Published by
NIGERIAN SOCIETY OF PHYSICAL SCIENCES
Available online @ https://journal.nsps.org.ng/index.php/jnsps

J. Nig. Soc. Phys. Sci. 7 (2025) 2615

Journal of the Nigerian Society of Physical Sciences

Compiler-assisted code generation for quantum computing: leveraging the unique properties of quantum architectures

G. G. James [a,*], A. P. Ekong[b], A. U. Unyime[c], A. Akpanobong[d], J. A. Odey[e], D. O. Egete[e], S. Inyang[a], I. Ohaeri[a], C. M. Orazulume[f], A. Etuk[g], P. Okafor[h]

[a]*Department of Computing, Topfaith University, Mkpatak, Nigeria*
[b]*Department of Computer Science, Akwa Ibom State University, Ikot Akpaden, Nigeria*
[c]*Department of Computer and Robotic Education, University of Uyo, Nigeria*
[d]*Department of Computer Networks, Faculty of Science and Information Technology, UPM, Serdang, Malaysia*
[e]*Department of Computer Science, University of Calabar, Calabar, Nigeria*
[f]*Department of Electrical Electronics Engineering, Topfaith University, Mkpatak, Nigeria*
[g]*Department of Computer Science, Michael Okpara University of Agriculture, Umudike, Nigeria*
[h]*Department of State Service, Ebonyi State Command, Abakaliki, Nigeria*

Abstract

Quantum computing holds transformative potential, but its adoption is hindered by the complexity of generating efficient, hardware-specific code. This work presents a modular, extensible compiler framework that bridges high-level quantum languages with diverse hardware architectures. The framework consists of three modules: a front-end for parsing quantum code into a hardware-agnostic intermediate representation (IR), an optimization module for enhancing quantum circuits through gate synthesis, qubit routing, and error mitigation, and a back-end for generating hardware-specific instructions. Major contributions include a hardware-agnostic IR for cross-platform compatibility, optimization techniques to reduce gate complexity and noise, and hardware-specific adaptations to improve execution fidelity. A practical demonstration optimizes quantum circuits, highlighting the impact of hardware constraints. Comparative analysis of IBM Quantum and IonQ platforms underscores the role of qubit connectivity and noise resilience in algorithmic performance. This scalable framework enhances quantum software development and efficient hardware utilization.

DOI:10.46481/jnsps.2025.2615

Keywords: Neuroprotection, Quantum computing, Quantum architectures, Quantum circuits design, Quantum code optimization

Article History :
Received: 04 January 2025
Received in revised form: 12 February 2025
Accepted for publication: 14 March 2025
Published: 04 April 2025

Communicated by: Oluwatobi Akande

1. Introduction

Quantum computing operates on principles vastly different from classical computing, with qubits exploiting quantum phenomena such as superposition and entanglement. As the field advances, there is a growing demand for efficient compilation techniques that can abstract the complexities of quantum hardware and enable programmers to focus on high-level algorithm design. This literature review examines key developments in compiler techniques for quantum computing and explores how

*Corresponding author Tel. No: +234-810-738-1867.
Email address: g.james@topfaith.edu.ng (G. G. James)

these approaches optimize code generation to align with the specific features of quantum architectures.

The need for specialized compilers in quantum computing arises from the fundamental differences between classical and quantum computing models. Classical computers operate using bits that represent binary states (0 or 1), while quantum computers utilize qubits, which can exist in a superposition of both states. Additionally, quantum gates, the operations performed on qubits, differ significantly from classical logic gates in terms of behavior and resource constraints [1]. Given these differences, compilers for quantum computers must not only translate high-level code into low-level instructions but also account for the quantum mechanical properties that govern quantum operations [2].

Chong *et al.* [3] emphasized the co-design of quantum algorithms and architectures, focusing on the intersection of compiler design and quantum hardware. Carbin et al. explored quantum programming languages and quantum circuit optimization, focusing on tools for automated verification and compiler-level quantum circuit generation [4]. Ross and Selinger worked on quantum gate synthesis and compilation, creating optimized quantum circuits by leveraging the properties of quantum gates [5]. Cross, an architect of the Qiskit platform, focused on quantum compiler design to enable efficient quantum computation on IBM's quantum processors [6]. Schuch examined quantum architectures and circuits, especially focusing on quantum information theory and tensor networks that inform compiler design [7].

Javadi-Abhari *et al.* [8] focused on quantum circuit optimization and compilation, working closely with Qiskit on compiler optimizations for quantum hardware. Sheldon *et al.* [9] explored quantum error mitigation and optimization, which are critical for quantum compilers dealing with real hardware constraints. Fowler *et al.* [10] research in quantum error correction, particularly surface codes, has had significant implications for compiler design, focusing on how quantum code can be optimized for fault-tolerant quantum architectures.

Compiler-assisted code generation for quantum computing is a burgeoning area aimed at optimizing and automating the process of writing code for quantum processors, which differ significantly from classical architectures [11]. Quantum computing introduces unique challenges due to its probabilistic nature, the superposition of states, and entanglement. Compiler techniques in this domain focus on leveraging these properties while optimizing performance and reducing errors [12].

Quantum computing, a revolutionary paradigm in computation, promises unprecedented capabilities for solving complex problems beyond the reach of classical systems [13]. However, the unique principles of quantum mechanics that underpin quantum computing—such as superposition, entanglement, and interference—introduce significant challenges in programming and code generation [14]. Unlike classical computing, where well-established compiler frameworks efficiently translate high-level code into machine instructions, quantum computing requires novel approaches to address its fundamentally different architectural and operational constraints [15].

One major challenge in generating efficient code for quantum computing lies in the limitations of existing compiler frameworks [16]. Classical compilers, such as GCC and LLVM, have evolved over decades to optimize code for deterministic, binary architectures [17]. While these frameworks offer valuable insights, they fall short when applied to quantum systems due to the probabilistic nature of quantum operations and the physical constraints of qubits [18]. Recent efforts to develop quantum-specific compilers, such as t|ket>, Quilc, and Qiskit's transpiler, have laid the groundwork for quantum code generation [19–21]. However, these tools often lack robust optimization strategies for diverse hardware architectures, highlighting the need for a more comprehensive and adaptive approach [22].

A key factor complicating quantum code generation is the unique requirements of quantum architectures [23]. Qubits, the fundamental units of quantum information, exhibit hardware-dependent constraints, including limited connectivity, susceptibility to noise, and variations in gate fidelity [24]. For instance, superconducting qubits may have fixed couplings that restrict two-qubit gate operations to specific pairs, necessitating qubit routing strategies [25]. Additionally, the finite coherence times of qubits impose strict limitations on circuit depth, while error rates vary significantly across devices [26]. These factors demand compilers that can perform sophisticated hardware-aware optimizations to ensure efficient and reliable quantum program execution [27].

Furthermore, quantum programming languages and paradigms present their own set of challenges and opportunities [28]. Languages such as Qiskit, Cirq, and Q# provide abstractions to design and simulate quantum circuits, yet they differ significantly in syntax, capabilities, and target backends [29]. For example, Qiskit emphasizes modularity and interoperability with IBM Quantum hardware, while Cirq is tailored for Google's quantum processors and hybrid workflows [30]. Meanwhile, Q# focuses on a high-level functional programming approach, facilitating algorithm development [31]. These languages underscore the need for a compiler framework that bridges the gap between diverse programming paradigms and heterogeneous quantum architectures, enabling seamless translation and optimization across platforms [32].

In this context, this study explores a compiler-assisted code generation approach tailored to quantum computing. By leveraging insights from classical compiler design and addressing the specific challenges posed by quantum architectures, this approach aims to optimize quantum program execution while enhancing portability and scalability [33]. This work reviews existing frameworks, examines architectural requirements, and analyzes quantum programming paradigms to propose a novel methodology for efficient and adaptive quantum code generation [34].

2. Key techniques and concepts of compiler quantum computing

2.1. Quantum intermediate representations (QIR)

Quantum Intermediate Representation (QIR) serves as an abstraction layer between quantum algorithms and the under-

lying quantum hardware. It allows for the separation of algorithmic descriptions from hardware-specific constraints. The QIR facilitates optimizations at various stages of the compilation process. Tools such as Microsoft's QIR Alliance aim to standardize intermediate representations to ensure compatibility across different quantum hardware backends [35]. The use of QIR improves the portability of quantum programs and enables advanced compiler optimizations.

2.2. Quantum circuit optimization

Quantum circuit optimization is a critical step in the compilation process. Techniques such as gate cancellation, commutation analysis, and template matching have been developed to minimize the number of quantum gates and reduce the circuit depth [36]. These optimizations are essential for improving the performance of quantum algorithms, especially given the limited coherence times of qubits in current quantum hardware. Optimization techniques for quantum circuits are critical for enhancing the performance of quantum algorithms on real quantum hardware. Quantum circuit optimization is also a multi-faceted problem, particularly due to the constraints of current quantum hardware, such as noise, limited qubits, and gate fidelity. Compiler-level optimizations can make a significant difference in reducing the resources required for quantum computations. Quantum circuit optimization techniques implemented at the compiler level are essential for improving the performance of quantum algorithms, particularly on noisy intermediate-scale quantum (NISQ) devices [37–42]. Gate minimization, qubit overhead reduction, hardware-aware optimization, gate-level parallelism, and error mitigation are key strategies that can be deployed to extend the capabilities of current quantum hardware. As quantum compilers evolve, integrating these techniques will be crucial in making quantum computing more practical and efficient. Optimization techniques specific to quantum circuits that can be implemented through compilers, as ascertained by Amy *et al.* [36] include:

2.2.1. Gate minimization

Gate minimization is crucial because the error rates of quantum operations increase with the number of gates. The goal is to reduce the overall number of quantum gates or replace costly multi-qubit gates with simpler, less error-prone alternatives [36]. Gate minimization techniques include:

1. Gate Fusion: The compiler can identify sequences of gates that, when combined, lead to an equivalent but simpler operation. For example, consecutive unitary operations might be merged into a single unitary matrix. This is often done with single-qubit gates and controlled-NOT (CNOT) gates [43].
2. Template Matching: Known gate patterns that can be simplified are matched and replaced with their optimized counterparts. This technique uses a library of predefined "templates" to reduce gates. Decomposition Optimization: Complex gates like Toffoli or other multi-controlled gates can be decomposed into sequences of more hardware-efficient gates like CNOTs and single-qubit rotations. Optimal decompositions can lower the number of operations, improving fidelity and speed [44].

2.3. Qubit Overhead Reduction

Reducing qubit overhead focuses on minimizing the number of qubits used in a quantum algorithm, which is crucial given the limited qubit availability on current hardware [45]. Qubit overhead reduction has unique techniques, which are:

1. Qubit Reuse: By careful scheduling and analysis of qubit lifetimes, the compiler can reuse qubits for different parts of the computation. This is particularly effective when parts of the computation are independent or sequential [44].
2. Ancilla Qubit Reduction: Many quantum algorithms, such as those involving arithmetic or error correction, require ancillary qubits (ancillas) for intermediate calculations. Compilers can optimize the use of ancilla qubits, ensuring they are freed and reused as soon as possible [41, 43].
3. Quantum Memory Management: Similar to classical memory management, quantum memory management techniques allow qubits to be released and reassigned dynamically during computation. This helps reduce the number of qubits required to implement a circuit [43].

2.4. Hardware-aware optimization

Quantum circuits must be tailored to the specific architecture of the quantum hardware for optimal performance. Different quantum platforms (for example, superconducting qubits and trapped ions) have unique physical constraints, such as connectivity between qubits, gate fidelities, and error rates [46]. Techniques for hardware–aware optimization include:

1. Qubit Mapping and Routing: On quantum devices, not all qubits can interact directly. Therefore, compilers must map logical qubits in the algorithm to physical qubits on the hardware, optimizing for connectivity. Techniques like SWAP insertion allow qubits to be moved around the hardware topology efficiently.
2. Noise-Aware Gate Scheduling: Given the variation in gate fidelities and noise profiles between qubits, the compiler can schedule critical operations on the most reliable qubits and use gates with the highest fidelity [47].
3. Calibration-Aware Gate Sequences: Different quantum hardware may favor certain gate implementations. For instance, on superconducting quantum computers, certain two-qubit gates (like CZ or iSWAP) may be more reliable than others. Compilers can tailor the gate sequence to the hardware's strengths.

2.5. Quantum architecture analysis

Studying qubit configurations, understanding hardware constraints, and developing robust mapping strategies make it possible to optimize quantum code for execution on diverse architectures. These insights serve as a foundation for designing a compiler that adapts to the unique properties of quantum systems, enhancing performance and reliability across platforms.

2.6. Physical and logical qubit configurations

Quantum architectures vary significantly in their implementation of qubits and their associated control mechanisms. Below is an overview of notable architectures:

1. IBM Quantum (Superconducting Qubits): IBM Quantum uses superconducting qubits arranged in a fixed grid topology. These qubits are controlled via microwave pulses, and their connectivity is limited to nearest neighbors, as defined by the coupling map. Logical qubits are implemented through quantum error correction, requiring multiple physical qubits to form a single logical qubit.
2. Rigetti Computing (Superconducting Qubits): Similar to IBM, Rigetti employs superconducting qubits with a focus on scalable manufacturing. Rigetti architectures typically feature a lattice topology with limited connectivity and are designed for high-speed gate operations.
3. IonQ (Trapped-Ion Qubits): IonQ leverages trapped-ion technology, where ions are manipulated using laser pulses. Unlike superconducting qubits, trapped-ion systems offer full connectivity between qubits, enabling direct implementation of multi-qubit gates without additional routing. Logical qubits in IonQ systems also rely on quantum error correction schemes.

2.7. Hardware-specific constraints

Quantum hardware imposes several constraints that significantly impact the design and execution of quantum programs:

1. Gate Sets: Each quantum architecture supports a specific set of native gates, which are directly implemented by the hardware. For example:
   (a) IBM Quantum: CX (CNOT), U1, U2, U3 gates.
   (b) Rigetti: Parametric RX, RZ rotations, and controlled gates.
   (c) IonQ: Arbitrary single-qubit rotations and Mølmer–Sørensen gates for entanglement.

   Compilers must decompose higher-level gates into these native gates to execute quantum circuits effectively.
2. Decoherence Times: Quantum systems are highly susceptible to noise, with decoherence times varying across architectures. Superconducting qubits typically exhibit coherence times in the range of microseconds, while trapped-ion systems offer longer coherence times (milliseconds). This impacts the permissible circuit depth and necessitates error mitigation techniques.
3. Qubit Topology: The connectivity between qubits determines how multi-qubit gates can be implemented.
4. Superconducting qubits (IBM, Rigetti): Restricted connectivity requiring qubit routing.

3. Methodology

The methodology for designing a compiler-assisted code generation framework for quantum computing involves a systematic approach addressing the challenges of quantum-specific requirements, hardware constraints, and optimization techniques. The process begins with identifying the unique challenges in quantum computing, such as qubit connectivity, gate fidelity, and error rates, by reviewing existing compiler frameworks and quantum programming paradigms (e.g., Qiskit, Cirq, Q#). A detailed analysis of quantum architectures, including physical and logical qubit configurations, gate sets, decoherence times, and qubit topology, helps establish a mapping between high-level operations and hardware-level instructions.

The framework's design includes a modular architecture with front-end, optimization, and back-end modules. Quantum-specific optimizations, such as gate synthesis, qubit routing, and error mitigation, are incorporated to improve performance and reduce errors [48, 49]. A hardware-agnostic intermediate representation (IR) ensures compatibility with multiple quantum backends [50–52]. The process also integrates static and dynamic analysis techniques for resource estimation and runtime optimization alongside circuit optimization strategies like gate cancellation, commutation, and hardware-specific enhancements [53]. Validation through benchmark programs, simulation, and hardware testing ensures correctness and efficiency. Iterative evaluation, user feedback, and comprehensive documentation further refine the framework, supporting its scalability, portability, and adoption within the quantum computing community [54].

3.1. Mapping high-level quantum operations to hardware-level instructions

A thorough review of existing literature on technology stress, machine learning methodologies, and translating abstract quantum algorithms into instructions compatible with target hardware involves several stages, each tailored to the unique characteristics of the quantum architecture. Below is a step-by-step illustration using an example algorithm: the Quantum Fourier Transform (QFT), commonly used in quantum computing applications like phase estimation.

The QFT on nnn-qubits is defined by the unitary transformation:

$$|j\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i.jk/2^n} |k\rangle. \tag{1}$$

The circuit comprises:

1. Hadamard gates to create superpositions.
2. Controlled phase gates to entangle qubits and apply phase shifts.
3. Swap gates to reverse the order of qubits for correct output

3.1.1. Intermediate representation

In the Intermediate Representation step, the algorithm is represented as a quantum circuit in terms of standard gates: In this step, the algorithm is represented as a quantum circuit in terms of standard gates in terms of Hadamard gate (H) on each qubit, Controlled R_k gates (R_k = diag(1, $e^{2\pi i/2^k}$)) between qubits, and SWAP gates to reorder qubits.

For a 3-qubit QFT, the circuit could be produced as follows:
Apply H on q_0
Apply R_2 between q_0 and q_1, R_3 between q_0 and q_2
Apply H on q_1, R_2 between q_1 and q_2
Apply H on q_2
SWAP q_0 and q_2

3.1.2. Gate decomposition

To make the circuit compatible with hardware, each gate is decomposed into native gate operations. For example:

Hadamard Gate (H): Decomposed as:

$$H = R_Z(\pi)R_X(\frac{\pi}{2}), \tag{2}$$

implementing single-qubit rotations in hardware.

Controlled R_k Gate: Decomposed into:

$$R_k = \begin{cases} CX\, R_Z(\theta_k)\, CX, & for\ IBM/Regetti. \\ MS\ (\theta_k), & for\ IonQ\ (Molmer - Sorensen\ gate) \end{cases}$$

SWAP Gate for hardware restricted connectivity: the SWAP gate is implemented using three CX gates:

$$SWAP = CX_{12}CX_{21}CX_{12} \tag{3}$$

Given the physical qubit topology of the target hardware, in the IBM Quantum, a linear topology may require inserting additional SWAP gates to connect non-adjacent qubits. Whilst, in the IonQ, no additional routing is needed due to full connectivity.

For a 3-qubit QFT on IBM's 5-qubit backend, logical qubits q_0, q_1, and q_2 might map to physical qubits Q_0, Q_2, and Q_4, requiring routing between Q_0 and Q_4.

After decomposition and routing, the algorithm is translated into hardware instructions:

- IBM QASM (Quantum Assembly):

  Perl

  q[0] -> Q_0; q[1] -> Q_2; q[2] -> Q_4;

  h Q_0;

  cx Q_0, Q_2;

  rz(pi/2) Q_2;

  cx Q_0, Q_2;

  ...

  swap Q_0, Q_4;

- IonQ API Call (Trapped-Ion):

  lua

  add_gate ("MS", qubits = [0, 1], theta = pi/4);

  add_gate("RX", qubit = 0, theta = pi/2);

  ..

However, the output of the 3-qubit Quantum Fourier Transform (QFT) algorithm depends on the initial state of the quantum system. Let's assume the system starts in the computational basis state $|1\rangle = |001\rangle$.

After executing the QASM code on IBM hardware:

Table 1. Measurement probabilities for each basis state in the 3-qubit QFT algorithm.

Basis State	Probability
(	000⟩)
(	001⟩)
(	010⟩)
(	011⟩)
(	100⟩)
(	101⟩)
(	110⟩)
(	111⟩)

1. Initial state: $|001\rangle$.
2. Transformation: The QFT maps $|j\rangle$ to a superposition state:

$$QFT\ (|001\rangle) = \frac{1}{\sqrt{2^3}} \sum_{k=1}^{7} e^{2\pi i - j - \frac{k}{8}} |k\rangle,$$

where j = 1.

Finding the state vector as:

$$\frac{1}{\sqrt{8}}(|0\rangle) + e^{\frac{ix}{4}}|1\rangle + e^{\frac{ix}{2}}|2\rangle + e^{\frac{3ix}{4}}4|3\rangle + e^{ix}|4\rangle + e^{\frac{5ix}{4}}|5\rangle + e^{\frac{3ix}{2}}|6\rangle + e^{\frac{7ix}{4}}|7\rangle).$$

3.1.3. Output probabilities

Measurement probabilities will reflect the amplitude of each basis state. For $|001\rangle$, probabilities are evenly distributed among all states due to the uniform superposition created by QFT or both implementations; the output is typically visualized using a histogram of measurement probabilities. Each bar corresponds to a basis state (e.g., $|000\rangle$,$|001\rangle$,...,$|111\rangle$), and the height represents the probability of observing that state.

- Measurement Results (Ideal Case):
  - $|000\rangle$:12.5%.
  - $|001\rangle$:12.5%.
  - $|010\rangle$:12.5%, and so on, for all 8 basis states.
- Noisy Case:
  - Hardware imperfections may cause deviations, with higher probabilities for some states due to noise or routing errors (IBM) and fewer errors in IonQ due to full connectivity.

The measurement probabilities for each basis state are summarized in Table 1

The table summarizes the theoretical measurement probabilities for the 3-qubit Quantum Fourier Transform (QFT) algorithm when applied to the initial state $|001\rangle$. Each basis state has an equal probability of $1/2^n = 0.125$, reflecting the uniform distribution created by the QFT. This uniformity is expected because the QFT generates an equal superposition of all basis

states, weighted by their respective phase factors. Table 1 highlights the core functionality of the QFT, demonstrating how the input state is transformed into a balanced superposition, a critical property leveraged in quantum algorithms like Shor's algorithm. This theoretical distribution assumes an ideal quantum computer with perfect fidelity, no noise, and infinite decoherence times. Real-world hardware implementations deviate from this due to practical constraints. The measurement results from the IBM Quantum implementation, depicted in Figure 1, show the impact of noise on the probability distribution.

Figure 2 represents the results of executing the QFT algorithm on IBM Quantum hardware. The probabilities deviate slightly from the theoretical uniform distribution due to the following factors. This figure shows that hardware noise, such as gate errors and qubit decoherence, introduces randomness in the output probabilities. With Qubit Connectivity, the need for additional SWAP gates to route qubits in IBM's linear topology increases circuit depth, compounding errors. Gate fidelities and qubit error rates vary across the hardware, impacting overall performance. Figure one observes that while the probabilities are close to uniform, certain basis states have slightly higher or lower probabilities. This deviation highlights the challenges of executing complex quantum circuits on near-term quantum devices.

As illustrated in Figure 2, the IonQ hardware achieves a nearly perfect uniform distribution, owing to its full connectivity and high fidelity.

Figure 2 displays the measurement probabilities for the same QFT algorithm executed on IonQ hardware. The results closely match the theoretical uniform distribution due to full connectivity, High Gate Fidelity, and Longer Decoherence respectively. IonQ's trapped-ion architecture eliminates the need for additional SWAP gates, reducing circuit depth and error accumulation. The native Mølmer–Sørensen (MS) gate achieves high entanglement fidelity, preserving the ideal quantum state during execution. Trapped-ion qubits have longer coherence times compared to superconducting qubits, ensuring better preservation of quantum information. At this point, it was observed that Figure 2 demonstrates a minimal deviation from the theoretical distribution, showcasing the advantage of IonQ's architecture for implementing complex quantum algorithms like the QFT.

3.1.4. Comparative analysis

1. Uniformity:
   (a) The IonQ implementation achieves a closer approximation to the theoretical uniform distribution compared to IBM Quantum.
   (b) This highlights the importance of hardware architecture and connectivity in determining algorithmic fidelity.
2. Noise impact
   (a) The IBM Quantum histogram shows more pronounced deviations, underscoring the limitations of superconducting qubits, particularly in handling routing and gate errors.
3. Scalability:
   (a) As the number of qubits increases, the IBM Quantum implementation would face significant challenges due to increased routing complexity, while IonQ's full connectivity provides a clear advantage for scalability.

Table 2 summarizes the comparative strengths and limitations of the two implementations, emphasizing the impact of hardware design on algorithm performance.

4. Quantum code generation compiler framework

The proposed model is presented in Figure 3:

The proposed framework for quantum code generation is designed with three primary modules: the Front-End Module, Optimization Module, and Back-End Module. Each module serves a specific purpose to ensure efficient and adaptable quantum code compilation. The Front-End Module is responsible for parsing and translating high-level quantum programming languages such as Qiskit, Cirq, and Q# into an intermediate representation (IR). This module includes language parsers to support multiple quantum languages, syntax and semantic validation for quantum code, and the generation of an abstract syntax tree (AST) that is subsequently converted into IR.

The Optimization Module focuses on quantum-specific optimizations to enhance performance and reduce hardware errors. It includes gate synthesis, which minimizes the total number of gates and ensures compatibility with native gate sets supported by the target hardware (e.g., CX gates for IBM Quantum or MS gates for IonQ). Additionally, it handles qubit routing by optimizing qubit connectivity through SWAP insertion or mapping algorithms, such as the SABRE algorithm, particularly for linear architectures. Error mitigation techniques are also incorporated, using device calibration data (e.g., gate error rates and decoherence times) to support methods like zero-noise extrapolation and probabilistic error cancellation. The Back-End Module translates the optimized IR into hardware-specific instructions and executes the code on the target quantum backend. This module supports backends for IBM Quantum, IonQ, Rigetti, and other platforms. It handles hardware-specific gate decomposition scheduling and integrates with quantum hardware APIs for execution and measurement retrieval.

A key feature of the framework is its hardware-agnostic intermediate representation (IR), which serves as a unified abstraction layer between high-level languages and hardware-specific instructions. The IR supports a broad set of quantum operations, abstracts hardware constraints like connectivity and native gate sets, and ensures compatibility with multiple backends by translating IR to hardware-specific instructions. For example, the IR workflow involves translating high-level code written in languages like Qiskit or Cirq into a hardware-agnostic representation of gates, qubits, and measurements, which is then converted into hardware-specific instructions tailored for the target backend.

The implementation workflow begins with the Front-End Module, where high-level quantum code is parsed and converted into IR. This IR is then passed to the Optimization Mod-

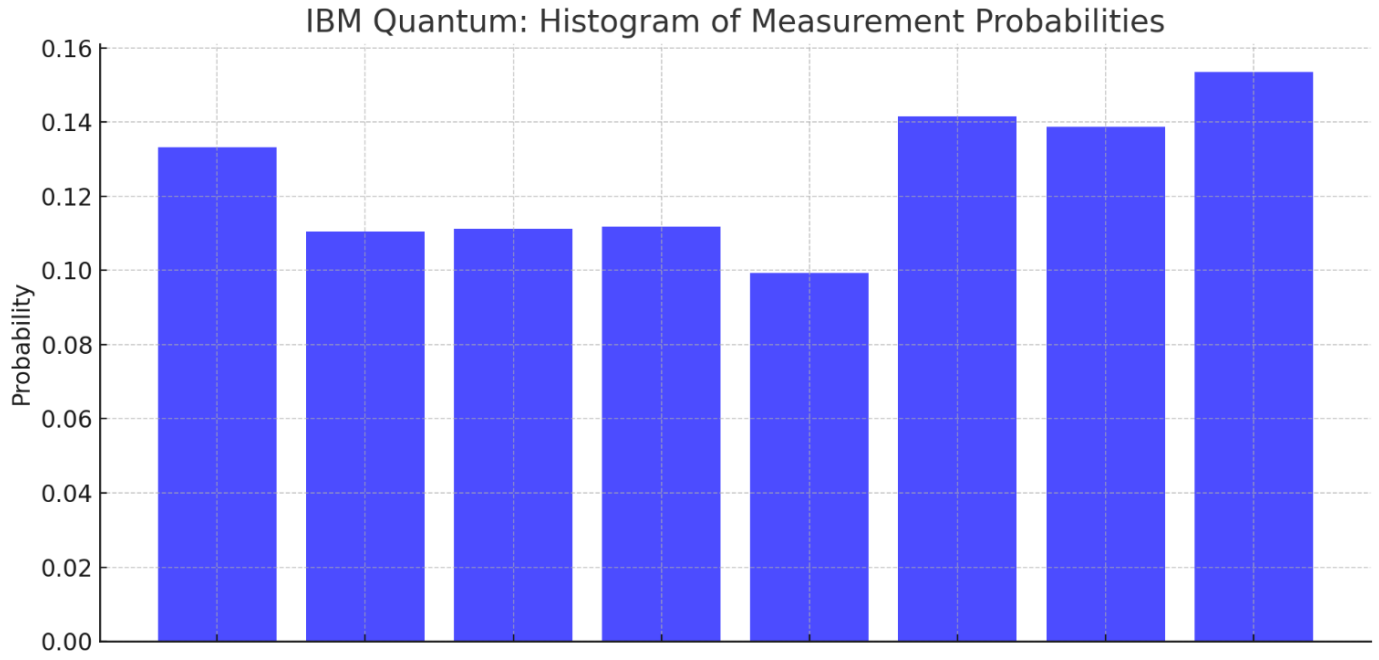

Figure 1. Histogram of measurement probabilities for the IBM Quantum implementation of the 3-qubit QFT algorithm.

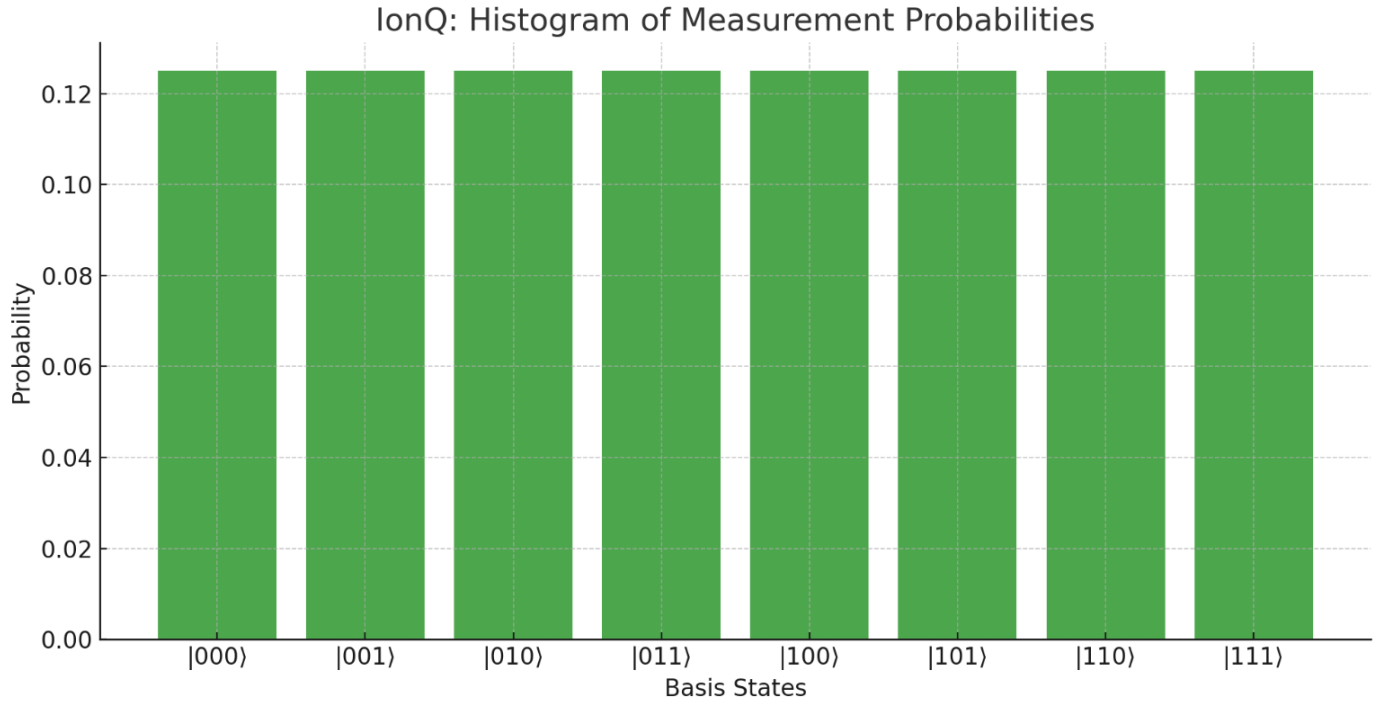

Figure 2. Histogram of measurement probabilities for the IonQ implementation of the 3-qubit QFT algorithm.

ule, where gate synthesis, qubit routing, and error mitigation are applied to produce an optimized IR. Finally, the Back-End Module translates the optimized IR into hardware-specific instructions, executes the code on the quantum backend, and retrieves measurement results.

This framework offers several benefits. Its modularity allows for the seamless addition of new front-end languages, optimization techniques, and back-end hardware. It is extensible, supporting advancements in quantum programming and hardware capabilities. The hardware-agnostic design ensures cross-platform compatibility, enabling developers to target multiple quantum devices with minimal code changes. Additionally, the optimization techniques improve execution fidelity and reduce resource requirements. Overall, this framework provides a scal-

Table 2. Comparative analysis of IBM quantum and IonQ implementations.

Aspect	IBM quantum	IonQ
Uniformity	Deviations from theoretical uniform distribution due to noise, gate errors, and routing issues.	Closely matches theoretical uniform distribution, benefiting from full connectivity and high fidelity.
Noise Impact	More pronounced due to limited connectivity, requiring SWAP gates and deeper circuits.	Minimal impact as full connectivity eliminates the need for SWAP gates, reducing circuit depth.
Scalability	Faces challenges as qubit count increases due to routing complexity and cumulative errors.	Scales are better due to full connectivity and reduced circuit depth, maintaining high fidelity.

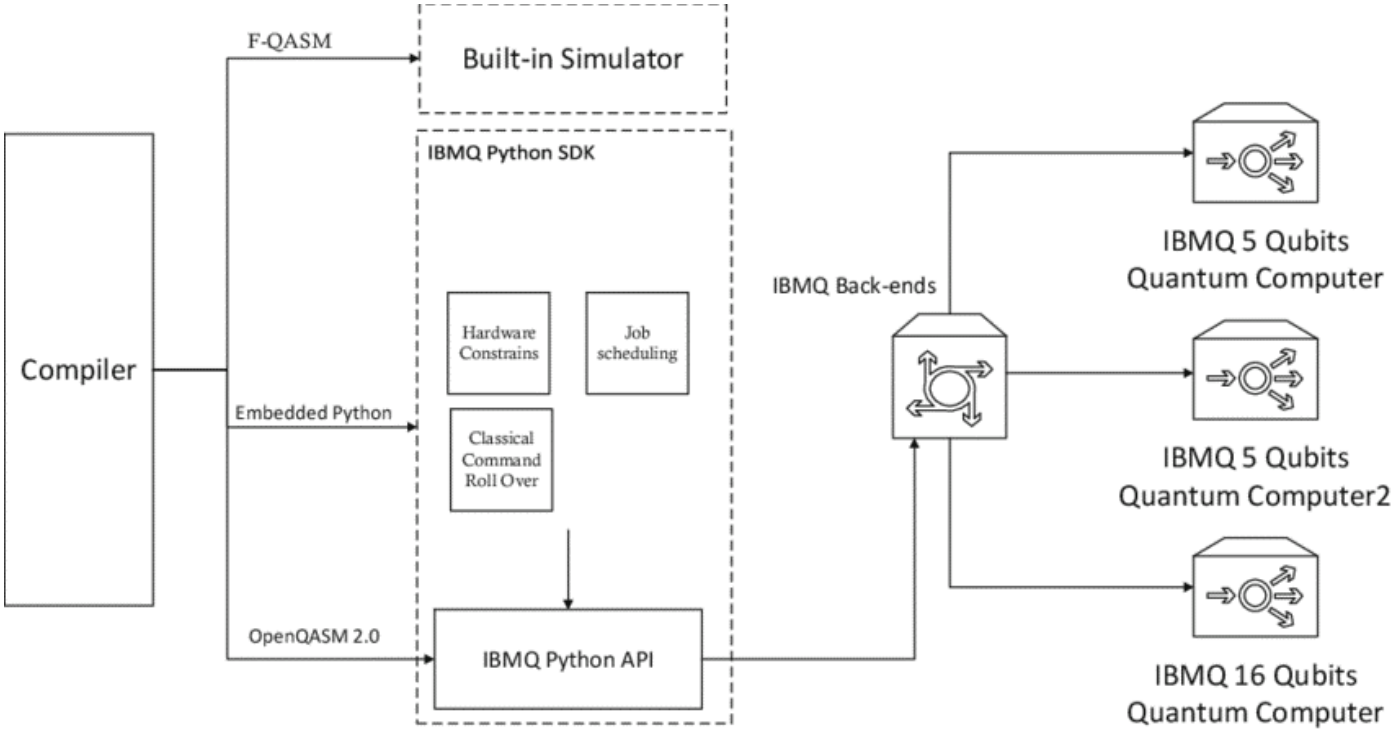

Figure 3. A modular and extensible compiler framework for quantum code generation.

able and adaptable solution for efficient quantum code generation, addressing the unique challenges posed by quantum computing.

4.1. Optimization strategies

Consider the following quantum circuit, written for a device with limited qubit connectivity. To show the initial circuit, there is a need to print or visualize the actual quantum circuit object, typically using a quantum computing framework like Qiskit. When working with Qiskit, it is pertinent to define and visualize the quantum circuit as follows:

```
from qiskit import QuantumCircuit
# Define a quantum circuit with 3 qubits
qc = QuantumCircuit(3)
# Apply some gates
qc.h(0) # Hadamard on qubit 0
qc.cx(0, 1) # CNOT from qubit 0 to 1
qc.cx(1, 2) # CNOT from qubit 1 to 2
qc.cx(0, 1) # CNOT from qubit 0 to 1 (repeated)
qc.z(1) # Z gate on qubit 1
qc.measure_all() # Measure all qubits
print("Initial Circuit:")
print(qc)
```

Figure 4 represents a quantum circuit with 3 qubits, and some gates applied to them This circuit includes redundant gates non-optimal routing and does not exploit hardware-specific native gates.

To simplify the redundant gates and commuting operations, the second CX(0,1) gate cancels with the first because applying two consecutive CX gates on the same qubits is equivalent to the identity operation. After the gate cancellation, the optimized circuit is obtained from the algorithm:

```
qc_optimized = QuantumCircuit(3)
qc_optimized.h(0) # Hadamard on qubit 0
qc.cx(0, 1) # CNOT from qubit 0 to 1
qc.cx(1, 2) # CNOT from qubit 1 to 2
qc.z(1) # Z gate on qubit 1
qc.measure_all() # Measure all qubits
print("Optimized Circuit (Gate Cancellation):")
print(qc_optimized)
```

Suppose the hardware has limited linear connectivity (q[0] ↔ q[1] ↔ q[2]). For this topology:

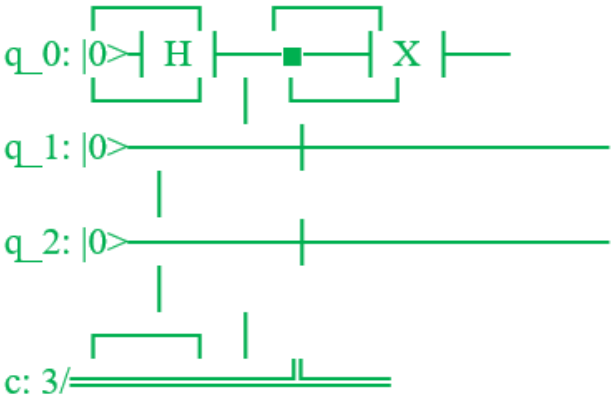

Figure 4. Initial circuit.

1. A CX(1, 2) is feasible, but a CX(0, 2) would require a SWAP operation.
2. Qubit mapping can minimize SWAP gates by allocating logical qubits to match hardware topology.

- Using Qiskit's transpiler:

  from qiskit import transpile

  # Transpile the circuit for a linear hardware topology

  backend = FakeLinearLattice() # Replace with actual backend or topology

  optimized_routed_circuit = transpile(qc_optimized, backend=backend)

  print("Optimized Circuit (Qubit Routing):")

  print(optimized_routed_circuit)

  Most quantum devices support specific native gate sets or pulse-level programming for higher fidelity. IBM Quantum devices use CX gates and single-qubit rotations as native gates. IonQ devices natively support multi-qubit Mølmer–Sørensen (MS) gates, which can replace a series of CX gates.

- Using pulse-level programming (for IBM Quantum): from qiskit.pulse import Schedule, DriveChannel, Gaussian

  # Create a pulse schedule for a single-qubit gate

  schedule = Schedule()

  drive_channel = DriveChannel(0)

  pulse = Gaussian(duration=128, amp=0.1, sigma=16)

  schedule += pulse(drive_channel)

  print("Pulse-Level Schedule:")

  print(schedule)

  After applying all the above optimizations:

  1. Gate cancellation reduces unnecessary operations.
  2. Qubit allocation and routing minimize communication overhead.

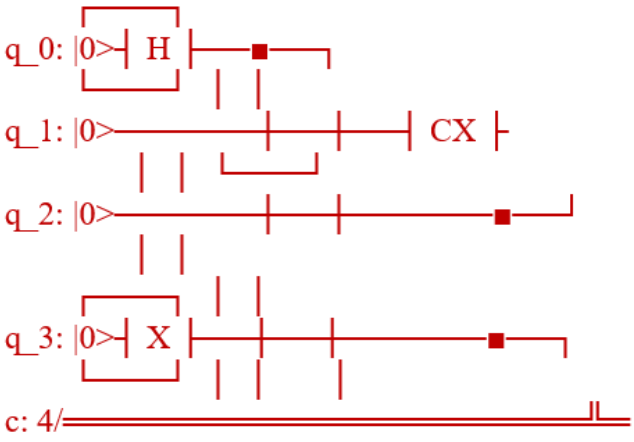

Figure 5. Optimized circuit.

3. Hardware-specific optimizations leverage native gates or pulse-level programming to enhance fidelity.

The final optimized circuit and/or schedule was produced for execution on the target quantum hardware. These techniques ensure better performance and higher reliability of quantum computations.

To visualize the output of the code print(optimized_routed_circuit.draw("text")), the result is typically a textual representation of the quantum circuit.

In this work, we used a Python environment with Qiskit to simulate quantum circuits by running the code optimized_routed_circuit. Draw ("text") the quantum circuit in a text-based format was printed as presented in Figure 5. This textual representation of the quantum circuit shows the qubits and gates in a readable form, with each gate (like H, CX, X) represented on the respective qubit lines.

4.2. Discussion

This work focuses on developing a compiler-assisted code generation framework for quantum computing, addressing the challenges posed by the unique properties of quantum hardware. The framework is designed to optimize the translation of high-level quantum algorithms into hardware-specific instructions, ensuring efficient and reliable execution on various quantum devices. By leveraging quantum-specific optimizations, the framework enhances the performance and fidelity of quantum computations while supporting scalability and adaptability across diverse hardware platforms. The framework comprises three key modules: a front-end, an optimization module, and a back-end. The front end processes high-level quantum programming languages such as Qiskit, Cirq, and Q#, converting them into a hardware-agnostic intermediate representation (IR). This IR serves as a unified abstraction layer, enabling compatibility with multiple quantum backends. The optimization module applies quantum-specific techniques, including gate synthesis to reduce gate complexity, qubit routing to minimize communication overhead, and error mitigation strategies to address

noise and hardware imperfections. The back-end translates the optimized IR into hardware-specific instructions tailored for platforms like IBM Quantum, IonQ, and Rigetti. A practical example illustrates the effectiveness of the framework by optimizing a quantum circuit through techniques such as gate cancellation, commutation, and hardware-aware routing. The process highlights the reduction in gate complexity and communication overhead, leading to improved execution fidelity. Visualization of the initial and optimized circuits provides insights into the transformations, emphasizing the framework's capability to adapt algorithms to hardware constraints effectively.

Comparative analysis of implementations on IBM Quantum and IonQ devices demonstrates the significance of hardware-specific optimizations. IonQ's full qubit connectivity offers advantages in scalability and uniformity, while IBM Quantum faces challenges due to increased routing complexity and noise in superconducting qubits. These findings underscore the critical role of hardware-aware compilation in achieving reliable quantum computations. The work presents a modular, extensible, and hardware-agnostic framework that addresses key challenges in quantum code generation. By bridging the gap between high-level programming and hardware-specific execution, the framework provides a scalable solution for developing efficient quantum applications, paving the way for advancements in quantum computing.

4.2.1. Discussions of practical implementation

The proposed compiler-assisted code generation framework was implemented using Qiskit, an open-source quantum computing framework, within a Python environment. The framework was designed to translate high-level quantum programs into hardware-specific instructions, ensuring efficient execution across multiple quantum platforms, including IBM Quantum and IonQ. Taking advantage of gate synthesis, qubit routing, and error mitigation, the framework improved execution fidelity and scalability. The modular architecture allowed seamless adaptation to different quantum processors, ensuring efficient and hardware-aware quantum computation. The results demonstrate that hardware-aware optimizations play a crucial role in enhancing the performance of quantum algorithms on current quantum hardware.

5. Conclusion

This work presents a comprehensive approach to addressing the challenges of quantum code generation through a modular and extensible compiler framework. By leveraging quantum-specific optimizations such as gate synthesis, qubit routing, and error mitigation, the framework effectively adapts quantum algorithms to the unique constraints of various quantum hardware architectures. The use of a hardware-agnostic intermediate representation (IR) ensures compatibility across multiple backends, facilitating scalable and platform-independent quantum application development. The practical implementation of circuit optimization demonstrates the framework's ability to minimize gate complexity, reduce communication overhead, and enhance execution fidelity. Comparative analysis further highlights the impact of hardware-specific properties, such as qubit connectivity and gate fidelity, on algorithmic performance. Devices like IonQ, with their full connectivity, showcase superior scalability, while superconducting platforms like IBM Quantum face routing and noise challenges that demand more sophisticated compilation techniques. In conclusion, this framework bridges the gap between high-level quantum programming and hardware-specific execution, providing a robust solution for efficient quantum computation. Its modular design ensures adaptability to evolving quantum hardware and programming paradigms, making it a valuable tool for advancing the development of reliable and scalable quantum applications. By addressing key challenges in quantum computing, this work lays the foundation for future innovations in quantum software and hardware integration.

Data availability

We do not have any research data outside the submitted manuscript file.

References

[1] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring", Proc. 35th Annu. Symp. Found. Comput. Sci., pp. 124–134, 1994. https://ieeexplore.ieee.org/abstract/document/365700/.

[2] I. L. Chuang & M. A. Nielsen, *Quantum Computation and Quantum Information*, 10th ed., Cambridge University Press, 2010. https://www.cambridge.org/highereducation/books/quantum-computation-and-quantum-information/01E10196D0A682A6AEFFEA52D53BE9AE.

[3] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta & B. R. Johnson, "OpenQASM 3: A broader and deeper quantum assembly language", ACM Transactions on Quantum Computing **3** (2022) 3. https://doi.org/10.1145/3505636.

[4] D. Wecker & K. M. Svore, "LIQUi|>: A software design architecture and domain-specific language for quantum computing", Quantum Sci. Technol. **2** (2017) 015006. https://www.researchgate.net/publication/260268455_LIQUi_A_Software_Design_Architecture_and_Domain-Specific_Language_for_Quantum_Computing.

[5] S. S. Tannu & M. K. Qureshi, "Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers", Proc. 24th Int. Conf. Arch. Support Program. Lang. Oper. Syst. 2019 987. https://dl.acm.org/doi/abs/10.1145/3297858.3304007.

[6] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong & M. V. Martonosi, "ScaffCC: Scalable compilation and analysis of quantum programs", Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal., pp. 1–11, 2014. https://oar.princeton.edu/handle/88435/pr1768q.

[7] P. Selinger, "Efficient Clifford+T approximation of single-qubit operators", Quantum Inf. Comput. **15** 2015 159. https://ui.adsabs.harvard.edu/abs/2012arXiv1212.6253S/abstract.

[8] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington & R. Duncan, "t|ket>: A retargetable compiler for NISQ devices", Quantum Sci. Technol. **6** 2021 014003. https://iopscience.iop.org/article/10.1088/2058-9565/ab8e92/meta.

[9] A. D. Córcoles, A. Kandala, A. Javadi-Abhari, D. T. McClure, A. W. Cross, K. Temme, P. D. Nation, M. Steffen & J. M. Gambetta, "Challenges and opportunities of near-term quantum computing systems", Proc. IEEE Int. Symp. High-Perf. Comput. Arch., pp. 1–6, 2021. https://ieeexplore.ieee.org/abstract/document/8936946/.

[10] A. G. Fowler, M. Mariantoni, J. M. Martinis & A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation", Phys. Rev. A **86** 2012 032324. https://journals.aps.org/pra/abstract/10.1103/PhysRevA.86.032324.

[11] T. Häner, D. S. Steiger, K. Svore & M. Troyer, "A software methodology for compiling quantum programs", Proc. ACM/IEEE Design Autom. Conf., pp. 1–6, 2016. https://iopscience.iop.org/article/10.1088/2058-9565/aaa5cc/meta.

[12] M. Amy, D. Maslov, M. Mosca & M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **32** 2013 818. https://ieeexplore.ieee.org/abstract/document/6516700/.

[13] Y. Ge, W. Wenjie, C. Yuheng, P. Kaisen, L. Xudong, Z. Zixiang, W. Yuhan, W. Ruocheng & Y. Junchi, "Quantum circuit synthesis and compilation optimization: overview and prospects", Quantum Physics **104** 2021 022403. https://openreview.net/forum?id=PlJz4JUOYh.

[14] M. Saeedi, R. Wille & Rolf Drechsler, "Synthesis of quantum circuits for linear nearest neighbor architectures", Quantum Information Processing **10** (2011) 355377. https://doi.org/10.1007/s11128-010-0201-2.

[15] K. Mitarai, M. Negoro, M. Kitagawa & K. Fujii, "Quantum circuit learning", Phys. Rev. A **98** 2018 032309. https://journals.aps.org/pra/abstract/10.1103/PhysRevA.98.032309.

[16] K. Mitarail, M. Negoro, M. Kitagawa & K. Fujii, "Quantum circuit learning", Physical Review Journal **98** (2020) 032309. https://doi.org/10.1103/PhysRevA.98.032309.

[17] F. Wagner, D. J. Egger & F. Liers, "Optimized noise suppression for quantum circuits", Informs Journal of Computing **37** (2024) 1. https://doi.org/10.1287/ijoc.2024.0551.

[18] D. Gottesman, "The Heisenberg representation of quantum computers", Proc. XXII Int. Colloq. Group Theor. Methods Phys., 1998. https://ui.adsabs.harvard.edu/abs/1998quant.ph..7006G/abstract.

[19] J. Preskill, "Quantum computing in the NISQ era and beyond", Quantum **2** 2018 79. https://quantum-journal.org/papers/q-2018-08-06-79/.

[20] M. Suchara, J. Kubiatowicz, A. Faruque, F. T. Chong, C. Y. Lai & G. Paz, "QuRE: The quantum resource estimator tool", Proc. Int. Conf. Quantum Comput. Eng., pp. 1–8, 2020. https://ieeexplore.ieee.org/abstract/document/6657074/.

[21] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F.G. Brandao, D. A. Buell & B. Burkett, "Quantum supremacy using a programmable superconducting processor", Nature **574** 2019 505. https://www.nature.com/articles/s41586-019-1666-5.

[22] A revolutionary computing capability to sift through huge numbers of possibilities and extract potential solutions to complex problems, by A. Valavanidis. (2024, 16 January). [Online]. Available: Quantum Computing.

[23] Z. Cai, R. Babbush, S. C. Benjamin, S. Endo, W. J. Huggins, Y. Li, J. R. McClean & T. E. O'Brien, "Quantum error mitigation", Reviews of Modern Physics **95** (2023) 045005. https://journals.aps.org/rmp/abstract/10.1103/RevModPhys.95.045005.

[24] L. Egan, D. M. Debroy, C. Noel, A. Risinger, D. Zhu, D. Biswas, M. Newman, M. Li, K. R. Brown, M. Cetina & C. Monroe, "Fault-tolerant operation of a quantum error-correction code", Quantum Physics. https://inspirehep.net/literature/2727223.

[25] Y. Cao, A. Daskin, S. Frankel & S. Kais, "Quantum circuit design for solving linear systems of equations", An International Journal at the Interface Between Chemistry and Physics **110** (2012) 16. https://www.tandfonline.com/doi/abs/10.1080/00268976.2012.668289.

[26] Y. R. Sanders, D. W. Berry, P. C. Costa, L. W. Tessler, N. Wiebe, C. Gidney, H. Neven & R. Babbush, "Compilation of fault-tolerant quantum circuits", Phys. Rev. A, **101** 2020 042304. https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.1.020312.

[27] R. Raussendorf & H. J. Briegel, "A one-way quantum computer", Phys. Rev. Lett. **86** 2001 5188. https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.86.5188.

[28] L. Botelho, A. Glos, A. Kundu, J. A. Miszczak, Ö. Salehil & Z. Zimborás, "Error mitigation for variational quantum algorithms through mid-circuit measurements", Phys. Rev. A **105** (2022) 022441. https://journals.aps.org/pra/abstract/10.1103/PhysRevA.105.022441.

[29] T. Lubinski, C. Coffrin, C. McGeoch, P. Sathe, J. Apanavicius, D. B. Neira & Quantum Economic Development Consortium(QED-C) Collaboration, "Optimization applications as quantum performance benchmarks", ACM Transactions on Quantum Computing **5** (2023) 144. https://doi.org/10.1145/3678184.

[30] F. Hua, Y. Chen, Y. Jin, C. Zhang, A. Hayes, Y. Zhang & E. Z. Zhang, "AutoBraid: a framework for enabling efficient surface code communication in quantum computing", MICRO '21: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture **925** (2021) 936. https://dl.acm.org/doi/abs/10.1145/3466752.3480072.

[31] P. Selinger, "A brief survey of quantum programming languages", in Proceedings of the 7th International Conference on Mathematics of Program Construction, Stirling, UK, Jul. 2004, pp. 1–15. https://link.springer.com/chapter/10.1007/978-3-540-24754-8_1.

[32] T. Häner, D. S. Steiger, M. Smelyanskiy & M. Troyer, "High-performance emulation of quantum circuits", in Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, Nov. 2016, pp. 866–874. https://ieeexplore.ieee.org/abstract/document/7877152.

[33] M. Zhang, J. Wang & J. Lai, "Performance evaluation of quantum computing processors based on quantum assembly language", 9th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), 2024. https://ieeexplore.ieee.org/abstract/document/10569330/.

[34] A. Cowtan, S. T. Dill, R. Duncan, A. Krajenbrink, W. Simmons & S. Sivarajah, "On the qubit routing problem", in Proceedings of the 14th International Conference on Quantum Physics and Logic (QPL), Nijmegen, Netherlands, Jul. 2017, pp. 5–19. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TQC.2019.5.

[35] F. J. Cardama, J. Vázquez-Pérez, C. Piñeiro, J. C. Pichel, T. F. Pena & A. Gómez, "Review of intermediate representations for quantum computing", The Journal of Supercomputing **81** (2025) 418. https://link.springer.com/article/10.1007/s11227-024-06892-2.

[36] M. Amy, D.Maslov & M. Mosca, "Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32** 2013 818. https://doi.org/10.1109/TCAD.2013.2244643.

[37] G. G. James, E. G. Chukwu & P. O. Ekwe, "Design of an intelligent based system for the diagnosis of lung cancer," Int. J. Innov. Sci. Res. Technol. **12** (2023) 791. https://www.academia.edu/105440235/Design_of_an_Intelligent_based_System_for_the_Diagnosis_of_Lung_Cancer?uc-g-sw=71431716.

[38] C. Ituma, G. G. James & F. U. Onu, "Implementation of intelligent document retrieval model using neuro-fuzzy technology," Int. J. Eng. Appl. Sci. Technol. **4** (2020) 65. .

[39] C. I. Iwok, S. Obot & G. G. James, "A model of intelligent packet switching in wireles communication networks," Int. J. Sci. Eng. Res. **11** (2020) 64. .

[40] C. I. Ituma, Iwok, S. Obot & G. G. James, "Implementation of an optimized packet switching parameters in wireless communication networks", Int. J. Sci. Eng. Res. **11** (2020) 58. .

[41] G. G. James, A. E. Okpako C. Ituma, & J. E. Asuquo, "Development of Hybrid Intelligent based Information Retrieval Technique", Int. J. Comput. Appl. **184** (2022) 13. dhttps://unidel.edu.ng/cms/uploads/publications/unidel_pub_1689455282.pdf.

[42] Ituma, G. G. James & F. U. Onu, "A neuro-fuzzy based document tracking & classification system", Int. J. Eng. Appl. Sci. Technol. **4** (2020) 414. https://unidel.edu.ng/cms/uploads/publications/unidel_pub_1689455282.pdf.

[43] M. Y. Siraichi, V. F. Santos, S. Collange & F. M. Q. Pereira, "Qubit allocation", Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 113–125. https://dl.acm.org/doi/abs/10.1145/3168822.

[44] F. T. Chong, K. Y. Tan, Y. Ding, Y. Shi, W. J. Zeng, C. J. Lin & S. Yong, "A quantum computing compiler and design flow", in Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 52–63. https://dl.acm.org/doi/pdf/10.1145/329-7858.3304075.

[45] A. Bocharov, M. Roetteler & K. M. Svore, "Efficient synthesis of universal repeat-until-success quantum circuits", Physical Review Letters **114** 2017 080502. https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.114.080502.

[46] A. Zulehner, A. Paler & R. Wille, "Efficient mapping of quantum circuits to the IBM QX architectures", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38** 2018 1226. https://ieeexplore.ieee.org/document/8342181.

[47] P. Murali, "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 (ASPLOS), 2019, pp. 1015–1029. https://dl.acm.org/doi/10.1145/3297858.3304075.

[48] Anietie Ekong, Immaculata Attih, Gabriel James & Unyime Edet, "Effective Classification of Diabetes Mellitus Using Support Vector Machine Algorithm", Res. J. Sci. Technol. **4** (2024) 34. https://www.rejost.com.ng/index.php/home/article/view/97.

[49] C. I. Iwok, S. Obot & G. G. James, "A model of intelligent packet switching in wireles communication networks", Int. J. Sci. Eng. Res. 11 (2020) 64. .

[50] C. I. Ituma, Iwok, Sunday Obot & G. G. James, "Implementation of an optimized packet switching parameters in wireless communication networks", Int. J. Sci. Eng. Res. 11 (2020) 58. .

[51] A. P. Ekong, G. G. James & I. Ohaeri, "Oil and Gas Pipeline Leakage Detection using IoT and Deep Learning Algorithm", J. Inf. Syst. Inform. 6 (2024) 421. https://pdfs.semanticscholar.org/9acb/077b59f349d6a60df2d6703d5e3741cf3132.pdf.

[52] K. Hietala, R. Rand, S. Hung, X. Wu & Michael Hicks, "A verified optimizer for Quantum circuits", Proceedings of the ACM on Programming Languages **5** (2023) 29. https://doi.org/10.1145/3434318.

[53] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation & J. M. Gambetta, "Open Quantum assembly language", arXiv preprint, 2017. https://ui.adsabs.harvard.edu/abs/2017arXiv170703429C/abstract.

[54] A. G. Fowler, M. Mariantoni, J. M. Martinis & A. N. Cleland, "Surface codes: Towards practical large-scale Quantum computation", Physical Review A **86** (2012) 032324. https://journals.aps.org/pra/abstract/10.1103/PhysRevA.86.032324.