\}^{\otimes n}$ is a length-n Pauli string, and $w_i \in \mathbb{R}$ is its coefficient.*

The Pauli expansion motivates the following proposition which provides a universal way to deal with arbitrary logical states in QEC programs:

Proposition 7.3.3. *$\forall |\psi\rangle \in \mathcal{H}_n$, there is a P which is a sum of stabilizers (Pauli strings), that satisfies $P|\psi\rangle = |\psi\rangle$, and $P \neq I$.*

Proposition 7.3.3 can be directly proved by setting $P = |\psi\rangle\langle\psi|$ and expand it into a linear combination of Pauli strings, i.e., stabilizer operators. Inspired by Lemma 7.3.2 and Proposition 7.3.3, we introduce arithmetic expressions of stabilizer operators, and define the stabilizer expression s_e as follows,

$$s_e ::= s \mid \lambda_0 s_{e0} + \lambda_1 s_{e1},\ \ \lambda_0, \lambda_1 \in \mathbb{C} \tag{7.8}$$

where s is a stabilizer operator. s_e is different from the s_e^u used for stabilizer variables which only consists of unary operations on stabilizer operators. However, by describing both the stabilizer variable and the predicate within the stabilizer-based language, we can easily incorporate the information from stabilizer measurement into predicates.

By Proposition 7.3.3, s_e is universal as $\forall |\psi\rangle, \exists s_e\ s.t.\ s_e|\psi\rangle = |\psi\rangle$. For example, the state $|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$ is a $+1$ eigenstate of $s_e \coloneqq \frac{1}{2}Z + \frac{\sqrt{3}}{2}X$. We then formulate the assertion language ***Veri-Assn*** on QEC programs as follows:

$$A ::= s_e \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid A_0 \Rightarrow A_1. \tag{7.9}$$

When $A \coloneqq s_e$, we say a QEC program state (ρ, σ) satisfies an assertion A if $A\rho = \rho$ (for $\rho = |\psi\rangle\langle\psi|$, $A\rho = \rho \Leftrightarrow A|\psi\rangle = |\psi\rangle$), and s_e commutes with all stabilizer variables in σ. We denote this relation by $(\rho, \sigma) \models A$. Requiring that s_e commutes with stabilizer variables in σ is to ensure the logical state can be in the $+1$ eigenspace of s_e. Otherwise this assertion never be satisfied.

The semantics of $A_0 \wedge A_1$ and other Boolean expressions can then be derived by structural induction:

- $(\rho, \sigma) \models A_1 \wedge A_2$ iff $(\rho, \sigma) \models A_1$ and $(\rho, \sigma) \models A_2$;
- $(\rho, \sigma) \models A_1 \vee A_2$ iff $(\rho, \sigma) \models A_1$ or $(\rho, \sigma) \models A_2$;
- $(\rho, \sigma) \models (A_1 \Rightarrow A_2)$ iff $((\rho, \sigma) \models A_1) \Rightarrow ((\rho, \sigma) \models A_2)$.

If an assertion A is satisfied by all program states (ρ, σ), we denote such property as $\models A$.

The following lemma presents stabilizer-based implicit rules.

Lemma 7.3.4 (Implication rule)**.** *For stabilizer expressions,*

1. *If* $(\rho, \sigma) \models s_{e0}$ *and* $(\rho, \sigma) \models s_{e1}$*, we have* $(\rho, \sigma) \models s_{e0}s_{e1}$ *and* $(\rho, \sigma) \models \lambda_0 s_{e0} + \lambda_1 s_{e1}$*, where* $|\lambda_0|^2 + |\lambda_1|^2 = 1$*.*
2. *Assume* s_{e0} *is not singular if we view it as a matrix. If* $(\rho, \sigma) \models s_{e0}$ *and* $(\rho, \sigma) \models s_{e1}s_{e0}$*, we have* $(\rho, \sigma) \models s_{e1}$*.*
3. *Assume* $(as_{e0} + bs_{e1})\rho = \rho$*, every stabilizer in* σ *commutes with* s_{e0} *and* s_{e1}*, and* $(\rho, \sigma) \models s_{e2}$*, then* $(\rho, \sigma) \models as_{e0} + bs_{e1}s_{e2}$*.*

Proof. This lemma can be proved by examining the definition of our proposed assertions. □

Rules from classical Boolean predicates can also be used for *Veri-Assn*, such as the rules for disjunction and conjunction. We will use these rules directly without extra description. Especially, the identity operator I represents **True** and the empty operator 0 represents **False** in *Veri-Assn*.

7.3.3 QECC Correctness

We use Hoare logic to formalize the QECC correctness. A Hoare tripe in *Veri-Assn* has the form: $\{A\}c\{B\}$, where $A, B \in$ *Veri-Assn*, and $c \in$ *Veri-Lang*. We first

$\{A\}\mathbf{skip}\{A\}$ (Skip)

$\{A[|0\rangle/q]\}q := |0\rangle\{A\}$ (Initialization)

$\{A\}\bar{q} := U\bar{q}\{UAU^\dagger\}$ (Unitary)

U is a unitary, but written in the sum of stabilizers.

$\{A\}\int := s_e^u\{A\}$ (Assignment)

where s_e^u commutes with A. Otherwise, $\{A\}\int := s_e^u\{I\}$.

$$\frac{\{A\}\text{Prog}_1;\text{Prog}_2\{B\}}{\{A\}\text{Prog}_1\{C\} \quad \{C\}\text{Prog}_2\{B\}}$$ (Sequencing)

$$\frac{\{\sum_{i=0}^{1} A_i M_i\}\mathbf{if}\, M[\int,\bar{q}]\,\mathbf{then}\,\text{Prog}_1\,\mathbf{else}\,\text{Prog}_0\,\mathbf{end}\{B\}}{\{A_1 \wedge \int\}\text{Prog}_1\{B\} \quad \{A_0 \wedge -\int\}\text{Prog}_0\{B\}}$$ (Condition)

$$\frac{\{A\}\text{Prog}\{B\}}{\models (A \Rightarrow A') \quad \{A'\}\text{Prog}\{B'\} \quad \models (B' \Rightarrow B)}$$ (Consequence)

Figure 7.3: Hoare rules for QECC correctness when $A := s_e$.

present the Hoare logic for partial correctness assertions in which the precondition A is a stabilizer expression s_e, as shown in Figure 7.3. We will extend the Hoare logic to Boolean expressions like $A_1 \wedge A_2$ in Proposition 7.3.9.

The proof rules in Figure 7.3 are syntax-directed and reduce proving a partial correctness assertion of a compound statement to proving partial correctness assertions of its sub-statements. We only explain some rules below, since most rules are self-explained.

In the initialization rule, $(\rho, \sigma) \models A[|0\rangle/q]$ means that $A\rho_0^q = \rho_0^q$ and A commutes with all stabilizer variables in σ. This can be seen as the quantum version substitution rule. A more useful case of the initialization rule is when all qubits are reset to $|0\rangle$, and for the n-qubit system, we have

$$\{I\}q_{n-1}\cdots q_0 := |0\rangle^{\otimes n}\{Z_0 \wedge Z_1 \wedge \cdots \wedge Z_n\}. \tag{7.10}$$

The following example shows the partial correctness on the initialization rule.

Example 7.3.5 (Initialization rule). *Let* $A := Z_0Z_1, \rho = |10\rangle\langle 10|, \sigma = \{\}$, *for initialization* $q_0 := |0\rangle$, $(\rho, \sigma) \models A[|0\rangle/q]$ *since* $(\rho_0^{q_0}, \sigma) = (|00\rangle\langle 00|, \{\}) \models Z_0Z_1$. *Then,*

$(\rho', \sigma) = [\![q_0 := |0\rangle]\!](\rho, \sigma) = (|00\rangle\langle 00|, \{\})$ *also satisfies* $Z_0 Z_1$.

For the assignment rule, $(\rho, \sigma) \models A[s_e^u/\int]$ means that $A\rho A^\dagger = \rho$ and A commutes with all stabilizer variables in $\sigma[s_e^u/\int]$. The following example shows the partial correctness on the assignment rule.

Example 7.3.6 (Assignment rule)**.** *Let* $A := Z, \rho = |0\rangle\langle 0|, \sigma = \{\int = X\}$*, for assignment* $\int := Z$*,* $(\rho, \sigma) \models A[Z/\int]$ *since* $(\rho, \sigma[Z/\int]) = (|0\rangle\langle 0|, \{Z\}) \models Z$*. Then,* $(\rho, \sigma') = [\![\int := Z]\!](\rho, \sigma) = (|0\rangle\langle 0|, \{Z\})$ *also satisfies* Z*.*

In the unitary rule, we represent unitary matrices as the sum of stabilizers in order to utilize the cheap computational cost of stabilizer multiplication.

The rules for condition and while loop resembles their classical counterparts except the state may be changed by the branching condition. A direct derivative of the Condition rule is to make $A_1 = A_0 = A$ as follows,

Lemma 7.3.7. $\dfrac{\{A \wedge \int\}Prog_1\{B\} \quad \{A \wedge -\int\}Prog_0\{B\}}{\{A\}\textbf{\textit{if}}\, M[\int, \bar{q}]\ \textbf{\textit{then}}\, Prog_1\ \textbf{\textit{else}}\, Prog_0\ \textbf{\textit{end}}\{B\}}$.

The consequence rule is a powerful tool for the verification of QEC programs since it can encode facts of QEC codes into partial correctness assertions. The following example demonstrates the usage of the proposed Hoare rules, including the consequence rule:

Example 7.3.8. *Assume* $Prog ::= \int := Z_1;$ ***if*** $M[\int, q_1]$ ***then skip else*** $q_1 := X\ q_1; q_0 := X\ q_0$ ***end****.*

We prove $\{Z_0 Z_1\}Prog\{Z_0\}$ *as follows: (for simplicity, we perform analysis sentence by sentence)*

$\{Z_0 Z_1\}\int := Z_1; \{Z_0 Z_1\}$ *(Assignment)*

$(Z_0 Z_1)M_1 = (Z_0 Z_1)\frac{I+Z_1}{2} = Z_0\frac{I+Z_1}{2}$, $(Z_0 Z_1)M_0 = -Z_0\frac{I-Z_1}{2}$

$\{Z_0 Z_1 \wedge Z_1\}\textbf{\textit{skip}}\{Z_0 Z_1 \wedge Z_1\}$ *(Skip)*

$\{Z_0 Z_1 \wedge -Z_1\}q_1 := X\ q_1\{-Z_0 Z_1 \wedge Z_1\}$ *(Unitary)*

$\{-Z_0Z_1 \wedge Z_1\}q_0 \coloneqq X\ q_0\{Z_0Z_1 \wedge Z_1\}$ *(Unitary)*

$\{Z_0Z_1 \wedge -Z_1\}q_1 \coloneqq X\ q_1; q_0 \coloneqq X\ q_0\{Z_0Z_1 \wedge Z_1\}$ *(Sequencing)*

$\{Z_0Z_1 = (Z_0Z_1)M_0 + (Z_0Z_1)M_1\}$***if*** $M[\int, \bar{q}]$ ***then skip***

else $q_1 \coloneqq X\ q_1; q_0 \coloneqq X\ q_0$ ***end*** $\{Z_0Z_1 \wedge Z_1\}$ *(Condition)*

Then, $\{Z_0Z_1\}Prog\{Z_0Z_1 \wedge Z_1\}$ *(Sequencing)*

$Z_0Z_1 \wedge Z_1 \Rightarrow Z_0$ *(Implication)*

With the consequence rule, we have $\{Z_0Z_1\}Prog\{Z_0\}$.

Now we extend the Hoare rules in Figure 7.3 to other Boolean assertions in *Veri-Assn.*

Proposition 7.3.9. *We restate the Hoare rules for classical Boolean assertions as follows,*

if $\{A_0\}Prog\{B_0\} \wedge \{A_1\}Prog\{B_1\}$, *then* $\{A_0 \wedge A_1\}Prog\{B_0 \wedge B_1\}$;

if $\{A_0\}Prog\{B_0\} \vee \{A_1\}Prog\{B_1\}$, *then* $\{A_0 \vee A_1\}Prog\{B_0 \vee B_1\}$;

$\{I\}Prog\{I\}, \{0\}Prog\{B\}$, *where* B *is any assertion, and* 0 *represents an empty set of program states. For example, if* s_{e_1} *and* s_{e_2} *anti-commute,* $s_{e_1} \wedge s_{e_2} = 0$.

Proof. We first prove the conjunction rule. Since $A_0 \wedge A_1 \Rightarrow A_0$, $A_0 \wedge A_1 \Rightarrow A_1$, then by the consequence rule, we have $\{A_0 \wedge A_1\}\text{Prog}\{B_0\}$ and $\{A_0 \wedge A_1\}\text{Prog}\{B_1\}$, i.e., $\{A_0 \wedge A_1\}\text{Prog}\{B_0 \wedge B_1\}$. For the disjunction rule, notice that if $(\rho, \sigma) \models (A_0 \vee A_1)$, then either $(\rho, \sigma) \models A_0$ or $(\rho, \sigma) \models A_1$. Finally, $\{I\}\text{Prog}\{I\}$ always holds since any state (ρ, σ) satisfies I. $\{0\}\text{Prog}\{B\}$ is true because $(\rho, \sigma) \models 0 \Rightarrow [\![P]\!](\rho, \sigma) \models B$. □

Finally, we have the soundness theorem of Hoare rules in Figure 7.3.

Theorem 7.3.10 (Soundness)**.** *The proof system in Figure 7.3 is sound for partial correctness assertions.*

Proof. (1) Skip. Note than the skip rule does not change the program state.

(2) Initialization. By the definition of the substitution rule, $(\rho, \sigma) \models A[|0\rangle / \rho]$ is equivalent to $(\rho_0^q, \sigma) \models A$, then the state after initialization $(\rho', \sigma) = (\rho_0^q, \sigma)$ also satisfies A.

(3) Unitary. Note that $(UAU^\dagger)(U\rho U^\dagger) = UA\rho U^\dagger$, so $(UAU^\dagger)(U\rho U^\dagger) = (U\rho U^\dagger) \Leftrightarrow A\rho = \rho$.

(4) Assignment. The rule is obviously correct, since it does not change the program state.

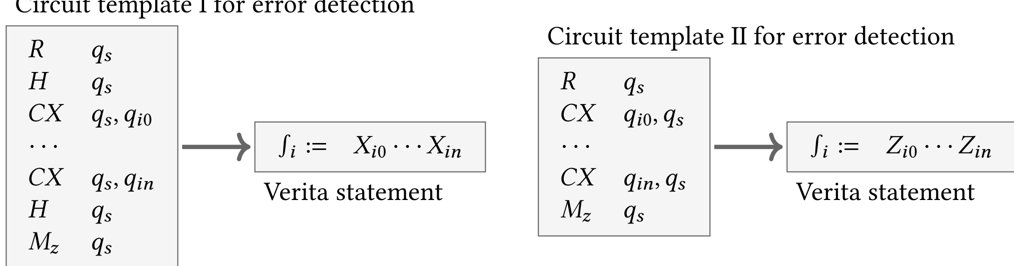
(5) Sequencing. Assume $(\rho,\sigma) \models A$, then $[\![P_0]\!](\rho,\sigma) \models C$ by the hypothesis $\{A\}P_0\{C\}$. On the other hand $[\![P_0;P_1]\!](\rho,\sigma) = [\![P_1]\!]([\![P_0]\!](\rho,\sigma)) \models B$ by the hypothesis $\{C\}P_1\{B\}$.

(6) Condition. First, $\sum A_i M_i$ is a legal stabilizer expression because $M_1 = \frac{I+\int}{2}$ and $M_0 = \frac{I-\int}{2}$ are legal stabilizer expressions. Assume $(\rho,\sigma) \models A$, then $\sigma(\int)$ commutes with A, so is M_1 and M_0. Thus, $AM_1\rho M_1^\dagger = M_1 A\rho M_1^\dagger = M_1\rho M_1^\dagger$. Likewise, we have $AM_0\rho M_0^\dagger = M_0\rho M_0^\dagger$. Let $A = \sum_i A_i M_i$, then $AM_1\rho M_1^\dagger = A_1 M_1(M_1\rho M_1^\dagger) + A_0 M_0(M_1\rho M_1^\dagger) = A_1(M_1\rho M_1^\dagger)$ since $M_1 M_1 = M_1$, $M_1 M_0 = 0$. Thus, we have $A_1 M_1 \rho M_1^\dagger = M_1\rho M_1^\dagger$. Since $\int$ commutes with both A_1 and A_0, we have $(M_1\rho M_1^\dagger, \sigma) \models A_1$ and $(M_0\rho M_0^\dagger, \sigma[-\int/\int]) \models A_0$. Also, $(M_1\rho M_1^\dagger, \sigma) \models \int$ and $(M_0\rho M_0^\dagger, \sigma[-\int/\int]) \models -\int$. Thus, if $(\rho,\sigma) \models \sum_i A_i M_i$, we have $(M_1\rho M_1^\dagger, \sigma) \models A_1 \wedge \int$ and $(M_0\rho M_0^\dagger, \sigma) \models A_0 \wedge -\int$. Since $\{A_1 \wedge \int\}P_1\{B\}$ and $\{A_0 \wedge -\int\}P_0\{B\}$, by the semantics of the condition statement, we have:
$\{\sum A_i M_i\}\,\mathbf{if}\, M[\int, \bar{q}]\,\mathbf{then}\, P_0\,\mathbf{else}\, P_1\,\mathbf{end}\{B\}$.

(7) Consequence. Assume $(\rho,\sigma) \models A$, then $(\rho,\sigma) \models A'$ by $\{A \Rightarrow A'\}$. Since $\{A'\}\text{Prog}\{B'\}$, we have $[\![P]\!](\rho,\sigma) \models B'$. Then $[\![P]\!](\rho,\sigma) \models B$ by $B' \Rightarrow B$. Thus, $\{A\}\text{Prog}\{B\}$. □

7.4 Weakest Precondition Computation

Based on the proposed language for QEC programs and assertions, we implement an automated inference tool for verifying QEC programs, which adopt the weakest precondition (WP) computation technique to verify the QEC circuit which describes a logical operation. Generally, WP computation (WPC) is of the judgement: $\Phi_{post} \vdash \textit{statement} : \Phi_{pre}$, where Φ_{post} and Φ_{pre} are assertions, named post- and pre-conditions, respectively. Φ_{pre} represents the constraint on the input logical state, so that, after the statement, the



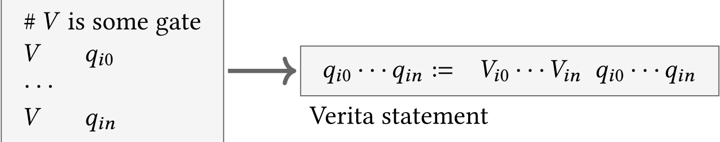


Figure 7.4: Templates and action rules for translating circuits into the VERITA language.

resulting logical state is in the quantum state subspace represented by Φ_{post} Here, we first give a sketch of the WPC for QECC correctness:

- **Step 1:** Converting the QEC circuit to a VERITA program by matching templates shown in Figure 7.4;
- **Step 2:** Converting the state-based precondition and post-condition to be stabilizer-based through program analysis and state computation;
- **Step 3:** Genenerating the WP of the QEC program for the post-condition of the VERITA version, using the inference rules shown in Figure 7.5.
- **Step 4:** Check if the precondition implies the generated WP.

Now, we will dive into the details of these four steps.

Step 1: program translation. Figure 7.4 shows the predefined circuit templates and the conversion rule. We can directly use these rules to translate the QEC circuit from the circuit language into our VERITA language. Figure 7.2(a) provides an example translation of the circuit in Figure 7.1(a).

Step 2: Pre/post-condition translation. The pre/post-condition of WPC for general quantum circuits are physical state-based. To speedup the verification on QEC circuits with VERITA, we need to translate pre/post-conditions into the sets of (generalized) stabilizer operators. The translation consists of two parts.

The first part is to extract stabilizer variables from QEC programs from Step 1. For each stabilizer variable $\int$, we will track its last assignment statement in the QEC program and use the stabilizer operator on the right hand size as part of the translated pre/post-conditions.

The second part involves computation over the predefined template. We first extract data qubit set $dqVar$ from the QEC program. This can be easily done by examing the stabilizer operators extracted by the first part of Step 2. Then, we will check if the logical state in the pre/post-condition is in the $+1$ eigenspace in one of following stabilizer operator templates:

$$\begin{gathered}\otimes_{i\in dqVar}X_i, \quad \otimes_{i\in dqVar}Z_i, \quad \otimes_{i\in dqVar}Y_i,\\ \frac{\otimes_{i\in dqVar}Y_i+\otimes_{i\in dqVar}X_i}{\sqrt{2}}, \frac{\otimes_{i\in dqVar}Z_i+\otimes_{i\in dqVar}X_i}{\sqrt{2}}, \frac{\otimes_{i\in dqVar}Y_i+\otimes_{i\in dqVar}Z_i}{\sqrt{2}}\\ (\otimes_{i\in dqVar}X_i)\otimes(\otimes_{i\in dqVar'}X_i), (\otimes_{i\in dqVar}Z_i)\otimes(\otimes_{i\in dqVar'}Z_i), (\otimes_{i\in dqVar}Y_i)\otimes(\otimes_{i\in dqVar'}Y_i),\end{gathered} \tag{7.11}$$

The bottom templates of Equ 7.11 are for the case where two logical qubits are involved in the QEC program.

If the logical state to be translated is already be in the $+1$ eigenspace of one stabilizer operator template, then we will end Step 2 by including that stabilizer operator in the translated pre/post-conditions. Otherwise, Lemma 7.4.1 provides us one way to find the remaining stabilizer operators for pre/post-condition translation.

Lemma 7.4.1. *If a logical state $|\psi_L\rangle = \alpha|0_L\rangle + \beta|1_L\rangle$, then it is the +1 eigenstate of $\frac{\alpha^2-\beta^2}{\alpha^2+\beta^2} \otimes_{i\in dqVar} Z_i + \frac{2\alpha\beta}{\alpha^2+\beta^2} \otimes_{i\in dqVar} X_i$.*

Lemma 7.4.1 can be simply proved by computing $(\frac{\alpha^2-\beta^2}{\alpha^2+\beta^2} \otimes_{i\in dqVar} Z_i + \frac{2\alpha\beta}{\alpha^2+\beta^2} \otimes_{i\in dqVar}$

$$\frac{\Phi_1 \equiv \Phi}{\Phi \vdash \textbf{skip} : \Phi_1} \tag{7.12}$$

$$\frac{\Phi_1 \equiv \Phi[\otimes_i |0\rangle / \bar{q}]}{\Phi \vdash \bar{q} := \otimes_i |0\rangle : \Phi_1} \tag{7.13}$$

$$\frac{\Phi_1 \equiv U \Phi U^\dagger}{\Phi \vdash \bar{q} := U\bar{q} : \Phi_1} \tag{7.14}$$

$$\frac{\Phi_1 \equiv \Phi}{\Phi \vdash \int := s : \Phi_1} \tag{7.15}$$

$$\frac{\Phi \vdash Prog\colon \Phi_2 \quad \Phi_1 \vdash Prog\colon \Phi_3}{\Phi \wedge \Phi_1 \vdash Prog : \Phi_2 \wedge \Phi_3} \tag{7.16}$$

$$\frac{\Phi \vdash Prog_2\colon \Phi_2 \quad \Phi_2 \vdash Prog_1\colon \Phi_1}{\Phi \vdash Prog_1; Prog_2 : \Phi_1} \tag{7.17}$$

$$\frac{\Phi \vdash Prog_2\colon \Phi_1 \quad \Phi \vdash Prog_1\colon \Phi_1}{\Phi \vdash \textbf{if}\, M[\int, \bar{q}]\, \textbf{then}\, \mathrm{Prog}_1\, \textbf{else}\, \mathrm{Prog}_0\, \textbf{end} : \Phi_1} \tag{7.18}$$

$$\frac{\Phi \vdash Prog_2\colon \Phi_2 \quad \Phi \vdash Prog_1\colon -\Phi_2}{\Phi \vdash \textbf{if}\, M[\int, \bar{q}]\, \textbf{then}\, \mathrm{Prog}_1\, \textbf{else}\, \mathrm{Prog}_0\, \textbf{end} : \Phi_2 * \int} \tag{7.19}$$

$$\frac{\Phi_1 \equiv \Phi}{\Phi \wedge (\wedge_i^n \int_i) \vdash \textbf{correct}(\int_1, \ldots, \int_n) : \Phi_1} \tag{7.20}$$

Figure 7.5: Weakest precondition computation rules.

$X_i)|\psi_L\rangle$. Then we can end Step 2 by including that the stabilizer operator by Lemma 7.4.1 in the translated pre/post-conditions. Notice that, the computation of this part is not affected by the exponential number of physical states in the logical state, as it only requires the superposition information of $|0_L\rangle$ and $|1_L\rangle$, which is already accessible from pre/post-conditions before translation.

Step 3: WP generation. Figure 7.5 shows rules for generating WP. These rules are basically the translation of the Hoare logic shown in Figure 7.3. The proof for these rules can be translated from our proof for the Hoare logic in Figure 7.3, and we omit them here for simplicity.

Rules 7.12—7.17 are already self-explained. Rule 7.20 is the blackbox-like inference over quantum error decoding and correction, which corresponds to the case where no error

observed. We elaborate on Rule 7.18 and 7.19 that are special cases of the condition rule in Figure 7.3. The proof is straightforward and omit it here for simplicity. Rule 7.18 is specifically designed for the case where the condition statement does not change the stabilizer operator in post-condition. This case is met when the post-condition is a error detection-related stabilizer operator. Rule 7.19 is designed for the case where the stabilizer operator for measurement is a 'substring' of the stabilizer operator in the pre-condition. For example, considering the pre-condition Z_1Z_2 and stabilizer measurement Z_1, the measurement by Z_1 will indeed cast the pre-condition to $Z_1 \wedge Z_2$. Rule 7.19 can be applied to recover Z_1Z_2 through WPC. Rule 7.19 is critical for nontranverse logical operations whose implementations heavily rely on the condition statement.

Note that, we will convert the intermediate precondition induced by the WPC into the standard form indicated in Equ 7.11 and Lemma 7.4.1. Such a conversion is cheap since the backward inference will not substantially increase the amount of stabilizer terms in the intermediate precondition, considering the stabilizer-based nature of the proposed *Veri-Lang*. This conversion not only simplifies the precondition implication process discussed below, but also decreases the overhead of the WPC since the number of terms in the pre/post-conditions are reduced.

Step 4: Precondition Implication. We can determine if a stabilizer operator s_1 implies s_2 or not by string comparison. If the pauli operator of s_1 on q is the same as the pauli operator of s_2 on q for each $q \in dqVar$, or s_2 is all of identity operators, then s_1 implies s_2. For example, Z_1Z_2 implies I_1I_2. The proof for this checking is obvious and we omit it here. This implication checking is sufficient for our verification of QECC logical operations. During the implication checking, we will further ignore error detection-related stabilizer operators (by applying Lemma 7.3.4) to reduce the computation overhead.

7.5 Evaluation

This section presents a series of experiments designed to answer the following research questions:

- Q1: Can VERITA automatically verify general QEC codes? How does it scale with their size?
- Q2: How does VERITA's performance compare against the state-of-the-art quantum verifier?
- Q3: How important is each of the components in VERITA?

7.5.1 Experiment Setup

Platform. All of our reasoning toolkits (including the weakest precondition computation engine) are implemented in Python 3.10. All experiments are run on a Ubuntu 18.04 server with a 6-core Intel E5-2603v4 CPU and 16GB RAM.

Baselines. We use the quantum **while**-language [120], in short qWhile, as our primary baseline: qWhile is the state-of-the-art verification framework for general quantum programs. Different from our approach, qWhile is based on the lowest-level circuit language and performs its reasoning directly over physical quantum states. Note that, for both verifiers, we treat the conditioned error correction operation as a blackbox function, over which the reasoning is defined by Rule 7.20 in Figure 7.5. Since our work is focused on demonstrating the correctness of logical operations, using this abstraction can rescue us from diving into the details of the error correction operation.

Benchmarks. We selected four popular QECCs for our benchmark suite, each featuring distinct implementations requirements for the common universal logic gate basis, known as the Clifford+T gate basis [1]. This logical gate set encompasses logical X, Z,

H, S, T, and CX gates, along with logical initialization. This diversity demonstrates our tool's broad applicability. Table 7.1 shows more detailed statistics (e.g., qubit count) about our benchmarks.

- *Steane code.* This QECC involves 7 data qubits and is a well-known example of the CSS code family [1]. Most logical operations of Steane code are based on qubit-wise physical quantum gates, like the logical H from Figure 7.1. The logical T gate instead requires the distilled magic state and the use of the branching statement [1].

- *Shor's code* [1]. This QECC involves 9 data qubits and heavily depends on distilled magic states and branching statements for logical H, logical S and logical T.

- *Reed Muller code* [127]. This QECC involves 15 data qubits, and uses the distilled magic state and the branching statement for logical H, with other logical operations based on qubit-wise gates.

- *The surface code family* [20]. Surface code is the one of the most popular QECC in the quantum community. Various versions of the surface code, distinguished by the code distance d, offer differing levels of protection: a larger d corresponds to more data qubits ($= d^2$) and thus enhanced protective power. Most logical operations of the surface code are based on qubit-wise gates, except for the logical T, which relies on the distilled magic state and the branching statement.

QECC correctness in weakest precondition computation (WPC). Table 7.2 summarizes pre/post-conditions to verify each logical operation of the Clifford+T gate basis, hence the QEC correctness. Due the linearity of quantum programs, we only need consider the pre/post-conditions based on the computational logical states, e.g., $|0_L\rangle$ and $|+_L\rangle$.

Table 7.1: Statistics of benchmark programs.

	Logical Initialization		Logical X/Z		Logical H		Logical S		Logical T		Logical CX	
	# var	# gate	# var	# gate	# var	# gate	# var	# gate	# var	# gate	# var	# gate
Steane code	13	60	13	49	13	49	13	56	26	137	26	91
Shor's code	17	70	17	57	34	154	34	145	34	154	34	105
Reed-Muller code	29	164	29	145	58	354	29	145	29	145	58	275
Surface code $d = 3$	17	66	17	59	17	59	34	127	34	142	34	115
Surface code $d = 5$	49	178	49	171	49	171	98	351	98	366	98	339
Surface code $d = 7$	97	346	97	339	97	339	194	687	194	702	194	675
Surface code $d = 11$	241	850	241	843	241	843	482	1695	482	1710	482	1683
Surface code $d = 21$	881	3090	881	3083	881	3083	1762	6175	1762	6190	1762	6163
Surface code $d = 31$	1921	6730	1921	6723	1921	6723	3842	13455	3842	13470	3842	13443
Surface code $d = 41$	3361	11770	3361	11763	3361	11763	6722	23535	6722	23550	6722	23523
Surface code $d = 51$	5201	18210	5201	18203	5201	18203	10402	36415	10402	36430	10402	36403
Surface code $d = 61$	7441	26050	7441	26043	7441	26043	14882	52095	14882	52110	14882	52083
Surface code $d = 71$	10081	35290	10081	35283	10081	35283	20162	70575	20162	70590	20162	70563

7.5.2 Q1: Can Verita Verify Large QEC Codes?

We evaluate Verita on all our benchmarks, especially focusing on its scalability for complex QEC codes (e.g., high-distance surface code). For example, large quantum programs (such as Shor's algorithm [126]) require complex surface code with distance at least 61, in order to protect the underlying computation and demonstrate the quantum advantage [20].

Results. Table 7.3 shows our main results. As we can see, Verita has significantly greater scalability compared to qWhile. In contrast, Verita can successfully verify the distance-71 surface code within 3 hours. The scalability of Verita stems from two factors. The first factor is the concise QEC-tailored language. By using abstracting groups of quantum gates into high-level stabilizer operators, Verita significantly reduces the overhead of reasoning, e.g., over QEC error detection subroutines in each QEC programs.

The second factor is the stabilizer-centric proof system, which greatly reduces the number of assertion terms during the verification process. Taking the surface code as an example, each logical state of the distance-d surface code is a highly-entangled physical quantum states over data qubits, containing at least $2^{num_x_stabilizers} = \sqrt{2}^{d^2-1}$ physical quantum states. With Verita, at most $d^2 + 3$ stabilizer operators are involved in

Table 7.2: QECC correctness in weakest precondition computation (WPC). For simplicity, the error detection-related stabilizer generator of QECC for Verita goals and the ancillary state for qWhile goals are omitted. I_L, X_L, Z_L, Y_L refer to stabilizer operators consisting of I, X, Z, Y on all data qubits, respectively.

	Logical initialization		Logical X		Logical Z		Logical H		Logical S	
	Pre	Post	Pre	Post	Pre	Post	Pre	Post	Pre	Post
Verita goals	I_L	Z_L	Z_L X_L	$-Z_L$ X_L	Z_L X_L	Z_L $-X_L$	Z_L X_L	X_L Z_L	Z_L X_L	Z_L Y_L
qWhile goals	0	$\|0_L\rangle$	$\|0_L\rangle$ $\|+_L\rangle$	$\|1_L\rangle$ $\|+_L\rangle$	$\|0_L\rangle$ $\|+_L\rangle$	$\|0_L\rangle$ $\|-_L\rangle$	$\|0_L\rangle$ $\|+_L\rangle$	$\|+_L\rangle$ $\|0_L\rangle$	$\|0_L\rangle$ $\|+_L\rangle$	$\|0_L\rangle$ $\|0_L\rangle + i\|1_L\rangle$

	Logical CX		Logical T	
	Pre	Post	Pre	Post
Verita goals	$I_L \otimes Z_L$ $Z_L \otimes I_L$ $I_L \otimes X_L$ $X_L \otimes I_L$	$Z_L \otimes Z_L$ $Z_L \otimes I_L$ $I_L \otimes X_L$ $X_L \otimes X_L$	Z_L X_L	Z_L $\frac{X_L+Y_L}{\sqrt{2}}$
qWhile goals	$(a\|0_L\rangle + b\|1_L\rangle)\|0_L\rangle$ $\|0_L(a\|0_L\rangle + b\|1_L\rangle)\rangle$ $(a\|0_L\rangle + b\|1_L\rangle)\|+_L\rangle$ $\|+_L\rangle(a\|0_L\rangle + b\|1_L\rangle)$	$a\|0_L\rangle\|0_L\rangle + b\|1_L\rangle\|1_L\rangle$ $\|0_L(a\|0_L\rangle + b\|1_L\rangle)\rangle$ $(a\|0_L\rangle + b\|1_L\rangle)\|+_L\rangle$ $a(\|0_L\rangle\|0_L\rangle + \|1_L\rangle\|1_L\rangle)+$ $b(\|0_L\rangle\|1_L\rangle + \|1_L\rangle\|0_L\rangle)$	$\|0_L\rangle$ $\|+_L\rangle$	$\|0_L\rangle$ $\|0_L\rangle + e^{i\frac{\pi}{4}}\|1_L\rangle$

		Logical Initialization	Logical X/Z	Logical H	Logical S	Logical T	Logical CX
Steane code	qWhile	9.4×10^{-4}	7.8×10^{-4}	3.7×10^{-3}	1.2×10^{-3}	9.3×10^{-3}	2.0×10^{-3}
	Verita	4.5×10^{-4}	4.3×10^{-4}	3.3×10^{-4}	4.7×10^{-4}	3.2×10^{-3}	1.2×10^{-3}
	Speedup by Verita	**2.1** $\times 10^{+0}$	**1.8** $\times 10^{+0}$	**1.1** $\times 10^{+1}$	**2.6** $\times 10^{+0}$	**2.9** $\times 10^{+0}$	**1.7** $\times 10^{+0}$
Shor's code	qWhile	6.3×10^{-4}	4.0×10^{-4}	4.1×10^{-2}	2.0×10^{-2}	2.0×10^{-2}	8.6×10^{-4}
	Verita	4.6×10^{-4}	2.6×10^{-4}	8.5×10^{-3}	8.1×10^{-3}	8.0×10^{-3}	8.4×10^{-4}
	Speedup by Verita	**1.4** $\times 10^{+0}$	**1.5** $\times 10^{+0}$	**4.8** $\times 10^{+0}$	**2.5** $\times 10^{+0}$	**2.5** $\times 10^{+0}$	**1.0** $\times 10^{+0}$
Reed-Muller code	qWhile	4.3×10^{-3}	4.0×10^{-3}	3.4×10^{-2}	5.1×10^{-3}	7.1×10^{-3}	1.1×10^{-2}
	Verita	1.0×10^{-3}	1.0×10^{-3}	2.5×10^{-2}	1.5×10^{-3}	1.8×10^{-3}	2.8×10^{-3}
	Speedup by Verita	**4.1** $\times 10^{+0}$	**4.0** $\times 10^{+0}$	**1.3** $\times 10^{+2}$	**3.3** $\times 10^{+0}$	**4.0** $\times 10^{+0}$	**3.7** $\times 10^{+0}$
Surface code $d = 3$	qWhile	9.5×10^{-3}	7.9×10^{-3}	1.3×10^{-1}	2.3×10^{-2}	2.3×10^{-1}	2.9×10^{-2}
	Verita	8.8×10^{-4}	4.1×10^{-4}	4.9×10^{-4}	3.8×10^{-3}	6.9×10^{-3}	1.7×10^{-3}
	Speedup by Verita	**1.1** $\times 10^{+1}$	**1.9** $\times 10^{+1}$	**2.6** $\times 10^{+2}$	**6.1** $\times 10^{+0}$	**3.4** $\times 10^{+1}$	**1.7** $\times 10^{+1}$
Surface code $d = 5$	qWhile	$2.8 \times 10^{+3}$	$2.8 \times 10^{+3}$	>3h	$6.3 \times 10^{+3}$	>3h	$8.1 \times 10^{+3}$
	Verita	1.6×10^{-3}	1.3×10^{-3}	2.4×10^{-3}	1.8×10^{-2}	2.9×10^{-2}	7.1×10^{-3}
	Speedup by Verita	**1.8** $\times 10^{+6}$	**2.2** $\times 10^{+6}$	N/A	**3.5** $\times 10^{+5}$	N/A	**1.1** $\times 10^{+6}$
Surface code $d = 7$	qWhile	Out of Memory					
	Verita	2.2×10^{-3}	2.1×10^{-3}	5.9×10^{-3}	5.7×10^{-2}	9.2×10^{-2}	2.5×10^{-2}
	Speedup by Verita	N/A					

Table 7.3: Verification time (in seconds, unless otherwise specified) of Verita and qWhile on our benchmarks.

assertions. Though each stabilizer operator involves d^2 symbolic Pauli matrices over all data qubits, the storage complexity of Verita for asserting the surface code are at most

		Logical Initialization	Logical X/Z	Logical H	Logical S	Logical T	Logical CX
Surface code $d = 11$	qWhile			Out of Memory			
	VERITA	6.9×10^{-3}	6.8×10^{-3}	3.3×10^{-2}	3.9×10^{-1}	5.4×10^{-1}	2.0×10^{-1}
	Speedup by VERITA			N/A			
Surface code $d = 21$	qWhile			Out of Memory			
	VERITA	3.7×10^{-2}	3.7×10^{-2}	4.3×10^{-1}	$6.0 \times 10^{+0}$	$1.1 \times 10^{+1}$	$3.3 \times 10^{+0}$
	Speedup by VERITA			N/A			
Surface code $d = 31$	qWhile			Out of Memory			
	VERITA	1.6×10^{-1}	1.1×10^{-1}	$2.2 \times 10^{+0}$	$3.7 \times 10^{+1}$	$7.0 \times 10^{+1}$	$2.4 \times 10^{+1}$
	Speedup by VERITA			N/A			
Surface code $d = 41$	qWhile			Out of Memory			
	VERITA	2.9×10^{-1}	2.9×10^{-1}	$6.6 \times 10^{+0}$	$1.5 \times 10^{+2}$	$2.4 \times 10^{+2}$	$1.0 \times 10^{+2}$
	Speedup by VERITA			N/A			
Surface code $d = 51$	qWhile			Out of Memory			
	VERITA	5.1×10^{-1}	5.0×10^{-1}	$1.6 \times 10^{+1}$	$4.0 \times 10^{+2}$	$6.0 \times 10^{+2}$	$2.8 \times 10^{+2}$
	Speedup by VERITA			N/A			
Surface code $d = 61$	qWhile			Out of Memory			
	VERITA	9.2×10^{-1}	8.9×10^{-1}	$3.3 \times 10^{+1}$	$8.4 \times 10^{+2}$	$1.3 \times 10^{+3}$	$6.3 \times 10^{+2}$
	Speedup by VERITA			N/A			
Surface code $d = 71$	qWhile			Out of Memory			
	VERITA	$1.5 \times 10^{+0}$	$1.4 \times 10^{+0}$	$6.1 \times 10^{+1}$	$1.5 \times 10^{+3}$	$2.2 \times 10^{+3}$	$1.2 \times 10^{+3}$
	Speedup by VERITA			N/A			

Table 7.4: Continued table to Table 7.3. Verification time (in seconds, unless otherwise specified) of VERITA and qWhile on our benchmarks.

$O(d^4)$. The computational complexity of VERITA depends on the program and are at most $O(d^6)$ since each program contains at most $O(d^2)$ lines. This complexity reasoning, together with results shown in Table 7.3, demonstrates the significantly better scalability of our framework.

7.5.3 Q2: How Does VERITA Compare Against Baseline?

In this section, we compare the performance of VERITA to that of the baseline, qWhile [120].

As shown in Table 7.3, compared to the baseline, VERITA achieves significant verification time reduction for all QECCs considered. The verification time reduction by VERITA becomes larger as the program size (see # var and # gate in Table 7.1) increases. For instance, VERITA demonstrates a notably higher speedup on the distance-5 surface

code compared to the speedup observed for the distance-3 surface code and Reed-Muller code. The only exception to this trend is Shor's code. In the case of Shor's code, the stabilizer operators are of small weights, mostly 2, leading to minimal overhead when reasoning over error detection circuits with qWhile. Additionally, Shor's code features the simplest logical states since it contains only two X-type stabilizer operators. Overall, we expect VERITA to be more efficient for QEC codes with high-weight stabilizer generators (as those will induce more reasoning overhead over error detection circuits) and more X-type stabilizer operators (as those will induce more complex logical states).

Furthermore, VERITA has significantly better scalability than the baseline. qWhile is limited to verifying surface codes up to distance 7. Instead, VERITA can successfully verify the distance-71 surface code within a verification time shorter than what qWhile requires to verify the distance-5 surface code. The main reason for this scalability discrepancy stems from the representation of assertions. For qWhile where assertions are based on quantum states, exponential number of physical quantum states will be visited in the verification process. Though we have tried to symbolize the inference computation, the computational and storage complexity of qWhile for a distance-d surface code still scales exponentially with d^2.

7.5.4 Q3: Ablation Studies

We consider the following ablations in this experiment. Note that the language for assertion and the verification algorithm must be used together by design.

- "qlang + vassn" is a variant of VERITA with our new stabilizer-based program abstractions disabled. Instead, it uses the existing qWhile language [120] to represent QEC circuits. The term 'qlang' denotes the qWhile language and 'vassn' denotes the assertion and proof system of VERITA.

Distance	Verification Time (s)	Logical Initialization	Logical X/Z	Logical H	Logical S	Logical T	Logical CX
3	vlang+qassn	9.1×10^{-4}	4.9×10^{-4}	1.2×10^{-1}	9.7×10^{-3}	2.3×10^{-1}	2.1×10^{-3}
	Speedup by VERITA	**1.0** $\times10^{+0}$	**1.2** $\times10^{+0}$	**2.5** $\times10^{+2}$	**2.6** $\times10^{+0}$	**3.3** $\times10^{+1}$	**1.2** $\times10^{+0}$
	qlang+vassn	7.2×10^{-3}	6.5×10^{-3}	7.0×10^{-3}	3.4×10^{-2}	3.0×10^{-2}	4.3×10^{-2}
	Speedup by VERITA	**8.2** $\times10^{+0}$	**1.6** $\times10^{+1}$	**1.4** $\times10^{+1}$	**8.9** $\times10^{+0}$	**4.3** $\times10^{+0}$	**2.6** $\times10^{+1}$
5	vlang+qassn	$3.1\times10^{+1}$	$3.0\times10^{+1}$	>3h	$3.8\times10^{+3}$	>3h	$6.3\times10^{+2}$
	Speedup by VERITA	**2.0** $\times10^{+4}$	**2.4** $\times10^{+4}$	N/A	**2.1** $\times10^{+5}$	N/A	**8.9** $\times10^{+4}$
	qlang+vassn	5.1×10^{-2}	5.1×10^{-2}	6.1×10^{-2}	3.5×10^{-1}	3.2×10^{-1}	5.5×10^{-1}
	Speedup by VERITA	**3.3** $\times10^{+1}$	**4.0** $\times10^{+1}$	**2.6** $\times10^{+1}$	**1.9** $\times10^{+1}$	**1.1** $\times10^{+1}$	**7.8** $\times10^{+1}$
≥7	vlang+qassn	Out of Memory					
	Speedup by VERITA	N/A					
	qlang+vassn Speedup by VERITA	Scales up to $d = 51$ (verifies each program within 3h), see Figure 7.6					

Table 7.5: Verification time of 'vlang+qassn' and 'qlang+vassn' on the surface code family.

- "vlang + qassn" is an ablation which replaces our proof system with the one from qWhile [120]. As a result, this variant also uses the verification algorithm from the qWhile work, in order for the approach as a whole to function. The term 'vlang' denotes the language of VERITA and 'qassn' denotes the assertion and proof system of qWhile.

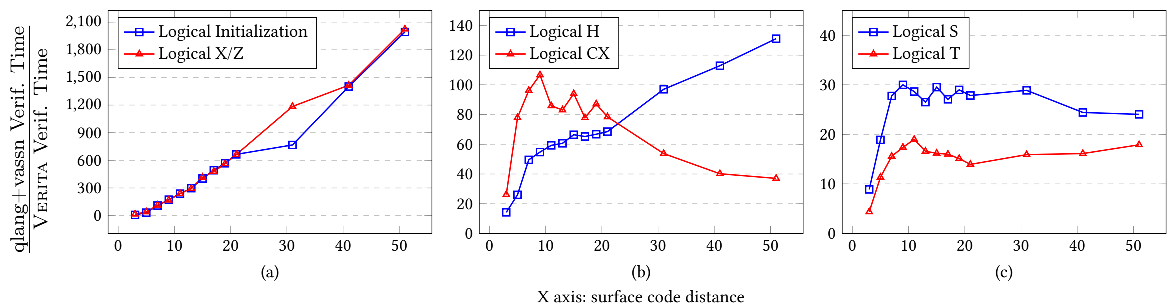


Figure 7.6: Comparing VERITA to 'qlang+vassn'.

Impact of stabilizer-based program states and proof system (vassn): As shown in Table 7.5 and Figure 7.6, vassn scales significantly better than qassn, combined with either vlang or qlang. Similar to results in Table 7.3, for the distance-5 surface code, qassn-based verification ('vlang+qassn') consumes thousands of times more time than vassn-based verification (VERITA and 'qlang+vassn'). Further, like qWhile, 'vlang+qassn' cannot scale to the surface code of distance higher than 7. Instead,

‘qlang+vassn’ can scale up to distance-51 surface code, though sharing the same language system for QEC circuits with qWhile. Overall, vassn plays the key role in scaling up the verification of QECC correctness.

Impact of stabilizer-based language for QEC circuits (vlang): As shown in Figure 7.6, vlang further helps reduce the verification time: the verification time by ‘qlang+vassn’ is dozens times, even thousands times larger than the time by VERITA. This comparison demonstrates the importance of vlang in scaling up the verification. Compared to qlang, vlang reduces the overhead of reasoning through stabilizer measurement circuits, through incorporating stabilizer operators into the circuit language. Even combined with qassn, vlang still significantly reduces the verification time (see the verification time of ‘vlang+qassn’ in Table 7.5 and the verification time in Table 7.3).

Chapter 8

Related Work

8.1 Optimization of Distributed Quantum Computing

Most existing quantum compilers [46, 28, 47, 48, 49] focus on the compilation of programs within a single quantum computer. These works do not consider inter-node communication. Extending them to DQC cannot achieve high communication throughput in distributed quantum programs.

Unfortunately, existing compilers for DQC adopt similar methodologies to single-node quantum compilers. One compiler design proposed by Ferrari et al. [45] exploits Cat-Comm to implement each remote CX gate independently, treating the remote CX like the local CX. Another compiler design by Ferrari et al. [45] and the compiler by Baker et al. [42] use remote SWAP gates to transform remote operations into local operations, resembling SWAP-based routing (e.g., SABRE [46]) for single-node quantum programs. Diadamo et al. [6] consider specific optimizations of inter-node controlled-unitary blocks. However, their work requires specialized circuit representations and cannot optimize general quantum programs. All these works do not consider the burst and collective communication proposed in the dissertation and thus cannot achieve high communication

throughput.

Another line of work executes distributed quantum programs without using inter-node quantum communication protocols [131, 132]. These works run large quantum circuits in a divide-and-conquer way. To overcome the expressibility loss due to no inter-node communication, these works rely heavily on classical post-processing techniques and cannot be extended to large-scale quantum programs. Häner et al. [68] present a DQC programming framework that optimizes collective operations as library functions. However, their work does not consider the program context and the underlying hardware topology, which are essential for communication optimizations proposed in the dissertation. Further, their definition of collective communication in DQC is based on the classical counterpart defined in MPI [133] and is only a subset of the proposed definition in the dissertation.

There are also works trying to reduce the communication overhead of distributed quantum programs by exploring various circuit partition/qubit mapping techniques [5, 67, 66, 64, 65]. These works are orthogonal and can be merged into the proposed framework in the dissertation.

8.2 QEC Code Synthesis

Compiling a general quantum circuit onto a quantum device with limited qubit connectivity has been widely studied [90, 91, 92, 93, 94, 95, 96, 97]. However, these general quantum compilers are not suitable for compiling and synthesizing QEC codes to quantum hardware. Firstly, they don't distinguish data qubits from other qubits. They may move data qubits frequently, invalidating logical operations designed for a fixed data qubit layout [20]. Secondly, the SWAP gates they use to overcome the connectivity issue of the SC device make the compiled measurement circuits too error-prone for practical

error correction. Finally, they do not account for the constraints imposed by QEC codes, e.g., the order of CNOT gates between syndrome qubits and data qubits [20].

As for the compilation over QEC codes, most circuit compilation works on QEC codes are at the higher logical circuit level. Javadi et al. [86] and Hua et al. [87] studied the routing congestion in circuits over the surface code. Ding et al. [84] and Paler et al. [85] studied the compilation of magic state distillation circuits with existing surface code logical operations for realizing a universal quantum gate set. Lao et al. [134] proposed a mapping process to execute lattice surgery-based quantum circuits on surface code architectures. These works assume that the ideal surface code architecture is already available and do not consider the problem of synthesizing surface codes on hardware. In contrast, this dissertation mainly focuses on building the underlying QEC-based quantum computing platform upon diverse quantum hardware. Also, the proposed compiler optimizations for QEC-based programs are designed for the code switching operation which is not considered in existing works.

As for the architecture design, most efforts on QEC code synthesis are still on looking for an hardware architecture that is suitable for the target code. Reichardt [135] proposed three possible planar qubit layouts for synthesizing the seven-qubit color code. Chamberland et al. [34] proposed a trivalent architecture where it is straightforward to allocate data qubits of triangular color codes. Chamberland et al. [30] introduced heavy architectures which reduce qubit frequency collisions while still providing support for the surface code synthesis. Instead, the proposed framework in the dissertation can automatically synthesize QEC codes onto various quantum hardware and avoid manually redesigning code protocols for the ever-changing quantum device architecture. Further, those works cannot be simply extended to support the code switching operation which involves dynamic conversion between two QEC codes, let alone providing compiler optimization and architecture-compiler co-design for code switching.

Another line of research targets compiling stabilizer measurement circuits to existing SC architectures. Lao and Almudever [31] proposed the flag-bridge circuit which can measure the stabilizer of the Steane code on the IBM-20 device. However, their work relies on manually appointed data qubits and bridge qubits, and focuses on the IBM-20 device. Methods in this category are orthogonal and can be easily merged into the proposed framework in the dissertation.

8.3 Verification of Quantum Programs

The dissertation is also related to works on quantum programming language and verification, especially the quantum Hoare logic. The comparison focuses on the constructs in quantum program languages and proof systems.

From the view of operator and state representation, existing quantum programming language works can be primarily divided into two categories. The first category of works [120, 121, 122, 123, 136, 137, 138, 139, 140] uses the general Hermitian matrix and density matrix to represent quantum operators and states. While being general and compatible with quantum measurements and branching statements, this representation may cause exponential computational/storage overhead to track the evolution of quantum states under quantum operators if the number of involved qubits is large, depending on the symbolization of the Hermitian and density matrix adopted. This category of languages is not suitable to express QEC programs where the QEC operator will involve many qubits.

The second category of works [141, 142] adopts the path-sum representation for quantum operators and the statevector description for quantum states. The path-sum representation can greatly reduce the computation cost of reasoning over the quantum program. Unfortunately, this representation is only applicable to quantum circuits. QEC

codes, in contrast, also contain classical-quantum procedures, e.g., the logical operation based on magic states. While a classical-quantum procedure can be converted into a pure quantum circuit by the deferred measurement principle [1], verifications of the classical-quantum procedure used in QEC and the associated quantum circuit are not equivalent as those two programs behave differently when quantum error presents: the classical part of a classical-quantum procedure is still exact under quantum errors while all parts of a quantum circuit would be potentially affected by quantum errors. Moreover, the path-sum representation still cannot overcome the exponential storage overhead induced by highly-entangled quantum states. [143] adopts a hybrid representation for quantum circuits by using both the matrix and path-sum representation. Once again, this hybrid representation is not a fit for QEC programs for the reasons just discussed. As a comparison, the framework proposed in the dissertation adopts the stabilizer representation for quantum operators, allowing a compact description of QEC-based programs and efficient tracking of quantum states.

From the perspective of predicate logic, existing works can be primarily divided into two categories. The first category of verification works [119, 120, 121, 122, 123, 124] exploits expectation-based predicates. The extent a quantum state ρ satisfies the quantum predicate O is measured by $Tr(O\rho)$. Based on such predicates, Ying et al. [120] proposed a sound and relatively complete Hoare logic for their quantum **while**-language which is further extended to handle parallel quantum programs [121], quantum programs with classical variables [122] and distributed quantum programs [123]. However, this type of quantum Hoare logic does not fit QEC programs where it is required to know exactly whether a program state is legal instead of a satisfactory probability, for the purpose of quantum error correction. Besides, this type of proof system may cause significant computation overhead when O and ρ are large matrices and involve exponential number of terms.

The second category of verification works [125, 136, 137, 138, 139] focuses on the bi-valued satisfactory relation between a quantum program state and a quantum predicate: satisfied or not satisfied, just as in classical predicate logic. One emerging research line lying in this category is to use projection-based predicates [125, 139]. With projection-based predicates, Unruh [139] proposed a quantum Hoare logic with ghost variables to enhance the expressive power of the quantum assertion language. The framework proposed in the dissertation also uses projection-based predicates but these predicates are described in the form of stabilizer expressions for an efficient reasoning with stabilizer variables. Though seeming similar, the ghost variable proposed by Unruh [139] is conceptually different from the stabilizer variable in this work. The ghost variable is only used in predicates and does not owe any QEC-related semantics. Instead, the stabilizer variable has a clear semantics: it represents a stabilizer measurement circuit.

Rand et al. [144, 145, 146] develop an elegant type-checking system for general quantum programs based on the stabilizer formalism. Although this work is also motivated by high-level insights in utilizing stabilizer formalism [147], the dissertation differs significantly in the overall optimization goal and the entire design framework. In addition, their works only consider quantum circuits and cannot deal with branching statements which are indispensable for QEC programs. In contrast, the proposed framework can handle branching statements by incorporating stabilizer variables in the design of quantum predicate logic. Last but not least, the proposed framework also develops a compact language for QEC programs while their work follows the vanilla quantum circuit language [1]. Yu et al. [148] develop an abstract interpretation technique for quantum computing. However, their technique has strict constraint on assertions and cannot be directly applied to the QEC verification. Finally, there are also other work which develop quantum predicate logic in debugging and testing (run-time analysis) [124, 125]. Those works are out of the scope of the dissertation which is for the verification (static analysis) of QEC codes.

Chapter 9

Conclusion and Discussion

The dissertation proposes a large-scale, fault-tolerant and computationally-efficient quantum computing framework by orchestrating architecture, compiler and programming language efforts. The proposed computing framework can speedup the way towards demonstrating practical quantum advantage. The proposed framework is both highly scalable and automated, and is the first systematic attempt that formalizes and optimizes design problems for large-scale quantum computing. The proposed framework can be used to automatically configure an efficient and reliable execution of a given quantum application on a specific quantum device architecture. A typical workflow can be: a) inspecting the quantum application and quantum hardware to determine the optimal QEC-based architecture design that guarantees the required noise-resilience; b) optimizing the communication efficiency between inter-node error-corrected qubits with the proposed system design; c) optimizing intra-node QEC-based quantum operations; d) fine-tuning the QEC-based architecture and the communication efficiency by the proposed co-designs.

Although The proposed framework significantly surpasses existing works in improving the communication efficiency and noise resilience of large-scale quantum applications, there is still much space left for potential improvements. Specifically, it is promising to

explore the following directions:

Improving QECC's on-device encoding. For better scalability, the proposed framework adopts a greedy strategy when constructing error-corrected logical qubits. However, when the device architecture becomes complicated, the proposed framework may not generate the optimal data qubit layout for logical qubits. The first way to solve the problem is to incorporate more device information like topological symmetries into the logical qubit design; the second way is to use advanced optimization algorithms like simulated annealing or neural network to discover better on-device encoding of logical qubits.

Adapting communication optimization to higher-level program abstraction. This proposed framework works with the low-level circuit language to maintain compatibility with existing compiling flows. However, if higher-level program information is provided, more aggressive communication optimization could be enabled. It is promising to extend existing quantum programming languages to provide burst/collective communication primitives which could expose extra burst/collective communication patterns that are difficult to uncover at a low level.

Co-designing DQC architecture and software. For a wide applicability of the proposed framework, the dissertation assumes a general DQC architecture/system. It is possible to use the proposed framework to guide the DQC network design: modifying the network topology and inspecting the communication cost of distributed programs compiled by the proposed framework on the modified topology.

Co-designing communication optimizations with QECC synthesis. To ensure the generality of the proposed framework, the dissertation makes few assumptions for the underlying QEC structure of the DQC architecture. For specific QEC, it is possible to optimize the communication cost of primitive QEC operations, e.g., logical operations (over logical qubits) and stabilizer measurements.

Bibliography

[1] M. A. Nielsen and I. Chuang, *Quantum computation and quantum information*, 2002.

[2] 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.

[3] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, *Quantum machine learning*, *Nature* **549** (2017), no. 7671 195–202.

[4] J. Preskill, *Quantum computing in the nisq era and beyond*, *Quantum* **2** (2018) 79.

[5] P. Andr'es-Mart'inez and C. Heunen, *Automated distribution of quantum circuits via hypergraph partitioning*, *Physical Review A* (2019).

[6] S. Diadamo, M. Ghibaudi, and J. R. Cruise, *Distributed quantum computing and network control for accelerated vqe*, *IEEE Transactions on Quantum Engineering* **2** (2021) 1–21.

[7] S. L. N. Hermans, M. Pompili, H. K. C. Beukers, S. Baier, J. Borregaard, and R. Hanson, *Qubit teleportation between non-neighbouring nodes in a quantum network*, *Nature* **605** (2022) 663–668.

[8] N. Laracuente, K. N. Smith, P. Imany, K. L. Silverman, and F. Chong, *Short-range microwave networks to scale superconducting quantum computation*, *ArXiv* **abs/2201.08825** (2022).

[9] A. Wu, H. Zhang, G. Li, A. Shabani, Y. Xie, and Y. Ding, *Autocomm: A framework for enabling efficient communication in distributed quantum programs*, *arXiv preprint arXiv:2207.11674* (2022).

[10] A. Wu, Y. Ding, and A. Li, *Qucomm: Optimizing collective communication for distributed quantum computing*, in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[11] A. Wu, G. Li, H. Zhang, G. G. Guerreschi, Y. Ding, and Y. Xie, *A synthesis framework for stitching surface code with superconducting quantum devices*,

Proceedings of the 49th Annual International Symposium on Computer Architecture (2022).

[12] A. Wu, K. Yin, A. W. Cross, A. Li, and Y. Ding, *Enabling full-stack quantum computing with changeable error-corrected qubits*, *arXiv preprint arXiv:2305.07072* (2023).

[13] A. Wu, G. Li, H. Zhang, G. G. Guerreschi, Y. Xie, and Y. Ding, *Qecv: Quantum error correction verification*, *arXiv preprint arXiv:2111.13728* (2021).

[14] J. Eisert, K. Jacobs, P. Papadopoulos, and M. B. Plenio, *Optimal local implementation of nonlocal quantum gates*, *Physical Review A* **62** (2000), no. 5 052317.

[15] P. W. Shor, *Scheme for reducing decoherence in quantum computer memory*, *Physical review A* **52** (1995), no. 4 R2493.

[16] A. M. Steane, *Error correcting codes in quantum theory*, *Physical Review Letters* **77** (1996), no. 5 793.

[17] A. R. Calderbank and P. W. Shor, *Good quantum error-correcting codes exist*, *Physical Review A* **54** (1996), no. 2 1098.

[18] A. Steane, *Multiple-particle interference and quantum error correction*, *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **452** (1996), no. 1954 2551–2577.

[19] S. B. Bravyi and A. Y. Kitaev, *Quantum codes on a lattice with boundary*, *arXiv preprint quant-ph/9811052* (1998).

[20] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, *Surface codes: Towards practical large-scale quantum computation*, *Physical Review A* **86** (2012), no. 3 032324.

[21] D. Gottesman, *Class of quantum error-correcting codes saturating the quantum hamming bound*, *Physical Review A* **54** (1996), no. 3 1862.

[22] A. R. Calderbank, E. M. Rains, P. W. Shor, and N. J. Sloane, *Quantum error correction and orthogonal geometry*, *Physical Review Letters* **78** (1997), no. 3 405.

[23] E. Knill, R. Laflamme, and L. Viola, *Theory of quantum error correction for general noise*, *Physical Review Letters* **84** (2000), no. 11 2525.

[24] B. Eastin and E. Knill, *Restrictions on transversal encoded quantum gate sets*, *Physical Review Letters* **102** (Mar, 2009) 110502.

[25] S. Bravyi and A. Kitaev, *Universal quantum computation with ideal clifford gates and noisy ancillas*, *Phys. Rev. A* **71** (Feb, 2005) 022316.

[26] J. T. Anderson, G. Duclos-Cianci, and D. Poulin, *Fault-tolerant conversion between the steane and reed-muller quantum codes*, *Phys. Rev. Lett.* **113** (Aug, 2014) 080501.

[27] M. E. Beverland, A. Kubica, and K. M. Svore, *Cost of universality: A comparative study of the overhead of state distillation and code switching with color codes*, *PRX Quantum* **2** (Jun, 2021) 020341.

[28] M. S. ANIS, Abby-Mitchell, H. Abraham, AduOffei, R. Agarwal, G. Agliardi, M. Aharoni, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, M. Amy, S. Anagolum, Anthony-Gandon, E. Arbel, A. Asfaw, A. Athalye, A. Avkhadiev, C. Azaustre, P. BHOLE, A. Banerjee, S. Banerjee, W. Bang, A. Bansal, P. Barkoutsos, A. Barnawal, G. Barron, G. S. Barron, L. Bello, Y. Ben-Haim, M. C. Bennett, D. Bevenius, D. Bhatnagar, A. Bhobe, P. Bianchini, L. S. Bishop, C. Blank, S. Bolos, S. Bopardikar, S. Bosch, S. Brandhofer, Brandon, S. Bravyi, N. Bronn, Bryce-Fuller, D. Bucher, A. Burov, F. Cabrera, P. Calpin, L. Capelluto, J. Carballo, G. Carrascal, A. Carriker, I. Carvalho, A. Chen, C.-F. Chen, E. Chen, J. C. Chen, R. Chen, F. Chevallier, K. Chinda, R. Cholarajan, J. M. Chow, S. Churchill, CisterMoke, C. Claus, C. Clauss, C. Clothier, R. Cocking, R. Cocuzzo, J. Connor, F. Correa, Z. Crockett, A. J. Cross, A. W. Cross, S. Cross, J. Cruz-Benito, C. Culver, A. D. Córcoles-Gonzales, N. D, S. Dague, T. E. Dandachi, A. N. Dangwal, J. Daniel, M. Daniels, M. Dartiailh, A. R. Davila, F. Debouni, A. Dekusar, A. Deshmukh, M. Deshpande, D. Ding, J. Doi, E. M. Dow, E. Drechsler, E. Dumitrescu, K. Dumon, I. Duran, K. EL-Safty, E. Eastman, G. Eberle, A. Ebrahimi, P. Eendebak, D. Egger, ElePT, Emilio, A. Espiricueta, M. Everitt, D. Facoetti, Farida, P. M. Fernández, S. Ferracin, D. Ferrari, A. H. Ferrera, R. Fouilland, 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, J. R. Garrison, J. Garrison, T. Gates, L. Gil, A. Gilliam, A. Giridharan, J. Gomez-Mosquera, Gonzalo, S. de la Puente González, J. Gorzinski, I. Gould, D. Greenberg, D. Grinko, W. Guan, D. Guijo, J. A. Gunnels, H. Gupta, N. Gupta, J. M. Günther, M. Haglund, I. Haide, I. Hamamura, O. C. Hamido, F. Harkins, K. Hartman, A. Hasan, 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, Ishwor, R. Iten, T. Itoko, A. Ivrii, A. Javadi, A. Javadi-Abhari, W. Javed, Q. Jianhua, M. Jivrajani, K. Johns, S. Johnstun, Jonathan-Shoemaker, JosDenmark, JoshDumo, J. Judge, T. Kachmann, A. Kale, N. Kanazawa, J. Kane, Kang-Bae, A. Kapila, A. Karazeev, P. Kassebaum, T. Kehrer, J. Kelso, S. Kelso, V. Khanderao, S. King, Y. Kobayashi, Kovi11Day, A. Kovyrshin,

R. Krishnakumar, V. Krishnan, K. Krsulich, P. Kumkar, G. Kus, R. LaRose, E. Lacal, R. Lambert, H. Landa, J. Lapeyre, J. Latone, S. Lawrence, C. Lee, G. Li, J. Lishman, D. Liu, P. Liu, Lolcroc, A. K. M, L. Madden, Y. Maeng, S. Maheshkar, K. Majmudar, A. Malyshev, M. E. Mandouh, J. Manela, Manjula, J. Marecek, M. Marques, K. Marwaha, D. Maslov, P. Maszota, D. Mathews, A. Matsuo, F. Mazhandu, D. McClure, M. McElaney, C. McGarry, D. McKay, D. McPherson, S. Meesala, D. Meirom, C. Mendell, T. Metcalfe, M. Mevissen, A. Meyer, A. Mezzacapo, R. Midha, D. Miller, Z. Minev, A. Mitchell, N. Moll, A. Montanez, G. Monteiro, M. D. Mooring, R. Morales, N. Moran, D. Morcuende, S. Mostafa, M. Motta, R. Moyard, P. Murali, J. Müggenburg, T. NEMOZ, D. Nadlinger, K. Nakanishi, G. Nannicini, P. Nation, E. Navarro, Y. Naveh, S. W. Neagle, P. Neuweiler, A. Ngoueya, T. Nguyen, J. Nicander, Nick-Singstock, P. Niroula, H. Norlen, NuoWenLei, L. J. O'Riordan, O. Ogunbayo, P. Ollitrault, T. Onodera, R. Otaolea, S. Oud, D. Padilha, H. Paik, S. Pal, Y. Pang, A. Panigrahi, V. R. Pascuzzi, S. Perriello, E. Peterson, A. Phan, K. Pilch, F. Piro, M. Pistoia, C. Piveteau, J. Plewa, P. Pocreau, A. Pozas-Kerstjens, R. Pracht, M. Prokop, V. Prutyanov, S. Puri, D. Puzzuoli, J. Pérez, Quant02, Quintiii, R. I. Rahman, A. Raja, R. Rajeev, I. Rajput, N. Ramagiri, A. Rao, R. Raymond, O. Reardon-Smith, R. M.-C. Redondo, M. Reuter, J. Rice, M. Riedemann, Rietesh, D. Risinger, M. L. Rocca, D. M. Rodríguez, RohithKarur, B. Rosand, M. Rossmannek, M. Ryu, T. SAPV, N. R. C. Sa, A. Saha, A. Ash-Saki, S. Sanand, M. Sandberg, H. Sandesara, R. Sapra, H. Sargsyan, A. Sarkar, N. Sathaye, B. Schmitt, C. Schnabel, Z. Schoenfeld, T. L. Scholten, E. Schoute, M. Schulterbrandt, J. Schwarm, J. Seaward, Sergi, I. F. Sertage, K. Setia, F. Shah, N. Shammah, R. Sharma, Y. Shi, J. Shoemaker, A. Silva, A. Simonetto, D. Singh, D. Singh, P. Singh, P. Singkanipa, Y. Siraichi, Siri, J. Sistos, I. Sitdikov, S. Sivarajah, M. B. Sletfjerding, J. A. Smolin, M. Soeken, I. O. Sokolov, I. Sokolov, V. P. Soloviev, SooluThomas, Starfish, D. Steenken, M. Stypulkoski, A. Suau, S. Sun, K. J. Sung, M. Suwama, O. Słowik, H. Takahashi, T. Takawale, I. Tavernelli, C. Taylor, P. Taylour, S. Thomas, K. Tian, M. Tillet, M. Tod, M. Tomasik, C. Tornow, E. de la Torre, J. L. S. Toural, K. Trabing, M. Treinish, D. Trenev, TrishaPe, F. Truger, G. Tsilimigkounakis, D. Tulsi, W. Turner, Y. Vaknin, C. R. Valcarce, F. Varchon, A. Vartak, A. C. Vazquez, P. Vijaywargiya, V. Villar, B. Vishnu, D. Vogt-Lee, C. Vuillot, J. Weaver, J. Weidenfeller, R. Wieczorek, J. A. Wildstrom, J. Wilson, E. Winston, WinterSoldier, J. J. Woehr, S. Woerner, R. Woo, C. J. Wood, R. Wood, S. Wood, J. Wootton, M. Wright, L. Xing, J. YU, B. Yang, U. Yang, J. Yao, D. Yeralin, R. Yonekura, D. Yonge-Mallo, R. Yoshida, R. Young, J. Yu, L. Yu, C. Zachow, L. Zdanski, H. Zhang, I. Zidaru, C. Zoufal, aeddins ibm, alexzhang13, b63, bartek bartlomiej, bcamorrison, brandhsn, charmerDark, deeplokhande, dekel.meirom, dime10, dlasecki, ehchen, fanizzamarco, fs1132429, gadial, galeinston, georgezhou20, georgios ts, gruu, hhorii, hykavitha, itoko,

jeppevinkel, jessica angel7, jezerjojo14, jliu45, jscott2, klinvill, krutik2966, ma5x, michelle4654, msuwama, nico lgrs, ntgiwsvp, ordmoj, sagar pahwa, pritamsinha2304, ryancocuzzo, saktar unr, saswati qiskit, septembrr, sethmerkel, sg495, shaashwat, smturro2, sternparky, strickroman, tigerjack, tsura crisaldo, upsideon, vadebayo49, welien, willhbang, wmurphy collabstar, yang.luh, and M. Čepulkovskis, *Qiskit: An open-source framework for quantum computing*, 2021.

[29] 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.

[30] C. Chamberland, G. Zhu, T. J. Yoder, J. Hertzberg, and A. Cross, *Topological and subsystem codes on low-degree graphs with flag qubits*, *Physical Review X* **10** (Jan, 2020) 011022.

[31] L. Lao and C. G. Almudéver, *Fault-tolerant quantum error correction on near-term quantum processors using flag and bridge qubits*, *Physical Review A* **101** (Mar, 2020) 032333.

[32] R. Chao and B. Reichardt, *Flag fault-tolerant error correction for any stabilizer code*, *arXiv: Quantum Physics* (2019).

[33] C. Chamberland and M. Beverland, *Flag fault-tolerant error correction with arbitrary distance codes*, *arXiv: Quantum Physics* **2** (2017) 53.

[34] C. Chamberland, A. Kubica, T. J. Yoder, and G. Zhu, *Triangular color codes on trivalent graphs with flag qubits*, *arXiv: Quantum Physics* **22** (Feb, 2019) 023019.

[35] 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, 2017.

[36] D. Gottesman and I. L. Chuang, *Quantum teleportation is a universal computational primitive*, *arXiv preprint quant-ph/9908010* (1999).

[37] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, *SIAM review* **41** (1999), no. 2 303–332.

[38] 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, 1996.

[39] 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.

[40] 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.

[41] 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.

[42] J. M. Baker, C. Duckering, A. Hoover, and F. T. Chong, *Time-sliced quantum circuit partitioning for modular architectures*, *Proceedings of the 17th ACM International Conference on Computing Frontiers* (2020).

[43] C. Young, A. Safari, P. Huft, J. Zhang, E. Oh, R. Chinnarasu, and M. Saffman, *An architecture for quantum networking of neutral atom processors*, *Applied Physics B* **128** (2022).

[44] A. Yimsiriwattana and S. J. Lomonaco Jr, *Generalized ghz states and distributed quantum computing*, *arXiv preprint quant-ph/0402148* (2004).

[45] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi, *Compiler design for distributed quantum computing*, *IEEE Transactions on Quantum Engineering* **2** (2021) 1–20.

[46] G. Li, Y. Ding, and Y. Xie, *Tackling the qubit mapping problem for nisq-era quantum devices*, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[47] M. Amy and V. Gheorghiu, *staq—a full-stack quantum processing toolkit*, *arXiv: Quantum Physics* (2019).

[48] N. Khammassi, I. Ashraf, J. van 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*, *ACM J. Emerg. Technol. Comput. Syst.* **18** (2022) 13:1–13:24.

[49] 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* (2020).

[50] M. Van Steen and A. Tanenbaum, *Distributed systems principles and paradigms*, *Network* **2** (2002) 28.

[51] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, *RevLib: An online resource for reversible functions and reversible circuits*, in *Int'l Symp. on Multi-Valued Logic*, pp. 220–225, 2008. RevLib is available at http://www.revlib.org.

[52] E. Farhi, J. Goldstone, and S. Gutmann, *A quantum approximate optimization algorithm*, *arXiv: Quantum Physics* (2014).

[53] Y. S. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. L. Maslov, *Automated optimization of large quantum circuits with continuous parameters*, *npj Quantum Information* **4** (May, 2017) 1–12.

[54] N. Isailovic, Y. Patel, M. Whitney, and J. Kubiatowicz, *Interconnection networks for scalable quantum computers*, in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 366–377, IEEE, 2006.

[55] R. S. Correa and J. P. David, *Ultra-low latency communication channels for fpga-based hpc cluster*, *Integration* **63** (2018) 41–55.

[56] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, *A quantum engineer's guide to superconducting qubits*, *Applied Physics Reviews* **6** (2019), no. 2 021318.

[57] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, *Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers*, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[58] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, *et. al.*, *Quantum supremacy using a programmable superconducting processor*, *Nature* **574** (2019), no. 7779 505–510.

[59] 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, *et. al.*, *Quantum computational advantage using photons*, *Science* **370** (2020), no. 6523 1460–1463.

[60] M. Pompili, S. L. Hermans, S. Baier, H. K. Beukers, P. C. Humphreys, R. N. Schouten, R. F. Vermeulen, M. J. Tiggelman, L. dos Santos Martins, B. Dirkse, *et. al.*, *Realization of a multinode quantum network of remote solid-state qubits*, *Science* **372** (2021), no. 6539 259–264.

[61] S. Daiss, S. Langenfeld, S. Welte, E. Distante, P. Thomas, L. Hartung, O. Morin, and G. Rempe, *A quantum-logic gate between distant quantum-network modules*, *Science* **371** (2021), no. 6529 614–617.

[62] M. Ruf, N. H. Wan, H. Choi, D. Englund, and R. Hanson, *Quantum networks based on color centers in diamond*, *Journal of Applied Physics* **130** (2021), no. 7 070901.

[63] R. Beals, S. Brierley, O. Gray, A. W. Harrow, S. Kutin, N. Linden, D. J. Shepherd, and M. J. Stather, *Efficient distributed quantum computing*, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **469** (2013).

[64] M. Z. Moghadam, M. Houshmand, and M. Houshmand, *Optimizing teleportation cost in distributed quantum circuits*, *International Journal of Theoretical Physics* **57** (2016) 848–861.

[65] Z. Davarzani, M. Z. Moghadam, M. Houshmand, and M. Nouri, *A dynamic programming approach for distributing quantum circuits by bipartite graphs*, *Quantum Inf. Process.* **19** (2020) 360.

[66] O. Daei, K. Navi, and M. Zomorodi-Moghadam, *Optimized quantum circuit partitioning*, *International Journal of Theoretical Physics* **59** (2020), no. 12 3804–3820.

[67] D. Dadkhah, M. Zomorodi, S. E. Hosseini, P. Plawiak, and X. Zhou, *Reordering and partitioning of distributed quantum circuits*, *IEEE Access* **10** (2022) 70329–70341.

[68] T. Häner, D. S. Steiger, T. Hoefler, and M. Troyer, *Distributed quantum computing with qmpi*, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021).

[69] S. L. Braunstein, *Quantum error correction for communication with linear optics*, *Nature* **394** (1998), no. 6688 47–49.

[70] J.-W. Pan, C. Simon, Č. Brukner, and A. Zeilinger, *Entanglement purification for quantum communication*, *Nature* **410** (2001), no. 6832 1067–1070.

[71] T. Park and C. Y. Lee, *Algorithms for partitioning a graph*, *Computers & Industrial Engineering* **28** (1995), no. 4 899–909.

[72] R. V. Meter, W. Munro, K. Nemoto, and K. M. Itoh, *Arithmetic on a distributed-memory quantum multicomputer*, *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **3** (2008), no. 4 1–23.

[73] M. Sarvaghad-Moghaddam and M. Zomorodi, *A general protocol for distributed quantum gates*, *Quantum Information Processing* **20** (2021), no. 8 1–14.

[74] 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.

[75] Y. Zhao, G. Zhao, and C. Qiao, *E2e fidelity aware routing and purification for throughput maximization in quantum networks*, in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pp. 480–489, IEEE, 2022.

[76] P. Kok, W. J. Munro, K. Nemoto, T. C. Ralph, J. P. Dowling, and G. J. Milburn, *Linear optical quantum computing with photonic qubits*, *Reviews of Modern Physics* **79** (2007) 135–174.

[77] D. P. DiVincenzo and Ibm, *The physical implementation of quantum computation*, *Protein Science* **48** (2000) 771–783.

[78] T. Asselmeyer-Maluga, *3d topological quantum computing*, *International Journal of Quantum Information* (2021).

[79] J. Kelly, "A Preview of Bristlecone, Google's New Quantum Processor." https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, 2017.

[80] H. Paik, D. I. Schuster, L. Bishop, G. Kirchmair, G. Catelani, A. P. Sears, B. R. Johnson, M. 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.*, *Physical review letters* **107 24** (2011) 240501.

[81] Y. Chen, C. J. Neill, P. Roushan, N. Leung, M. Fang, R. Barends, J. Kelly, B. Campbell, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, A. Megrant, J. Mutus, P. J. J. O'Malley, C. Quintana, D. T. Sank, A. Vainsencher, J. Wenner, T. White, M. R. Geller, A. N. Cleland, and J. M. Martinis, *Qubit architecture with high coherence and fast tunable coupling.*, *Physical review letters* **113 22** (2014) 220502.

[82] E. J. Zhang, S. Srinivasan, N. Sundaresan, D. F. Bogorin, Y. Martin, J. B. Hertzberg, J. Timmerwilke, E. J. Pritchett, J.-B. Yau, C. Wang, *et. al.*, *High-fidelity superconducting quantum processors via laser-annealing of transmon qubits*, *arXiv preprint arXiv:2012.08475* (2020).

[83] A. Gold, A. Stockklauser, M. Reagor, J.-P. Paquette, A. Bestwick, C. J. Winkleblack, B. Scharmann, F. Oruc, and B. Langley, *Experimental demonstration of entangling gates across chips in a multi-core qpu*, *Bulletin of the American Physical Society* (2021).

[84] Y. Ding, A. Holmes, A. JavadiAbhari, D. Franklin, M. Martonosi, and F. T. Chong, *Magic-state functional units: Mapping and scheduling multi-level distillation circuits for fault-tolerant quantum architectures*, *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018) 828–840.

[85] A. Paler, *Surfbraid: A concept tool for preparing and resource estimating quantum circuits protected by the surface code*, *ArXiv* **abs/1902.02417** (2019).

[86] A. JavadiAbhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, *Optimized surface code communication in superconducting quantum computers*, *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017) 692–705.

[87] F. Hua, Y.-H. Chen, Y. Jin, C. Zhang, A. B. Hayes, Y. Zhang, and E. Z. Zhang, *Autobraid: A framework for enabling efficient surface code communication in quantum computing*, *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021).

[88] S. S. Tannu, Z. Myers, P. J. Nair, D. M. Carmean, and M. K. Qureshi, *Taming the instruction bandwidth of quantum computers via hardware-managed error correction*, *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017) 679–691.

[89] G. Li, Y. Ding, and Y. Xie, *Towards efficient superconducting quantum processor architecture design*, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[90] 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*, pp. 1015–1029, 2019.

[91] B. Tan and J. Cong, *Optimal layout synthesis for quantum computing*, in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[92] 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** (2018), no. 7 1226–1236.

[93] R. Wille, L. Burgholzer, and A. Zulehner, *Mapping quantum circuits to ibm qx architectures using the minimal number of swap and h operations*, in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2019.

[94] M. Y. Siraichi, V. F. d. Santos, S. Collange, and F. M. Q. Pereira, *Qubit allocation*, in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 113–125, 2018.

[95] W. Finigan, M. Cubeddu, T. Lively, J. Flick, and P. Narang, *Qubit allocation for noisy intermediate-scale quantum computers*, *arXiv preprint arXiv:1810.08291* (2018).

[96] 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*, pp. 1001–1014, 2019.

[97] 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.

[98] R. Chao and B. Reichardt, *Fault-tolerant quantum computation with few qubits*, *npj Quantum Information* **4** (2017) 1–8.

[99] C. Gidney, *Stim: a fast stabilizer circuit simulator*, *Quantum* **5** (July, 2021) 497.

[100] O. Higgott, *PyMatching: A python package for decoding quantum codes with minimum-weight perfect matching*, *arXiv preprint arXiv:2105.13082* (2021).

[101] P. Jurcevic, A. Javadi-Abhari, L. Bishop, I. Lauer, D. Bogorin, M. Brink, L. Capelluto, O. Günlük, T. Itoko, N. Kanazawa, A. Kandala, G. Keefe, K. D. Krsulich, W. Landers, E. Lewandowski, D. McClure, G. Nannicini, A. Narasgond, H. Nayfeh, E. Pritchett, M. Rothwell, S. Srinivasan, N. Sundaresan, C. Wang, K. X. Wei, C. J. Wood, J. Yau, E. Zhang, O. Dial, J. Chow, and J. Gambetta, *Demonstration of quantum volume 64 on a superconducting quantum computing system*, *Quantum Science & Technology* **6** (2020).

[102] A. Kandala, K. X. Wei, S. Srinivasan, E. Magesan, S. Carnevale, G. A. Keefe, D. Klaus, O. Dial, and D. C. McKay, *Demonstration of a high-fidelity cnot gate for fixed-frequency transmons with engineered zz suppression*, *Phys. Rev. Lett.* **127** (Sep, 2021) 130501.

[103] J. Preskill, *Quantum computing in the NISQ era and beyond*, *Quantum* **2** (aug, 2018) 79.

[104] N. Sundaresan, T. J. Yoder, Y. Kim, M. Li, E. H. Chen, G. Harper, T. Thorbeck, A. W. Cross, A. D. Córcoles, and M. Takita, *Matching and maximum likelihood decoding of a multi-round subsystem quantum error correction experiment*, *arXiv preprint arXiv:2203.07205* (2022).

[105] G. Q. AI, *Suppressing quantum errors by scaling a surface code logical qubit*, *Nature* **614** (2023), no. 7949 676–681.

[106] C. Ryan-Anderson, N. Brown, M. Allman, B. Arkin, G. Asa-Attuah, C. Baldwin, J. Berg, J. Bohnet, S. Braxton, N. Burdick, *et. al.*, *Implementing fault-tolerant entangling gates on the five-qubit code and the color code*, *arXiv preprint arXiv:2208.01863* (2022).

[107] 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.

[108] L. Egan, D. M. Debroy, C. Noel, A. Risinger, D. Zhu, D. Biswas, M. Newman, M. Li, K. R. Brown, M. Cetina, *et. al.*, *Fault-tolerant operation of a quantum error-correction code*, *arXiv preprint arXiv:2009.11482* (2020).

[109] R. Iten, R. Colbeck, I. Kukuljan, J. Home, and M. Christandl, *Quantum circuits for isometries*, *Phys. Rev. A* **93** (Mar, 2016) 032318.

[110] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang, *Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation*, *ACM Transactions on Quantum Computing* (2022).

[111] C. Chamberland and T. Jochym-O'Connor, *Error suppression via complementary gauge choices in reed-muller codes*, *Quantum Science and Technology* **2** (2017), no. 3 035008.

[112] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, *Revlib: An online resource for reversible functions and reversible circuits*, in *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, pp. 220–225, IEEE, 2008.

[113] A. Kissinger and J. van de Wetering, *Pyzx: Large scale automated diagrammatic reasoning*, *arXiv preprint arXiv:1904.04735* (2019).

[114] C. Chamberland and P. Ronagh, *Deep neural decoders for near term fault-tolerant experiments*, *Quantum Science and Technology* **3** (jul, 2018) 044002.

[115] N. C. Jones, R. Van Meter, A. G. Fowler, P. L. McMahon, J. Kim, T. D. Ladd, and Y. Yamamoto, *Layered architecture for quantum computing*, *Physical Review X* **2** (2012), no. 3 031007.

[116] Z. Chen, A. Megrant, J. Kelly, R. Barends, J. Bochmann, Y. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, J. Mutus, *et. al.*, *Fabrication and characterization of aluminum airbridges for superconducting microwave circuits*, *Applied Physics Letters* **104** (2014), no. 5.

[117] J. Preskill, *Quantum computing in the nisq era and beyond*, *Quantum* (2018).

[118] A. Holmes, M. R. Jokar, G. Pasandi, Y. Ding, M. Pedram, and F. T. Chong, *Nisq+: Boosting quantum computing power by approximating quantum error correction*, *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020) 556–569.

[119] E. D'Hondt and P. Panangaden, *Quantum weakest preconditions*, *Mathematical Structures in Computer Science* **16** (2006) 429–451.

[120] M. Ying, *Floyd–hoare logic for quantum programs*, *ACM Trans. Program. Lang. Syst.* **33** (2012) 19:1–19:49.

[121] M. Ying and Y. Li, *Reasoning about parallel quantum programs*, *ArXiv* **abs/1810.11334** (2018).

[122] Y. Feng and M. Ying, *Quantum hoare logic with classical variables*, *ArXiv* **abs/2008.06812** (2020).

[123] Y. Feng, S. Li, and M. Ying, *Verification of distributed quantum programs*, *ArXiv* **abs/2104.14796** (2021).

[124] L. Zhou, N. Yu, and M. Ying, *An applied quantum hoare logic*, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).

[125] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, *Projection-based runtime assertions for testing and debugging quantum programs*, *Proceedings of the ACM on Programming Languages* **4** (2020) 1–29.

[126] P. W. Shor, *Quantum computing*, *Documenta Mathematica* **1** (1998), no. 1000 1.

[127] A. M. Steane, *Quantum reed-muller codes*, *IEEE Transactions on Information Theory* **45** (1999), no. 5 1701–1703.

[128] V. P. Su, C. Cao, H.-Y. Hu, Y. Yanay, C. Tahan, and B. Swingle, *Discovery of optimal quantum error correcting codes via reinforcement learning*, *arXiv preprint arXiv:2305.06378* (2023).

[129] K. Yin, H. Zhang, Y. Shi, T. Humble, A. Li, and Y. Ding, *Optimal synthesis of stabilizer codes via maxsat*, *arXiv preprint arXiv:2308.06428* (2023).

[130] M. M. Wilde, *Quantum information theory.* Cambridge university press, 2013.

[131] W. Tang, T. Tomesh, M. Suchara, J. Larson, and M. Martonosi, *Cutqc: using small quantum computers for large quantum circuit evaluations*, in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 473–486, 2021.

[132] T. Peng, A. W. Harrow, M. A. Ozols, and X. Wu, *Simulating large quantum circuits on a small quantum computer.*, *Physical review letters* **125 15** (2020) 150504.

[133] D. W. Walker and J. J. Dongarra, *Mpi: a standard message passing interface*, *Supercomputer* **12** (1996) 56–68.

[134] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever, *Mapping of lattice surgery-based quantum circuits on surface code architectures*, *Quantum Science and Technology* (2018).

[135] B. Reichardt, *Fault-tolerant quantum error correction for steane's seven-qubit color code with few or no extra qubits.*, *arXiv: Quantum Physics* **6** (Jan, 2018) 015007.

[136] R. Chadha, P. Mateus, and A. Sernadas, *Reasoning about imperative quantum programs*, in *MFPS*, 2006.

[137] Y. Kakutani, *A logic for formal verification of quantum programs*, in *ASIAN*, 2009.

[138] P. Selinger, *Towards a quantum programming language*, *Mathematical Structures in Computer Science* **14** (2004) 527–586.

[139] D. Unruh, *Quantum hoare logic with ghost variables*, *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (2019) 1–13.

[140] F. Voichick, L. Li, R. Rand, and M. Hicks, *Qunity: A unified language for quantum and classical computing*, in *Proceedings of the ACM on Programming Languages*, vol. 5, 2023.

[141] C. Chareton, S. Bardin, F. Bobot, V. Perrelle, and B. Valiron, *An automated deductive verification framework for circuit-building quantum programs*, *Programming Languages and Systems* **12648** (2021) 148–177.

[142] M. Amy, *Towards large-scale functional verification of universal quantum circuits*, *CoRR* **abs/1805.06908** (2018) 1–21.

[143] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. W. Hicks, *A verified optimizer for quantum circuits*, *Proceedings of the ACM on Programming Languages* **5** (2021) 1–29.

[144] A. Sundaram, R. Rand, K. Singhal, and B. Lackey, *A rich type system for quantum programs*, 2022.

[145] R. Rand, A. Sundaram, K. Singhal, and B. Lackey, *Extending gottesman types beyond the clifford group*, in *The Second International Workshop on Programming Languages for Quantum Computing (PLanQC 2021)*, 2021.

[146] R. Rand, A. Sundaram, K. Singhal, and B. Lackey, *Gottesman types for quantum programs*, in *Proceedings of the 17th International Conference on Quantum Physics and Logic, QPL*, vol. 20, 2020.

[147] D. Gottesman, *Stabilizer codes and quantum error correction*, *arXiv: Quantum Physics* (1997).

[148] N. Yu and J. Palsberg, *Quantum abstract interpretation*, in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 542–558, 2021.