Cycle-Aware Parallel Optimization for Mitigating ZZ Crosstalk on Quantum Hardware

Jiayi Zhong
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University
Shanghai, China
jiayialice323@gmail.com

Yuxin Deng
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University and MoE Key Laboratory of
Interdisciplinary Research of Computation and Economics,
Shanghai University of Finance and Economics
Shanghai, China
yxdeng@msg.sufe.edu.cn

Abstract

ZZ crosstalk and decoherence hinder superconducting quantum computing. Mitigation strategies often require sequential gate execution, which restricts parallelism. We reformulate ZZ crosstalk mitigation as a parallel task scheduling problem by integrating quantum cycles and qubit interference. We then propose CYCO, a **CY**cle-aware ZZ **C**rosstalk **O**ptimization algorithm, which uses a timing-based greedy strategy to schedule gates through cycles within quantum circuits. A novel data structure called Time and Distance Dependency Graph (TDDG) is designed to model gate dependencies and physical qubit distances. Based on TDDG, barrier punching is introduced to eliminate redundant synchronization barriers by merging independent gate groups, improving gate concurrency per cycle. Simulations show a reduction of up to 37.44% in quantum program cycle (14.19% on average) on 53- to 127-qubit NISQ devices, with up to 1.6× higher parallelism than state-of-the-art methods. Real-device experiments demonstrate significant acceleration in quantum computing while maintaining fidelity.

CCS Concepts

• **Computer systems organization → Quantum computing**.

Keywords

Quantum Computing, Crosstalk Mitigation, Superconducting Quantum Computer

ACM Reference Format:
Jiayi Zhong and Yuxin Deng. 2025. Cycle-Aware Parallel Optimization for Mitigating ZZ Crosstalk on Quantum Hardware. In *ICPP '25: International Conference on Parallel Processing, September 08–11, 2025, San Diego, CA.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Quantum computing represents a transformative paradigm, capable of addressing complex computational challenges beyond classical systems, as exemplified by Shor's factorization and Grover's search algorithms [6, 27]. While diverse quantum platforms (e.g. neutral atoms, trapped ions) are emerging, superconducting quantum processors positioned as dominant Noisy Intermediate-Scale Quantum (NISQ) technology remain pivotal due to their scalability and industrial viability. However, these devices are hindered by noise sources that severely compromise computational accuracy and reliability.

ZZ crosstalk, characterized by unwanted $\sigma_z \otimes \sigma_z$ couplings between superconducting qubits, induces phase errors even in the absence of gate operations, as noted in engineering studies [14, 22]. Despite the implementation of tunable couplers, quantum processors such as Google's Bristlecone [1] and IBM's Eagle [13] continue to experience residual ZZ interactions and fidelity drops due to crosstalk in parallel gate executions, respectively. These challenges highlight that tunable couplers alone cannot fully eliminate ZZ crosstalk. Decoherence further compounds these issues by limiting the operational window of quantum circuits, as longer execution times result in increased fidelity degradation [21, 26].

The presence of ZZ crosstalk and decoherence often necessitates sequential gate execution to mitigate interference, significantly limiting parallelism. This is analogous to classical computing, where the instruction cycle scheduling aims to maximize parallel execution to minimize runtime. In quantum computing, gate operations are executed via pulse signals with varying durations based on hardware specifications [7], synchronized through quantum clock cycles. However, the rigid approach of inserting full barriers, as in existing methods such as ZZXSched [29] for ZZ crosstalk mitigation, increases idle time and causes parallel resource underutilization. For example, if the longest gate in a set takes 10 times longer than the others, the shorter gates must idle, reducing parallel efficiency of the quantum circuit.

To overcome these limitations, we reformulate ZZ crosstalk mitigation as a parallel task scheduling problem, integrating quantum cycles and accounting for two forms of qubit interference: active-qubit interference and cross-qubit interference. Based on the above, we propose CYCO, a Cycle-aware ZZ Crosstalk Optimization algorithm, which is specifically developed to boost parallelism in quantum computing compilers. CYCO leverages a timing-based greedy strategy to intelligently schedule quantum gates across quantum cycles, effectively balancing the suppression of ZZ crosstalk with the minimization of execution time. By mapping gate pulses to discrete quantum cycle intervals, CYCO ensures precise alignment of operations, reducing qubit idle time and mitigating interference. Our contributions can be summarized as follows.

ICPP '25, San Diego, CA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

- We propose a quantum cycle model considering quantum cycles and qubit interference constraints, enabling parallel task scheduling under ZZ crosstalk mitigation.
- We introduce an innovative data structure, Time and Distance Dependency Graph (TDDG), to capture temporal and spatial dependencies between quantum gates. Barrier punching helps reduce sequential gate execution through quantum cycles. Based on the above, we propose a polynomial-time algorithm that optimizes the execution order of gates.
- Simulations on four superconducting quantum platforms demonstrate that CYCO reduces the total program time by up to 37.44% (average 14.19%) compared to state-of-the-art methods, while achieving a 1.6× increase in concurrency per cycle. As the benchmark size increases, our results confirm computational scalability. Experiments show that CYCO is architecture-independent and can be effectively implemented on systems with distinct qubit connectivity topologies.
- Real-device experiments on IBMQ-Brisbane confirm that CYCO maintains fidelity while significantly accelerating computations compared to ZZXSched.

The rest of the paper is organized as follows. Section 2 provides detailed review of quantum computing basics for quantum gate duration and ZZ crosstalk. Section 3 formally describes the cycle-aware ZZ crosstalk mitigation problem. Section 4 describes the key techniques and tools used in CYCO and presents the complete flow of the CYCO algorithm. Sections 5 and 6 evaluate the algorithm's performance on both simulated and real devices and illustrate the results of our algorithm respectively. Section 7 compares with previous work relevant to our research. Section 8 concludes this work.

2 Overview

This section provides essential background on the physical aspects of quantum computing, focusing on gate scheduling after circuit mapping onto target devices. An informal definition of ZZ crosstalk mitigation not considering the time property is introduced.

Physical Basis Gates. The hardware compiled quantum program combines physical basis gates that manipulate qubits on NISQ devices. Qubits are the fundamental units of quantum information, and physical gates act as hardware-level operations, similar to an instruction set architecture in classical computing. Common physical gates on superconducting platforms include single-qubit and two-qubit operations, such as iSWAP and CZ. Their matrix representations are as follows.

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

The iSWAP gate swaps two qubit states with a phase factor of $-i$, while the CZ gate applies a Z-phase shift when the control qubit is in state $|1\rangle$.

Gate Duration. In quantum computing, quantum gate duration refers to the time required for a gate to perform its operation on one or more qubits [24]. This duration varies significantly depending on the gate type and the hardware implementation. For example, on superconducting platforms, single-qubit gates typically have durations of around $20ns$, while multi-qubit gates, such as the Controlled-NOT (CNOT) gate, can take up to $200ns$ [3]. In quantum circuits, these durations contribute to the total execution time, accumulating along the sequence of operations. When gates are executed in parallel, their operations occur simultaneously within the same time step, with the duration of that step determined by the longest gate involved, as shown in Example 2.1. Consequently, variations in gate durations can increase the overall circuit execution time if not carefully optimized.

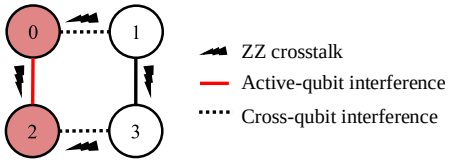


Figure 1: A 4-qubit NISQ device. Nodes represent qubits and edges show couplings.

Example 2.1. *Consider a quantum circuit with a two-qubit gate g_1 and a single-qubit gate g_2, with durations of $200ns$ and $20ns$, respectively. When executed in parallel, the total execution time τ is determined by the longer gate g_1, resulting in $\tau = 200ns$.*

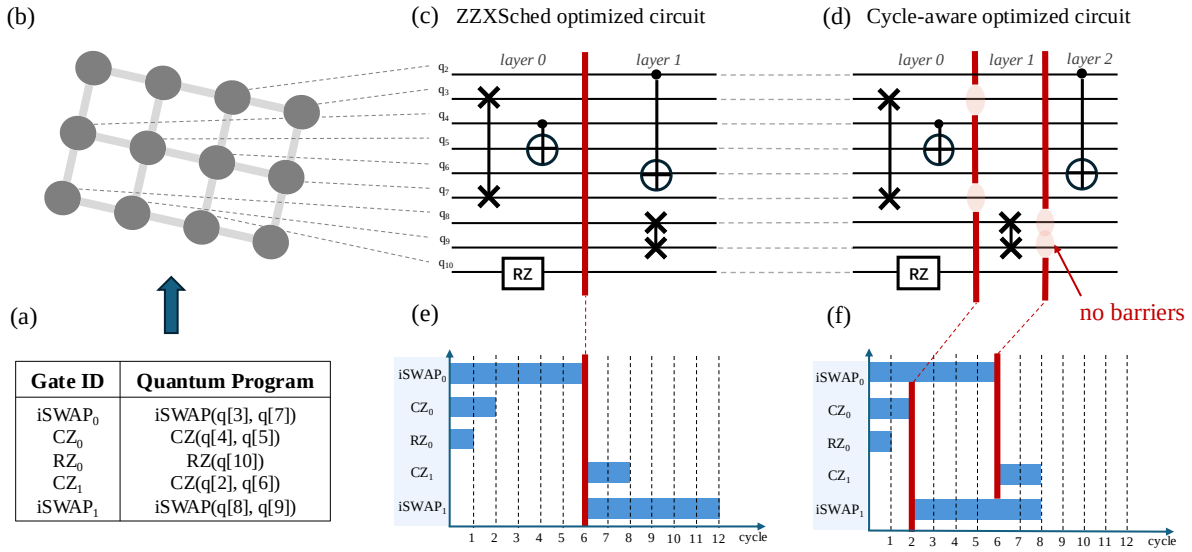
ZZ Crosstalk Mitigation — An Informal Overview. The spatial arrangement of physical qubits is fixed in superconducting quantum computers. As shown in Figure 1, qubits linked by couplings experience ZZ crosstalk due to unwanted interactions, even without active gate operations. This interference is inherent and unavoidable. Two types of qubit interference arise from ZZ crosstalk:

- **Active-qubit interference (I_A)**: Connections between qubits with excited quantum states generate I_A. More specifically, I_A persists regardless of whether gates are executed on these qubits. We call such qubits as active qubits. The intensity of this interference is determined by the density d_A, which represents the number of connections among the active qubits. A higher density d_A, reflecting a greater number of active qubits, increases the intensity of I_A.
- **Cross-qubit interference (I_C)**: Active qubits interfere with nearby idle qubits. The more physical links d_C between active and idle qubits, the stronger I_C becomes. Although less harmful than I_A, I_C still introduces performance-degrading dependencies.

The following example illustrates the relationship between I_A and I_C, highlighting various scenarios in which I_A emerges.

Example 2.2. *In the 4-qubit NISQ processor shown in Figure 1, qubits 0 and 2 are both active, causing I_A, indicated by the red edge. Additionally, qubit 3 is idle but close to active qubits, resulting in I_C through the dashed edges. Without mitigation, both I_A and I_C introduce phase errors and reduce fidelity.*

In general, I_A is more disruptive than I_C, but I_A can sometimes be reduced to I_C. Through pulse optimization, the harmful effects of



Gate ID	Quantum Program
$iSWAP_0$	iSWAP(q[3], q[7])
CZ_0	CZ(q[4], q[5])
RZ_0	RZ(q[10])
CZ_1	CZ(q[2], q[6])
$iSWAP_1$	iSWAP(q[8], q[9])

Figure 2: (a) An example quantum program. (b) The NISQ device topology with nearest-neighbor connectivity. (c) and (d) Red vertical lines represent barriers corresponding to qubit sleeping control for certain qubits to wait. In (d), barriers are removed (highlighted by ovals) to increase gate overlap. (e) and (f) Blue blocks show quantum clock cycles and red lines indicate barriers. In (f), cycle optimization allows $iSWAP_0$ and $iSWAP_1$ to execute parallel.

I_C can be reduced [29]. According to the above description, we give an informal definition of the ZZ crosstalk mitigation as follows:

Definition 2.1 (ZZ Crosstalk Mitigation). *Let QC be a quantum circuit executed on a device. A gate schedule S for QC should minimize the interference cost:*

$$\mathcal{J}(S) = I_A(S) + \alpha\, I_C(S) \tag{1}$$

where $I_A(S)$ is the active-qubit interference, $I_C(S)$ is the cross-qubit interference, and $\alpha > 0$ is a weighting factor.

3 Problem Definition

In this section, we formalize quantum cycles and the ZZ crosstalk mitigation problem based on prior knowledge. Motivating examples are given in this section.

3.1 Quantum Cycles

To analyze the efficiency of the quantum program, we develop a three-tiered quantum cycle model that addresses three interdependent factors: control system timing resolution (*clock cycles*), parallel gate durations (*layer cycles*), and total runtime (*program cycles*). While classical computing relies on uniform clock synchronization, quantum scheduling must simultaneously optimize these temporal evolutions to minimize both ZZ crosstalk and decoherence effects.

Our formalization builds on three fundamental components:

- **Quantum clock cycles (τ)** is the fundamental synchronization unit determined by control electronics, defining the minimum time resolution for scheduling operations (typically 1-10 ns in superconducting platforms).
- **Layer Cycle (λ_l)** represents the time required to execute a set of parallel quantum gates, determined by the gate with the longest duration in the group. This constraint arises from the hardware limitation that all parallel gates must wait for the slowest gate to complete before the next set of operations can begin.
- **Program cycle (Σ)** is the total execution time that accounts for both quantum parallelism and sequential dependencies, serving as the ultimate metric for the algorithm runtime.

We formalize these concepts considering *QC* composed of L gate layers:

Definition 3.1 (Quantum Cycle Model). *For any layer l with gates $G_l = \{g_1, ..., g_k\}$ containing parallel-executable gates:*

$$\lambda_l = \tau \cdot \max_{g \in G_l} \pi(g) \tag{2}$$

where $\pi(g)$ denotes the duration of gate g in quantum clock cycles. The total program cycle combines all layer cycles through:

$$\Sigma = \sum_{l=0}^{L} \lambda_l \tag{3}$$

This model reveals a critical trade-off: aggressive gate parallelization reduces the layer count but may increase individual λ_l through long-duration gates, while conservative scheduling minimizes λ_l at the cost of more layers. We give an example of the model below.

Example 3.1. *Figure 2 (e) demonstrates a gate allocation plan with five gates. The quantum clock cycle τ establishes the fundamental one-unit time grid (vertical dashed lines). In Layer 0, three gates are executed in parallel:* $iSWAP_0$ *(6 cycles),* CZ_0 *(1 cycle), and* RZ_0 *(2 cycles). The layer cycle $\lambda_0 = 6$ is dictated by* $iSWAP_0$*'s duration. The circuit ultimately achieves a total program cycle $\Sigma = 12\tau$.*

Table 1: Physical Basis Gate Duration Mapping.

Gate Type	Duration (cycles)
RZ gate	1
CZ gate	2
iSWAP gate	6

3.2 Cycle-Aware ZZ Crosstalk Mitigation

We demonstrate CYCO's scheduling advantages through a concrete example on a 3×4 grid topology (Figure 2 (a-b)). Logical qubits $0 \sim 11$ are mapped directly to physical qubits for clarity. Table 1 specifies the gate durations in our case.

Prior Scheduling Limitations. Conventional approaches like ZZXSched [29] use full barriers to partition gates into crosstalk-safe layers. As shown in Figure 2 (c), splitting our 5-gate circuit into two layers reduces ZZ crosstalk but creates significant idle periods. During iSWAP operation (6 cycles in q_3 and q_7), the neighboring qubits remain inactive due to the strict synchronization barriers in Figure 2 (e). This results in resource underutilization (4 cycles idle for q_8 and q_9) despite achieving 98% crosstalk suppression.

CYCO's Punching Barrier Technique. Our method introduces partial barriers that release qubits immediately after their gates are complete, while maintaining gate dependencies. Figure 2 (d) shows the key innovations of CYCO:

- **Early Gate Release**: Gates without data dependencies can bypass full-layer synchronization. After 2-cycle CZ_0 and 1-cycle RZ_0 finish in Layer 0, their qubits immediately begin to execute operation $iSWAP_1$ in Layer 1.
- **Selective Synchronization**: 6-cycle $iSWAP_0$ maintains dependencies for CZ_1, preventing ZZ crosstalk.

This strategic barrier removal reduces the total cycle from 12 to 8 (33% improvement) while maintaining equivalent crosstalk suppression. The optimized schedule uses three layers instead of two, demonstrating CYCO's ability to improve parallelism through layer fragmentation.

To capture the three-way optimization between the program cycle Σ, ZZ crosstalk $\mathcal{J}(S)$, and quantum parallelism, we formulate the problem in the following way.

Definition 3.2 (Cycle-Aware ZZ Mitigation Problem). *Given a set of qubits $Q = \{q_i\}_{i=0}^{n}$, a set of quantum gates $G = \{g_k\}$ with durations $\pi(g_k)$ and a dependency relation $E_D \subseteq G \times G$, the goal is to find a schedule S that divides G into layers L (with optional barrier sets BS) and minimizes the combined cost*

$$\min_{S} \; C(S) = \Sigma + \beta \mathcal{J}(S),$$

where Σ is the total program cycle, $\mathcal{J}(S)$ is the ZZ crosstalk error rate and $\beta > 0$ is a weighting factor.

The schedule S must satisfy:

- ***Connectivity Constraint**: For every two-qubit gate in S, the qubits involved must be neighbors on the quantum hardware.*
- ***Gate Dependency Constraint**: For every dependency $(g_i, g_j) \in E_D$, the assignment of the layer must satisfy $L(g_i) < L(g_j)$.*

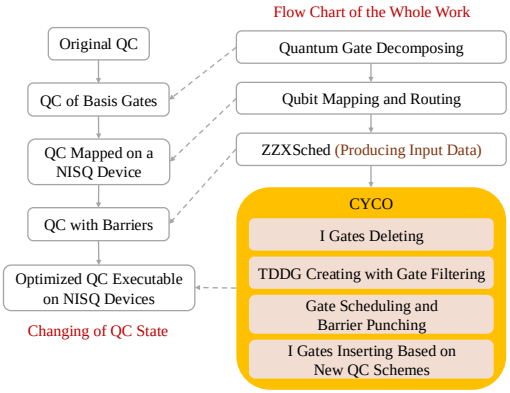


Figure 3: The flow chart illustrates the complete workflow. The right branch outlines the primary procedures of our approach, with the yellow block highlighting our innovative steps. Meanwhile, the left branch represents the state of the Quantum Circuit (QC) during the algorithm execution. The innovative steps are detailed in Section 4, while the preparatory steps in grey blocks are covered in Section 5.

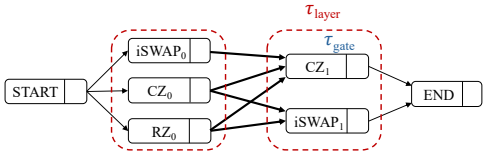


Figure 4: A TDDG example for the quantum program in Figure 2 (a). The red dotted box represents the layer of gates. τ equals to the finish time.

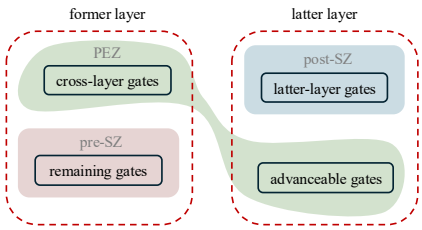


Figure 5: Illustration of Parallel Execution Zone (PEZ). The green block denotes PEZ containing cross-layer gates from the former layer and the advanceable gates from latter layer.

4 The Algorithm

In this section, we fully discuss how the TDDG data structure and the punching barrier strategy are efficient in the CYCO algorithm. The methodology is outlined in Figure 3, with the core process highlighted in yellow and explained in detail in this section. The gate scheduling details are demonstrated with the quantum cycles.

4.1 TDDG

We introduce the Time and Distance Dependency Graph (TDDG), the core data structure of the CYCO algorithm. TDDG is designed

to model spatio-temporal dependencies between quantum gates for efficient scheduling. This novel data structure ensures precise dependency tracking and parallel scheduling. Structured as a Directed Acyclic Graph (DAG), TDDG nodes represent gates, each with a Gate Finish Time (GFT) tracking completion. Edges denote two dependency types: *data dependency*, where gates share a qubit and must execute sequentially, and *distance dependency*, reflecting physical qubit proximity and enabling optimized separation of non-neighboring gates (see Figure 4, with thick edges for distance and thin for data dependencies). TDDG includes start and end nodes to manage qubit access and time. The start node links initial gates, marking execution's onset, while the end node connects final gates, recording total execution time for performance evaluation.

To streamline the algorithm, we define specific zones within TDDG. Cross-layer gates are those that outlast other gates in their layer, allowing them to run simultaneously with gates in subsequent layers. Advanceable gates from the next layer can execute earlier and run parallel to cross-layer gates from the previous layer if no dependencies between them (see Figure 5). These gates define the following zones: Parallel Execution Zone (PEZ) combines cross-layer gates and advanceable gates from adjacent layers for simultaneous execution; Pre-Scheduled Zone (Pre-SZ) holds non-selected gates ready for immediate execution; and Post-Scheduled Zone (Post-SZ) contains unready gates awaiting the next iteration for scheduling.

To identify cross-layer gates, each layer is associated with a *Layer's Maximum Finish Time (LMFT)*, which records the latest GFT between the gates within that layer. In addition, each gate is assigned a *Gate's Earliest Start Time (GEST)*, which defines the earliest possible time the gate can begin. A gate becomes a cross-layer gate if all its successor gates start after its own GFT. This allows the gate to run earlier than initially scheduled.

4.2 Preprocessing

CYCO applies gates generated from ZZXSched to construct the TDDG. ZZXSched [29] uses identity gates to mitigate ZZ crosstalk. These gates are inserted in two cases: to handle parallel single-qubit gate sets and to convert active-qubit interference into cross-qubit interference, reducing qubit interactions. However, identity gates limit program cycle compression. To address this, ZZXSched first schedules gates to minimize ZZ crosstalk, then removes all identity gates. The resulting gate set, free of identity gates, is used as input for CYCO.

4.3 TDDG Creation with Gate Filtering

This section outlines the method for building the TDDG, which involves two key steps: (1) filtering valid successor (or predecessor) gate candidates to identify the nodes (gates) to be connected in the TDDG, and (2) constructing the TDDG by connecting these gates based on distance matrices.

4.3.1 Candidates Filtering for One Gate. The *FilterGateCandidates* function (Algorithm 1) selects valid successors or predecessors for a gate based on spatial proximity in the circuit. It considers gates beyond immediate neighbors to capture key dependencies while preventing interference between selected gates.

Algorithm 1: FilterGateCandidates Function

Input: Gate A to find successors (predecessors), subsequent (preceding) sets of the set containing A, distance matrix D

Output: A list of valid successors (predecessors) finalists for gate A

```
function FilterGateCandidates(A, sets, D):
    gate_candidates, valid_finalists ← [ ]
    for set S_j in sets do
        for gate B in S_j do
            distance ← D[A][B]
            if distance < 2 then
                if gate_candidates ≠ ∅ then
                    tmp_distance ← min({D[tmpgate][B] |
                      tmpgate ∈ gate_candidates})
                    if tmp_distance ≥ 2 then
                        valid_finalists.append(B)
                else
                    valid_finalists.append(B)
                gate_candidates.append(B)
    return valid_finalists
```

A distance matrix defines spatial relationships on NISQ devices, with a unit distance of 1 between qubits. The function processes a current gate A and related gate sets, handling both successors and predecessors. It initializes two empty lists: gate_candidates and valid_finalists. For each gate set S_j, it evaluates gates B, calculating their distance from A using the distance matrix D. If the distance is less than 2, the gates are filtered. If gate_candidates is not empty, the minimum distance between candidates and B is checked; a distance of 2 or more qualifies B as valid. If gate_candidates is empty, B is automatically valid. Valid gates are added to both lists. The function returns valid_finalists, containing valid successors or predecessors for A.

4.3.2 TDDG Creation for all Gates. In the previous paragraph, we discussed the method for selecting successor (or predecessor) candidates. To construct a complete TDDG for a set of gate groups, both nodes and edges should be added to the graph. Algorithm 2 describes the entire procedure for creating a TDDG.

The main procedures are as follows: **(a)** The TDDG is constructed by first initializing an empty DAG. Sepecially, we use AddNode (A, B) function to create DAG, where A means the name of current node and B are successors of A. **(b)** As explained in Section 4.1, a start_node is added to represent the initial layer of the TDDG, connecting it to all gates in the first set of parallel gates. Similarly, an end_node is added to mark the final layer. **(c)** For each subsequent set of gates, S_i, each gate A is added to the graph, and its valid successors are identified using the FilterGateCandidates function. The directed edges are then added to connect the gate A to each of its valid successors, establishing the forward dependencies between the gates. **(d)** After processing all sets, the algorithm reverses the process by iterating from the last set back to the first, this time

Algorithm 2: TDDG Creation

Input: Sets of paralleled gates, distance matrix D
Output: A TDDG with nodes (gates) connected by directed edges

```
TDDG ← InitializeEmptyDAG()
TDDG.AddNode('start_node', 0)
TDDG.AddNode('end_node', 0)
for gate G in the first set do
    TDDG.AddEdge('start_node', G)
for gate G has no successors do
    TDDG.AddEdge(G, 'end_node')
for set S_i in sets do
    for gate A in S_i do
        TDDG.AddNode(A, 0)
        valid_successors ← FilterGateCandidates(A, sets[i + 1:], D)
        for gate VS in valid_successors do
            TDDG.AddEdge(A, VS)
for set S_i from last to first do
    for gate B in S_i do
        valid_predecessors ← FilterGateCandidates(B, layers[:i], D)
        for gate VP in valid_predecessors do
            TDDG.AddEdge(VP, B)
return TDDG
```

focusing on adding the predecessor edges. For each gate B, the algorithm identifies valid predecessors using the FilterGateCandidates function and then adds directed edges from each valid predecessor to gate B, ensuring that backward dependencies are captured.

4.4 Gate Scheduling and Barrier Punching

After the creation of the TDDG, we can schedule gates through quantum cycles. This section presents the gate scheduling component of the CYCO algorithm, which optimizes quantum program cycles by consolidating delayed gates into fewer layers and strategically punching barriers within the circuit.

Initialization. Algorithm 3 takes a TDDG $G = (gate, time)$, a gate duration map π, and topological order L as inputs. L represents TDDG layers. The duration map π, derived from NISQ device calibration, provides gate execution times. The first layer's LMFT starts at zero from the start node and updates to the maximum GFT in that layer.

Iterative Gate Scheduling. The algorithm processes TDDG layers sequentially. It identifies gates from the previous layer ready to execute by calculating their GEST using *FindPredecessors*. Gates with the minimum GEST form the Pre-SZ for immediate execution. These gates are scheduled to avoid conflicts with the LMFT, and barriers ensure layer synchronization. Cross-layer gates not in Pre-SZ are then selected for parallel execution.

Algorithm 3: Gate Scheduling and Barrier Punching

Input: TDDG of a quantum circuit G, layers of the TDDG L, gate latency map $\pi(gate)$
Output: Updated layers of TDDG L, list of barriers for QC sets BS

```
1  barriers, BS ← {}
2  for each pair of consecutive layers (previousLayer, currentLayer) in L do
3      gatesForNewLayer ← Elements from FindPredecessors(currentLayer) with the smallest GEST
4      LMFT ← GEST of gate in gatesForNewLayer
5      Pre-SZ ← gates in previousLayer with GFT < LMFT
6      for gate in Pre-SZ do
7          barriers = barriers ∪ {AddBarrier(gate)}
8      BS = BS ∪ {barriers}
9      crossLayerGate ← { gate | gate ∈ previousLayer ∧ gate ∉ Pre-SZ}
10     PEZ ← gatesForNewLayer + crossLayerGate
11     Insert PEZ into L between previousLayer and currentLayer
12     for gate in gatesForNewLayer do
13         gate.GFT = LMFT ∪ π(gate)
14     Delete all gates from Pre-SZ in TDDG
15 return L, BS
16 Procedure FindPredecessors(currentLayer)
17     earliestGates ← {}
18     for gate in currentLayer do
19         GEST ← max{ predecessor.GFT | predecessor ∈ Predecessors(gate)}
20         earliestGates = earliestGates ∪ {(gate, GEST)}
21     return earliestGates
```

Barrier Punching and Cross-Layer Scheduling. Barriers are "punched" to enable parallel execution by allowing gates to bypass synchronization. Cross-layer gates, which have valid dependencies but do not conflict with Pre-SZ gates, are scheduled earlier in the Parallel Execution Zone (PEZ) alongside current-layer gates. This process is shown in Algorithm 3, lines 6-8.

Updating the TDDG and Topology Order. After scheduling Pre-SZ and cross-layer gates, the algorithm updates the TDDG and L. A new layer combines PEZ and cross-layer gates is added to the layer sequence. GFTs are recalculated, executed gates are removed from the TDDG, and the algorithm proceeds with the remaining gates.

Mitigating ZZ crosstalk. As a final step, CYCO reintroduces identity gates into each set of gates to effectively mitigate ZZ crosstalk. The strategy for applying identity gates follows the same principles as ZZXSched. As shown in Figure 6, two identity gates are applied to transfer active-qubit interference to cross-qubit interference. The red edges indicate cross-qubit interference, while the dashed edges represent active-qubit interference.

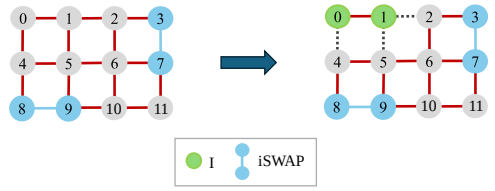


Figure 6: Mitigating ZZ Crostalk by adding identity gates.

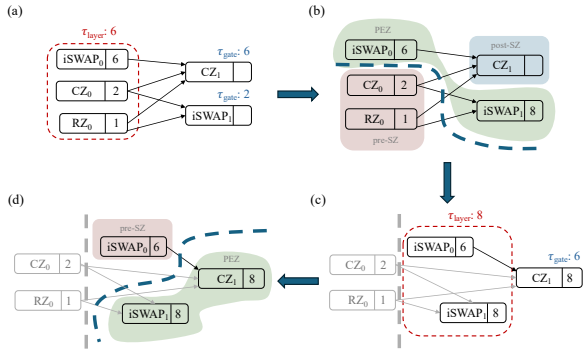


Figure 7: An example of the CYCO algorithm extending from the quantum program in Figure 2 (a).

4.5 An example

To illustrate the CYCO algorithm, we provide an example in Figure 7 that includes the creation of the TDDG and the subsequent gate scheduling process. This example considers a quantum circuit with five gates — $iSWAP_0$, CZ_0, RZ_0, CZ_1, and $iSWAP_1$ — executed on nine qubits ($q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}$, as shown in Figure 2) with latencies of 6, 2, and 1 time units. Each gate is represented as a node in the TDDG, with dependencies as directed edges, such as $iSWAP_0 \rightarrow CZ_1$ and $iSWAP_0 \rightarrow iSWAP_1$, indicating that both gates must follow $iSWAP_0$. Initially, gates are assigned to layers based on dependencies. Layer 1 contains $iSWAP_0$, CZ_0, and RZ_0, which can be executed in parallel, while Layer 2 contains CZ_1 and $iSWAP_1$. The LMFT for the first layer is $\tau_{layer} = 6$, and the GEST for CZ_1 and $iSWAP_1$ in the following layer is $\tau_{gate} = 6$ and $\tau_{gate} = 2$, respectively, based on their predecessors' finish times. The cross-layer gate is $iSWAP_0$, as $iSWAP_1$ is advanceable. Both SWAP gates are placed in PEZ, updating the finish time of the second SWAP gate to 8. In the next iteration, PEZ = {$iSWAP_0$, $iSWAP_1$} will be used to determine the next cross-layer gates. The next identified cross-layer gate is $iSWAP_1$.

5 Evaluation

We evaluated CYCO using benchmarks on the latest quantum hardware, focusing on both simulations and real-device experiments. The evaluation compares CYCO's effectiveness with existing ZZ crosstalk mitigation methods, using program cycle reduction, gate parallelism and fidelity as key metrics. In this work, our aim is to address the following research questions.

- [*RQ*1]: How does CYCO improve parallelism compared to existing ZZ crosstalk mitigation methods?

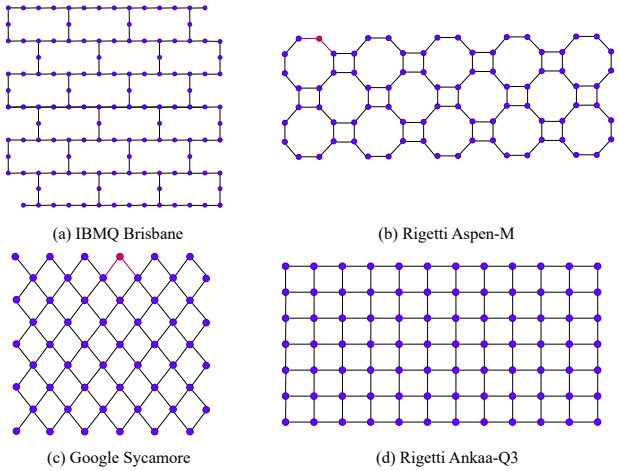


Figure 8: Coupling graphs for the four quantum hardware platforms used in our evaluation. Red nodes represent unusable physical qubits in practice.

- [*RQ*2]: What is the impact of the quantum topologies on the different algorithms to solve our problem?
- [*RQ*3]: What is the reliability of the different methods under the real-device condition?
- [*RQ*4]: How does CYCO scale with circuit size?

Baseline Comparison. To ensure a fair evaluation of CYCO, we compare it with standard quantum execution and ZZXSched, an advanced framework for co-optimizing gate scheduling and pulse control to mitigate ZZ crosstalk [29]. ZZXSched includes gate scheduling at the software level and pulse optimization at the hardware level. However, to ensure a fair comparison that isolates the software-based scheduling improvements introduced by CYCO, we exclude ZZXSched's pulse optimization component in our experiments. This enables us to directly evaluate the improvements in execution time and parallelism derived purely from CYCO's scheduling optimizations.

Benchmark Selection. To evaluate CYCO's performance, we use 72 benchmarks from the QASMBench suite [16], a widely recognized collection designed to assess NISQ devices. The benchmarks cover various quantum algorithms, categorized by size: small-scale (2–10 qubits, 11–1008 gates, depths of 2–551), medium-scale (11–27 qubits, 22–2016 gates, depths of 10–2987), and large-scale (28–76 qubits, 40–959 gates, depths of 32–9265).

Compiler and Implementation Details. We implemented our CYCO scheduling algorithm using Python 3.9, interfacing with the IBM Qiskit software library [2]. The Qiskit transpiler was utilized to compile the logical quantum circuits from the QASMBench benchmarks into executable forms on actual quantum hardware. To ensure that CYCO's contribution to scheduling optimization is isolated, we set the Qiskit optimization level to 0, disabling any other transpiler-level optimizations that could interfere with the results of our scheduling algorithm. The compilation process primarily uses the SABRE mapping algorithm [17], which is widely

Table 2: Gate Specifications Across Quantum Hardware Platforms.

Machine Name	Single-Qubit Gate	Duration (ns)	Two-Qubit Gate	Duration (ns)
IBMQ Brisbane	id, rz, sx, x	0–60	ECR	660
Google Sycamore	Phased XZ, Virtual Z, Physical Z	0–25	Sycamore, $\sqrt{\text{iSWAP}}$, CZ	12–32
Rigetti Aspen-M	RX, RZ	60	iSWAP, CZ	160
Rigetti Ankaa-Q3	RX, RZ	60	iSWAP, CZ	160

used to map logical qubits to physical ones on a quantum processor. For evaluating performance across different quantum hardware platforms, we adapted the coupling maps and physical basis gates to suit each machine's architecture.

Architectural Features of Quantum Hardware. We conducted our evaluations on four quantum devices including IBM's 127-qubit processor from the Eagle family [10], Google's Sycamore chip [3], and Rigetti's Aspen-M and Ankaa-Q3 devices [25].

1) Settings for Simulations: For the simulation experiments, we employed hardware-specific coupling graphs and gate duration data. The coupling graphs for these platforms are shown in Figure 8, and the corresponding gate duration data is summarized in Table 2. As observed across all platforms, two-qubit gates such as CZ or iSWAP have significantly longer durations compared to single-qubit gates. Accurate modeling of gate latencies ensures that our simulations closely reflect real-world hardware performance. For example, on IBM's Eagle processors, the ECR gate has a duration of $660ns$, whereas single-qubit gates like rz and sx are almost instantaneous ($0 - 60ns$).

2) Settings for Real-Device Experiments: We evaluated CYCO on IBMQ-Brisbane, a 127-qubit superconducting quantum processor for real condition tests. The topology of IBMQ-Brisbane offers a balanced qubit connectivity that helps reduce crosstalk while maintaining high coherence times [11]. Its architecture makes it ideal for testing large-scale quantum circuits.

5.1 Evaluation Metrics

We evaluated CYCO by three key metrics: speedup ratio, gate parallelism and fidelity. The calculation method is detailed as follows.

Speedup Ratio. This measures CYCO's performance gains:

$$\Delta = \frac{\tau_{ZZXSched} - \tau_{CYCO}}{\tau_{ZZXSched}} \tag{4}$$

where Δ is the speedup ratio, τ_{CYCO} and $\tau_{ZZXSched}$ are the total program cycles using CYCO and ZZXSched, respectively, computed as:

$$\tau_{CYCO(ZZXSched)} = \sum_{n=0}^{layers} t_{layer\_cycle}. \tag{5}$$

A higher Δ indicates better efficiency from CYCO's optimized gate scheduling [24].

Fidelity. We use Hellinger fidelity to measure state similarity:

$$F = (1 - H^2)^2 \tag{6}$$

where F is fidelity, and H is the Hellinger distance between ideal and actual states. F ranges from 0 to 1, with 1 being perfect fidelity [8].

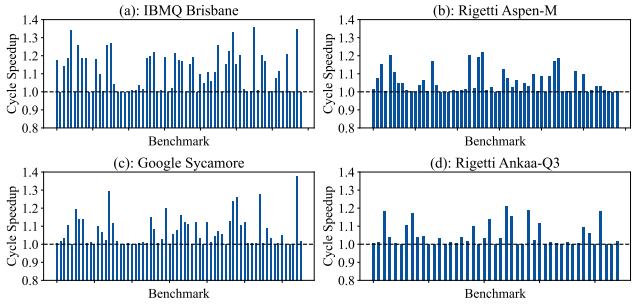


Figure 9: Speedup ratio simulation of four devices.

Gate Parallelism. We evaluate the improvement in parallel gate execution between CYCO and ZZXSched. This metric compares the average number of gates executed simultaneously per cycle, with a higher value indicating CYCO's superior concurrency optimization.

6 Results

6.1 RQ1 — Parallelism and Efficiency

The primary goal of the CYCO algorithm is to reduce the execution time of quantum circuits. Our experiments demonstrate that CYCO substantially outperforms the current state-of-the-art algorithm, ZZXSched, across multiple quantum topologies including IBM, Google, and Rigetti systems.

6.1.1 IBMQ Devices. Figure 9 (a) illustrates the speedup achieved by CYCO on IBM's 127-qubit Brisbane processor. On average, CYCO reduced the circuit execution time by 14.19% across the set of benchmarks, with the *dnn_n16* benchmark showing the most significant improvement, achieving a speedup of 35.85%. This result highlights the efficiency of CYCO in handling larger circuits, where qubit interactions and gate scheduling become more complex.

6.1.2 Google Sycamore Device. On Google's 53-qubit Sycamore processor (Figure 9 (b)), CYCO achieved an average execution time reduction of 6.02%. Although the speedup on Sycamore was less pronounced than on IBMQ devices, the CYCO's performance is still notable, with a maximum speedup of 22.02% observed in certain benchmarks. This difference in performance can be partly attributed to Sycamore's native gate set, which includes faster gate operations such as the $\sqrt{\text{iSWAP}}$ and CZ gates. As a result, the relative gains from optimizing gate scheduling are smaller compared to devices where gate latencies are more varied.

6.1.3 Rigetti Aspen-M and Ankaa-Q3 Devices. Rigetti's Aspen-M and Ankaa-Q3 devices also showed positive results with CYCO (Figures 9 (c) and 9 (d)). CYCO achieved an average speedup of

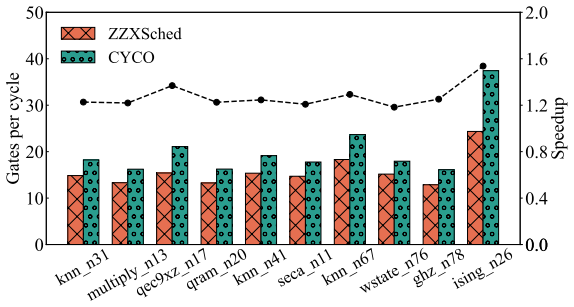


Figure 10: Parallelism comparison on IBMQ-Brisbane.

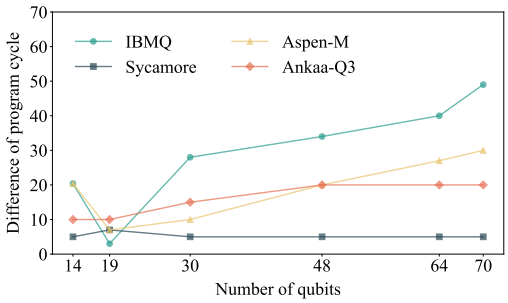


Figure 12: Scalability comparison across different architectures.

6.27% on Aspen-M and 4.25% on Ankaa-Q3, with the best case on Aspen-M reaching a remarkable 37.44% improvement in execution time. This shows that the CYCO algorithm is effective in various quantum topologies.

6.1.4 Parallelism Analysis. As evidenced by the cycle-resolved measurements in Figure 10 (Partial results on IBMQ-Brisbane), CYCO achieves up to 1.7× higher gates per cycle than ZZXSched across benchmark circuits. This cycle-level parallelism advantage stems from CYCO's dynamic dependency resolution, which packs compatible gates into fewer computational cycles. By maximizing gate density per cycle, CYCO reduces total execution cycles by 14% on average compared to ZZXSched. Notably, circuits with clustered multi-qubit operations (e.g., ghz-n78) exhibit pronounced improvements, confirming that reduced idle cycles through optimized scheduling drive the acceleration mechanism.

6.2 RQ2 — Qubit Connectivity

Interestingly, our results suggest that CYCO performs particularly well in environments with low-connectivity qubit architectures. Low connectivity structures, such as those found in IBMQ-Brisbane and Rigetti's Aspen-M, tend to benefit more from intelligent gate scheduling, since physical qubit interactions are constrained by the hardware topology. CYCO's ability to optimize scheduling in such cases leads to better parallelism and resource utilization, as demonstrated by the greater speedups observed on these devices.

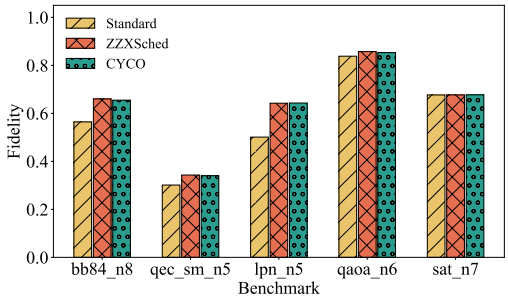


Figure 11: Fidelity results on real IBMQ-Brisbane device.

In contrast, Linear Nearest Neighbor architectures [9], which inherently limit gate concurrency, show less dramatic improvements in execution time, though CYCO still provides notable gains.

6.3 RQ3 — Fidelity Maintenance

Figure 11 presents the fidelity results from running six representative benchmarks on the IBMQ Brisbane device. While there is a slight reduction in fidelity compared to ZZXSched due to the increased circuit density, the overall fidelity remains high. Across all benchmarks, the fidelity degradation exceeded less than 5%, and in many cases, the difference can be accepted. This is a key result, as it indicates that CAZZO can significantly reduce execution time without introducing substantial errors in quantum computing.

6.3.1 Trade-offs Between Density and Fidelity. Our analysis suggests a subtle trade-off between circuit density and fidelity. As CYCO increases the quantum circuit density by scheduling gates more compactly in time, it reduces the overall execution time, which benefits fidelity due to the decreased exposure to decoherence. However, the denser packing of gates also increases the likelihood of crosstalk and other noise sources. Key takeaway from our experiments is that CYCO strikes a favorable balance, achieving significant reductions in execution time while maintaining high fidelity.

6.4 RQ4 — Computational Scalability

Figure 12 presents a scalability comparison of CYCO versus ZZXSched on four different quantum devices. The Y-axis represents the difference in program cycles between ZZXSched and CYCO, while the X-axis represents the scale of the benchmark. As circuit size increases, CYCO demonstrates a consistently lower program cycle. This trend indicates better computational scalability, suggesting that CYCO performs gate scheduling in parallel more effectively. Besides, CYCO is architecture independent and can be effectively implemented on systems with distinct topologies.

7 Related Work

ZZ Crosstalk Suppression. Heterogeneous qubits [5, 15, 30], tunable couplers [18, 23, 28], and multiple coupling paths [12, 20] have been proposed to mitigate ZZ crosstalk by hardware solutions. However, these methods may increase decoherence and add fabrication complexity [12, 19]. Our work builds on the pulse and scheduling co-optimization called ZZXSched [29], a software-based method

that avoids specialized hardware and is applicable across various devices. We specifically focus on optimizing gate scheduling to reduce execution time rather than improving pulse optimization techniques from prior work.

Duration-Aware Gate Scheduling. Our work is inspired by [4], which addresses the qubit mapping problem by considering gate duration differences and the impact of program context. They propose a duration-aware remapping algorithm that leverages gate duration variations and program context to extract more parallelism, achieving an average speedup of 1.23× while maintaining circuit fidelity. This work highlights the importance of gate duration in qubit mapping, an aspect often overlooked in previous solutions that assume uniform gate durations.

To our knowledge, no previous work has proposed using quantum cycles to schedule quantum gates in quantum error mitigation. This work first conquered the time compression problem in the systemic mitigation of ZZ crosstalk.

8 Conclusions

We formally defined the cycle-aware ZZ crosstalk mitigation problem and proposed CYCO, an optimization framework that dynamically balances ZZ crosstalk suppression with gate parallelism by maximizing time-resource utilization. Using TDDG, CYCO adaptively schedules gates with varying durations while resolving spatiotemporal dependencies. Evaluations with QASMBench benchmarks on four different superconducting quantum devices demonstrate CYCO achieves efficient performance: it gains up to 37.44% program cycle reduction (average 14.19%) across 53- to 127-qubit systems, primarily by increasing gate parallelism by 1.6× compared to the ZZXSched scheduler. Real-device validations further confirm CYCO's fidelity-preserving acceleration. Overall, our algorithm has achieved significant progress in optimizing quantum circuit execution efficiency and in the area of mitigating ZZ crosstalk. Future work could focus on extending CYCO's capabilities to incorporate additional error mitigation techniques.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2023YFA1009403, the National Natural Science Foundation of China under Grant No. 62472175, the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant No. 22510750100, and the Shanghai Frontiers Science Center of Molecule Intelligent Syntheses.

References

[1] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681.
[2] Gadi et. al. Aleksandrowicz. 2019. Qiskit: An Open-source Framework for Quantum Computing. *Zenodo* (2019). doi:10.5281/zenodo.2562110
[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[4] Haowei Deng, Yu Zhang, and Quanxi Li. 2020. Codar: A contextual duration-aware qubit mapping for various nisq devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[5] Noguchi et. al. 2020. Fast parametric two-qubit gates with suppressed residual interaction using the second-order nonlinearity of a cubic transmon. *Physical Review A* 102, 6 (2020), 062408.
[6] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing(STOC)* (1996), 212–219.
[7] G Giacomo Guerreschi and Jongsoo Park. 2017. Gate scheduling for quantum algorithms. *arxiv.1708.00023v1* (2017).
[8] Ernst Hellinger. 1909. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik* 1909, 136 (1909), 210–271.
[9] Wei Hu, Yang Yang, Weiye Xia, Jiawei Pi, Enyi Huang, Xin-Ding Zhang, and Hua Xu. 2022. Performance of superconducting quantum computing chips under different architecture designs. *Quantum Information Processing* 21, 7 (2022), 237.
[10] IBM. 2021. IBM Quantum. https://quantum-computing.ibm.com/.
[11] Petar Jurcevic, Ali Javadi-Abhari, Lev S Bishop, Isaac Lauer, Daniela F Bogorin, Markus Brink, Lauren Capelluto, Oktay Günlük, Toshinari Itoko, Naoki Kanazawa, et al. 2021. Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology* 6, 2 (2021), 025020.
[12] Abhinav Kandala, Ken X Wei, Srikanth Srinivasan, Easwar Magesan, S Carnevale, GA Keefe, D Klaus, O Dial, and DC McKay. 2021. Demonstration of a high-fidelity cnot gate for fixed-frequency transmons with engineered zz suppression. *Physical Review Letters* 127, 13 (2021), 130501.
[13] Youngseok et. al. Kim. 2023. Evidence for the utility of quantum computing before fault tolerance. *Nature* 618, 7965 (2023), 500–505.
[14] Philip Krantz, Morten Kjaergaard, Fei Yan, Terry P Orlando, Simon Gustavsson, and William D Oliver. 2019. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews* 6, 2 (2019), 021318.
[15] Jaseung Ku, Xuexin Xu, Markus Brink, David C McKay, Jared B Hertzberg, Mohammad H Ansari, and BLT Plourde. 2020. Suppression of unwanted ZZ interactions in a hybrid two-qubit system. *Physical review letters* 125, 20 (2020), 200504.
[16] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing* 4, 2 (2023), 1–26.
[17] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems(ASPLOS'19)*. 1001–1014.
[18] X Li, T Cai, H Yan, Z Wang, X Pan, Y Ma, W Cai, J Han, Z Hua, X Han, et al. 2020. Tunable coupler for realizing a controlled-phase gate with dynamically decoupled regime in a superconducting circuit. *Physical Review Applied* 14, 2 (2020), 024070.
[19] Moein Malekakhlagh, Easwar Magesan, and David C McKay. 2020. First-principles analysis of cross-resonance gate operation. *Physical Review A* 102, 4 (2020), 042605.
[20] Pranav Mundada, Gengyan Zhang, Thomas Hazard, and Andrew Houck. 2019. Suppression of qubit crosstalk in a tunable coupling superconducting circuit. *Physical Review Applied* 12, 5 (2019), 054023.
[21] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems(ASPLOS'19)*. 1015–1029.
[22] Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'20)*. 1001–1016.
[23] AO Niskanen, Khalil Harrabi, F Yoshihara, Y Nakamura, S Lloyd, and Jaw Shen Tsai. 2007. Quantum coherent tunable coupling of superconducting qubits. *Science* 316, 5825 (2007), 723–726.
[24] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.
[25] Rigetti. 2023. Rigetti Quantum Cloud Services. https://qcs.rigetti.com/qpus.
[26] Maximilian Schlosshauer. 2019. Quantum decoherence. *Physics Reports* 831 (2019), 1–57.
[27] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *Society for Industrial and Applied Mathematics(SIAM review)* 41, 2 (1999), 303–332.
[28] Youngkyu Sung, Leon Ding, Jochen Braumüller, Antti Vepsäläinen, Bharath Kannan, Morten Kjaergaard, Ami Greene, Gabriel O Samach, Chris McNally, David Kim, et al. 2021. Realization of high-fidelity CZ and ZZ-free iSWAP gates with a tunable coupler. *Physical Review X* 11, 2 (2021), 021058.
[29] Lei Xie, Jidong Zhai, ZhenXing Zhang, Jonathan Allcock, Shengyu Zhang, and Yi-Cong Zheng. 2022. Suppressing zz crosstalk of quantum computers through pulse and scheduling co-optimization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'22)*. 499–513.
[30] Peng Zhao, Peng Xu, Dong Lan, Ji Chu, Xinsheng Tan, Haifeng Yu, and Yang Yu. 2020. High-contrast zz interaction using superconducting qubits with opposite-sign anharmonicity. *Physical review letters* 125, 20 (2020), 200503.