DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

DQS: A Framework for Efficient Distributed Simulation of Large Quantum Circuits

Nathaniel Tornow

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

DQS: A Framework for Efficient Distributed Simulation of Large Quantum Circuits

DQS: Ein Programm für die Effiziente Verteilte Simulation von Großen Quantenschaltungen

Author:	Nathaniel Tornow
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisors:	Emmanouil Giortamis, Prof. Dr.-Ing. Pramod Bhatotia
Submission Date:	15.08.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2022　　　　　　　　Nathaniel Tornow

Acknowledgments

First and foremost, I am very grateful to Professor Pramod Bhatotia for welcoming me to the chair, for giving me the opportunity to work on exciting research projects, and for his kind mentorship. I would also like to sincerely thank my advisor Emmanouil Giortamis for his support and guidance during the time of this project.

We acknowledge the use of IBM Quantum services for this work and to advanced services provided by the IBM Quantum Researchers Program. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

Abstract

Quantum computers promise to solve otherwise intractable computational problems. They may efficiently simulate quantum physics and chemistry, factorize large numbers, find solutions to combinatorial optimization problems, or perform quantum machine learning tasks. However, current quantum computers are subject to severe limitations, including noise and a limited number of qubits. In order to test reasonable applications and algorithms on current, noise-impaired quantum computing hardware, strategies must be found to overcome or mitigate these limitations. To this end, the technique of gate virtualization has shown promise as a divide-and-conquer technique for optimizing quantum circuits by dividing large quantum circuits into multiple smaller and less noisy circuit fragments. However, this technique incurs a significant exponential cost of $\mathcal{O}(6^k)$ for virtualizing k binary gates, both in quantum and classical computations. Therefore, it is imperative to find strategies that make optimal use of gate virtualization while minimizing the computational cost. To advance research in finding optimal strategies, we introduce the Distributed Quantum System (DQS) framework in this work. It provides a novel virtual circuit abstraction that allows researchers to easily define automated circuit optimizations using gate virtualization in a transparent manner. For virtual circuit execution, we present an execution system that efficiently executes virtual circuits with distributed quantum and classical computations. We implement a set of circuit transpilers that make optimal use of gate virtualization. Our investigation on the 7-qubit processor IBM Falcon r5.11H shows that using our transpilers in addition to state-of-the-art transpilers improves accuracy by up to 1.5× and enables the execution of circuits more than twice the size of the quantum device without sacrificing accuracy.

Kurzfassung

Quantencomputer versprechen sonst unlösbare Rechenprobleme zu lösen. Sie können effizient Quantenphysik und -chemie simulieren, große Zahlen faktorisieren, Lösungen für kombinatorische Optimierungsprobleme finden oder Aufgaben des maschinellen Lernens auf Quantenbasis durchführen. Die derzeitigen Quantencomputer unterliegen jedoch schwerwiegenden Einschränkungen, darunter Rauschen und eine begrenzte Anzahl von Qubits. Um sinnvolle Anwendungen und Algorithmen auf aktueller, rauschbehinderter Quantencomputer-Hardware zu testen, müssen Strategien gefunden werden, um diese Einschränkungen zu überwinden oder abzumildern. Zu diesem Zweck hat sich die Technik der Gattervirtualisierung als ein vielversprechendes Verfahren zur Optimierung von Quantenschaltungen erwiesen, indem große Quantenschaltungen in mehrere kleinere und weniger rauschende Schaltkreisfragmente unterteilt werden. Allerdings verursacht diese Technik erhebliche exponentielle Kosten von $\mathcal{O}(6^k)$ für die Virtualisierung von k binären Gattern, sowohl bei Quanten- als auch bei klassischen Berechnungen. Daher ist es zwingend notwendig, Strategien zu finden, die die Gattervirtualisierung optimal nutzen und gleichzeitig die Rechenkosten minimieren. Um die Forschung bei der Suche nach optimalen Strategien voranzutreiben, stellen wir in dieser Arbeit das Distributed Quantum System (DQS) Framework vor. Es bietet eine neuartige Abstraktion für virtuelle Schaltkreise, die es Forschern ermöglicht, automatisierte Schaltkreisoptimierungen unter Verwendung von Gattervirtualisierung auf transparente Weise zu definieren. Für die Ausführung virtueller Schaltungen präsentieren wir ein Ausführungssystem, das virtuelle Schaltungen mit verteilten Quanten- und klassischen Berechnungen effizient ausführt. Wir implementieren eine Reihe von Schaltungstranspilern, die die Gattervirtualisierung optimal nutzen. Unsere Untersuchung auf dem 7-Qubit-Prozessor IBM Falcon r5.11H zeigt, dass die Verwendung unserer Transpiler zusätzlich zu den existierenden Transpilern die Genauigkeit um das bis zu 1,5× verbessert und die Ausführung von Schaltungen ermöglicht, die mehr als doppelt so groß sind wie das Quantengerät, ohne dass die Genauigkeit darunter leidet.

Contents

1 Introduction

Quantum computing is a new computing paradigm that enables exponential speedups for certain algorithms by exploiting the phenomena of quantum mechanics. Research areas on which quantum computing will likely have a major impact include quantum physics, chemistry, finance, and machine learning [1]. Remarkable technological advances have made a large number of quantum computers available in recent years [2, 3, 4]. However, quantum computing is currently in the so-called noisy intermediate-scale (NISQ) era [5], meaning that today's quantum hardware prevents it from outperforming classical computation for certain algorithms due to the following two main problems [6]: On the one hand, quantum computers are still very small; the largest quantum computer is expected to have only about 400 qubits (computational units) by the end of 2022 [7]. However, running quantum algorithms for complex real problems may require several million qubits [8]. On the other hand, quantum computing is severely limited by the quality of current quantum hardware [5]. This generally means that executing quantum circuits on quantum devices leads to noisy results, and adding more qubits and operations to quantum circuits greatly amplifies this noise (see Section 2.1.6).

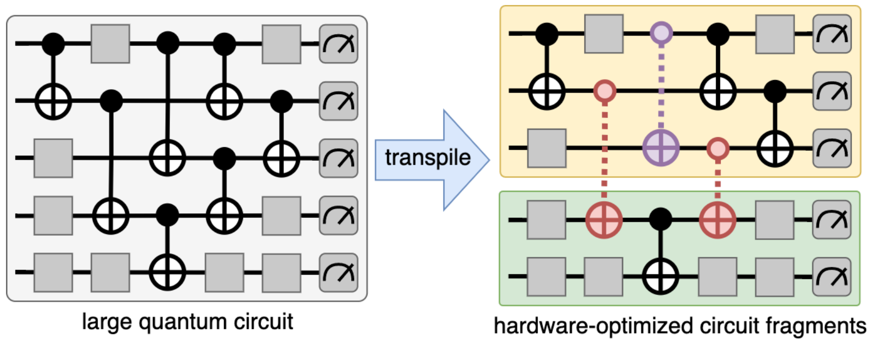


Figure 1.1: Using gate virtualization to optimize a large quantum circuit to run on noisy and intermediate scale quantum (NISQ) hardware. Gate virtualization allows the decomposition into multiple smaller fragments (red dotted gates) and noise-mitigation when executing fragments (purple dotted gate in top fragment).

One of the main focuses of quantum research is the optimization of quantum circuits to run as performant as possible despite the current noisy and intermediate-scale quantum hardware. As a new area of circuit optimization, the promising techniques of circuit cutting and knitting, especially **gate virtualization**, have attracted much interest [9, 10, 11]. This technique can be used to virtualize binary gates (Figure 1.1), which means that at runtime of the virtualization process, the binary gate is removed from the circuit, but hybrid quantum and classical computations allows us to obtain the same result as if the original quantum circuit had been executed. This results in two important opportunities that can be used to optimize the circuit. First, because of the removed dependencies between qubits during computation, large quantum circuits can be decomposed into multiple fragments that can be executed independently. This is useful because the smaller fragments fit on current small hardware and often produce less noise during execution. Second, binary gates are one of the main sources of noise and the most difficult part of a quantum circuit to optimize. Therefore, simply virtualizing a binary gate can significantly reduce the overall noise, even if we do not decompose the circuit by virtualizing the gate (top virtual gate in Figure 1.1).

Nevertheless, when we virtualize gates as circuit optimization, we have to trade the gained fidelity (the quality of the execution result) with a significant exponential computational cost of up to $\mathcal{O}(6^k)$ for the execution of the virtualized circuit when virtualizing k binary gates. Therefore, the number of virtual gates that can be computed in a reasonable amount of time is very limited, and it is imperative to develop optimal strategies that determine the few binary gates that achieve the highest noise reduction when virtualized. To date, however, gate virtualization has been explored mainly theoretically, and there is not yet a system that allows users to optimally utilize and design this new class of circuit optimization for quantum algorithms.

To this end, we present the design of the end-to-end DQS system with the following contributions:

- We design the novel abstractions of a virtual circuit (Section 4.1) and a distributed transpilation (Section 4.2) that allow users to define optimizations of circuits, customizable in all aspects that gate virtualization entails (Section 3.1.1).
- We describe the general approaches to distributed transpilation that make the most of gate virtualization, and discuss the implementation of a set of commonly used distributed transpiler passes (Section 5).
- For an efficient computation of gate virtualizations, we introduce the use of approximation techniques in order to find a better trade-off between the computational cost and the gained fidelity for specific circuits (Section 5.4).
- For the execution of the virtual circuits, we introduce an execution engine that allows parallel execution of circuit fragments and scalable classical post-processing to compute the gate virtualizations (Section 6).
- The evaluation of our distributed transpilers shows that we gain up to 1.5× fidelity in comparison to state-of-the-art transpilers and can scale the sizes of executable circuits to 2.3× the size of the quantum device the circuit runs on (Section 7).

2 Background

In this chapter we establish the necessary background for this work by giving a brief introduction into qubits, unary and binary gates, quantum circuits, quantum hardware and the technique of binary gate virtualization.

2.1 Quantum Computing

2.1.1 Qubit

A qubit (quantum-bit) in quantum computing is synonymous to a bit in classical computing. Similar to a bit, a qubit can have the states of 0 or 1, denoted as $|0\rangle$ and $|1\rangle$, where the two states are the vectors

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

However, other than a classical bit, qubits can also have the state of a *combination* of $|0\rangle$ and

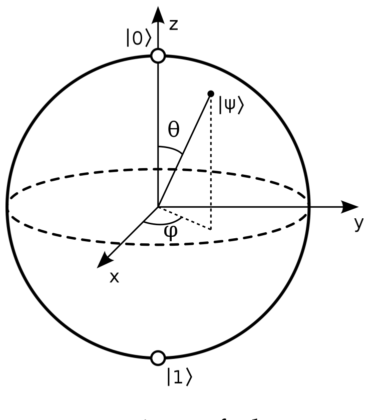


Figure 2.1: The Bloch sphere representation of the state vector: $|\psi\rangle = \cos(\theta/2)\,|0\rangle + \sin(\theta/2)e^{i\phi}\,|1\rangle$. In cartesian coordinates, the state vector has the x, y and z components: $(x, y, z) = (\sin(\theta) \cdot \cos(\phi), \sin(\theta) \cdot \sin(\phi), \cos(\theta))$.

$|1\rangle$, which is called a *superposition* of the two states. Subsequently, a state $|\psi\rangle$ of a qubit is represented as a two-dimensional, complex vector of the length of 1 that results from a linear combination of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = a_0\,|0\rangle + a_1\,|1\rangle \;\wedge\; a_0, a_1 \in \mathbb{C} \;\wedge\; |a_0|^2 + |a_1|^2 = 1 \tag{2.1}$$

Thus, the state of a qubit is often graphically represented as a point on the *Bloch sphere* [12] (see Figure 2.1) that the general state vector $|\psi\rangle = \cos(\theta/2)\,|0\rangle + \sin(\theta/2)e^{i\phi}\,|1\rangle$ points to, where $0 \leq \phi < 2\pi$, and $0 \leq \theta \leq \pi$.
By convention, the Bloch sphere lies in a 3-dimensional coordinate system of the three axes X, Y and Z, with $(x, y, z) = (\sin(\theta) \cdot \cos(\phi), \sin(\theta) \cdot \sin(\phi), \cos(\theta))$. Both $|0\rangle$ and $|1\rangle$ are placed on the Z axis at the points $(0, 0, 1)$ and $(0, 0, -1)$, respectively.

2.1.2 Measurement

Qubits have the ability to be in a superposition state, but measurement disrupts this state in a fundamental way. When a qubit is measured in the Z-basis, its state collapses into the states *either* $|0\rangle$ *or* $|1\rangle$, with a certain probability for each. The probabilities of the two states are given by the norm square of their coefficients, so according to Equation 2.1, the probability of Z-basis measuring $|0\rangle$ is $|a_0|^2$ and Z-basis measuring $|1\rangle$ is $|a_1|^2$.
For example, consider the state

$$|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \ , \tag{2.2}$$

which is a superposition consisting of an even share of $|0\rangle$ and $|1\rangle$, represented as the point $(1, 0, 0)$ on the Bloch sphere lying on the X axis. Measuring this qubit in the Z-basis will result in a 50% probability for both $|0\rangle$ and $|1\rangle$, since $|a_0|^2 = |a_1|^2 = 1/2$.

In general, the result of a series of a number N of measurements (*shots*) of a qubit in a quantum state $|\psi\rangle$ computes the probability distribution $P : \left\{|0\rangle \mapsto \frac{N_{|0\rangle}}{N}, |1\rangle \mapsto \frac{N_{|1\rangle}}{N}\right\}$, where $N_{|x\rangle}$ refers to the number of times when the state $|x\rangle$ is measured. This probability distribution converges to the probability distribution $\left\{|0\rangle \mapsto |a_0|^2, |1\rangle \mapsto |a_1|^2\right\}$ which we theoretically obtain if we measure a qubit in state $|\psi\rangle$ an infinite amount of times.

2.1.3 Unary Gates

Just as logical gates modify the state of bits in classical computing, in quantum computing quantum gates are used to alter the state of qubits. While in classical computing the only single-bit gate is the *NOT* gate, a single-qubit quantum gate can be represented as any unitary matrix that is multiplied to the qubit's state-vector to change its state. Since all valid quantum gates are unitary, the probabilities of measuring $|0\rangle$ or $|1\rangle$ always adds up to 1, i.e. $|a_0|^2 + |a_1|^2 = 1$.
The simplest example of a unary quantum gate is the *NOT* (or X) gate that always reverses the state of a qubit (e.g. $|1\rangle \to X \to |0\rangle$):

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Another example for a standard quantum gate is the Hadamard gate

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Applying the Hadamard gate to a qubit in the state $|0\rangle$ has the following result:

$$|0\rangle \rightarrow H \rightarrow |+\rangle \text{ , because } H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle .$$

In addition to standard gates that rotate the state by a fixed angle, there are also general gates that are dependent on a parameter. An example for such a gate is the $R_Z(\lambda)$ gate that rotates the state by the angle λ about the Z axis:

$$R_Z(\lambda) = \begin{pmatrix} e^{-i\frac{\lambda}{2}} & 0 \\ 0 & e^{i\frac{\lambda}{2}} \end{pmatrix}.$$

2.1.4 Binary Gates

The quantum state of multiple qubits is denoted as the tensor-product of all single-qubit states. For example, the state of a 2-qubit system where the first qubit q_0 has the state $|q_0\rangle$ and the second qubit q_1 has the state $|q_1\rangle$ is denoted as $|q_0\rangle \otimes |q_1\rangle$ or $|q_0q_1\rangle$, where $\otimes$ is the tensor-product. The two-qubit state vector is represented by a 4-dimensional complex vector of length 1:

$$|\psi\rangle = \begin{pmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{pmatrix} = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle$$

$$\wedge \; a_{00}, a_{01}, a_{10}, a_{11} \in \mathbb{C} \; \wedge \; |a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$$

where the basis states that can be observed when measuring both qubits in the Z-basis are

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

On multi-qubit systems it is now possible to apply binary gates that change the states of two qubits at the same time. Similar to unary qubit gates, binary gates are unitary matrices that keep the sum of all probabilities to 1. While in quantum computing there are also gates that operate on three or more qubits, we limit this work to binary gates. This is sufficient, since every multi-qubit gate can be expressed as a series of unary and binary gates.

Table 2.1 gives an overview of the binary gates used in this work. It shows the visual representation of how the gates are drawn in quantum circuits (Section 2.1.5) and the state transitions that each gate enforces, also shown as a matrix. Informally, the state transitions

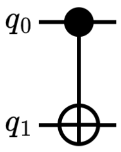
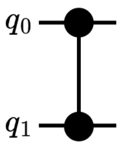
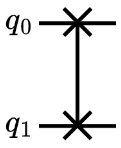
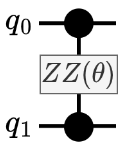
Name	Visual	State Transition $\lvert q_0q_1\rangle \mapsto \lvert q_0q_1\rangle'$	Matrix
CX (Controlled NOT)		$\lvert 00\rangle \mapsto \lvert 00\rangle$ $\lvert 01\rangle \mapsto \lvert 01\rangle$ $\lvert 10\rangle \mapsto \lvert 11\rangle$ $\lvert 11\rangle \mapsto \lvert 10\rangle$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$
CZ (Controlled Z)		$\lvert 00\rangle \mapsto \lvert 00\rangle$ $\lvert 01\rangle \mapsto \lvert 01\rangle$ $\lvert 10\rangle \mapsto \lvert 10\rangle$ $\lvert 11\rangle \mapsto$ - $\lvert 11\rangle$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{pmatrix}$
SWAP		$\lvert 00\rangle \mapsto \lvert 00\rangle$ $\lvert 01\rangle \mapsto \lvert 10\rangle$ $\lvert 10\rangle \mapsto \lvert 01\rangle$ $\lvert 11\rangle \mapsto \lvert 11\rangle$	$\begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}$
RZZ (ZZ Rotation)		$\lvert 00\rangle \mapsto e^{-i\theta/2}\lvert 00\rangle$ $\lvert 01\rangle \mapsto e^{i\theta/2}\lvert 01\rangle$ $\lvert 10\rangle \mapsto e^{i\theta/2}\lvert 10\rangle$ $\lvert 11\rangle \mapsto e^{-i\theta/2}\lvert 11\rangle$	$\begin{pmatrix}e^{-i\theta/2}&0&0&0\\0&e^{i\theta/2}&0&0\\0&0&e^{i\theta/2}&0\\0&0&0&e^{-i\theta/2}\end{pmatrix}$

Table 2.1: The four standard binary gates used in this work. The table shows the conventional visual representation of the gates, the transition of the binary state when the binary gate is applied, and the matrix representation of each binary gate.

mean that if the two qubits project to the state $|q_0q_1\rangle$ with a certain probability $p = |a_{q_0q_1}|^2$ when measured, then after applying the gate, p is the probability to measure the new state $|q_0q_1\rangle'$.

The controlled-NOT (CNOT or CX) gate is one of the most important binary gates. The gate acts on the control qubit q_0 and the target qubit q_1. When the gate is applied to the qubits, it flips the target qubit using an X gate if the control qubit is in the state of $|1\rangle$, otherwise the state does not change. The Controlled-Z (CZ) gate is similar to a CX gate, but instead of an X gate, it applies a Z gate to the target qubit when the control qubit is $|1\rangle$. The SWAP gate reverses the state of two qubits by changing the $|01\rangle$ state to $|10\rangle$ and $|10\rangle$ to $|01\rangle$. The RZZ gate rotates both qubits q_0 and q_1 by the angle θ along the Z-axis.

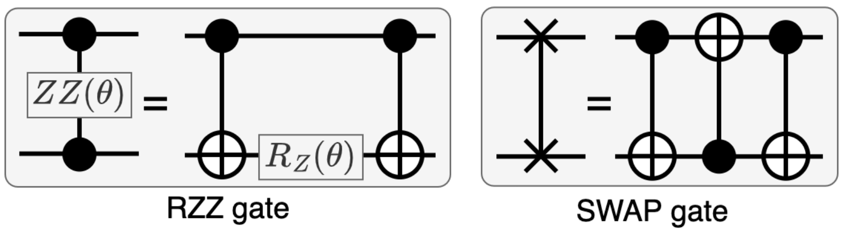


Figure 2.2: Two common gate-identities used in this work. An RZZ gate can be expressed as two CX gates with a R_Z rotation on the target qubit in between. A SWAP gate can be expressed as a sequence of three alternating CX gates.

Figure 2.2 shows two examples of expressing a binary gate as a sequence of multiple unary or binary gates. An RZZ gate is the same as a sequence of two CX gates with a R_Z gate in the middle, and a SWAP gate is a sequence of three alternating CX gates.

2.1.5 Quantum Circuits

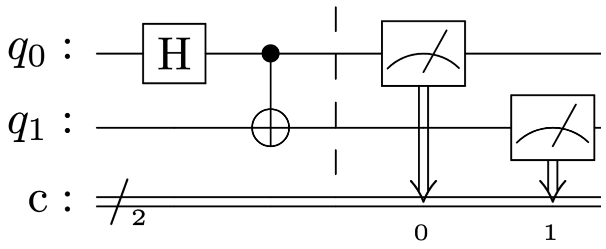


Figure 2.3: An example quantum circuit with two qubits. The time evolves from left to right: First, the Hadamard gate is applied to q_0, then a CX gate is applied to both qubits. At the end both qubits are measured.

With unary gates, binary gates, and measurements, it is now possible to build quantum circuits that execute gates and measurements for an entire quantum computation. An example circuit with two qubits is shown in Figure 2.3. Conventionally, the two qubits q_0 and q_1 are

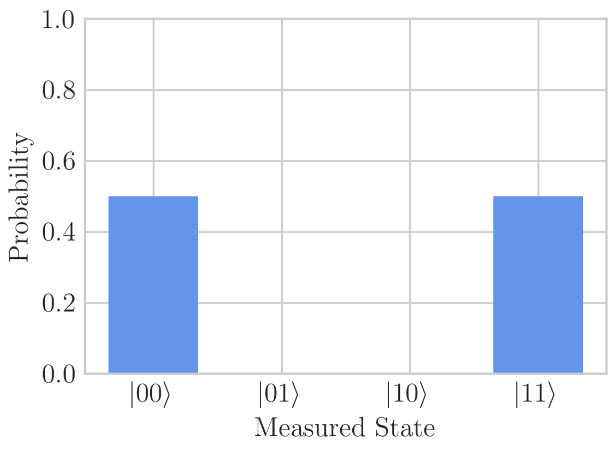


Figure 2.4: The probability distribution P on all possible states that can be measured when running the circuit of Figure 2.3 $N = 8192$ times on a noise-free quantum simulator.

initialized with the state $|0\rangle$ and the gates of the circuit are applied to the qubits from left to right: First, a Hadamard gate is applied to qubit q_0, meaning that the qubit is now in a $|+\rangle$ superposition state (Equation 2.2), which means that the state of q_0 has a 50% chance of collapsing to $|0\rangle$ or $|1\rangle$ when measured. Second, a CX gate is applied to the control qubit q_0 and the target qubit q_1. Since q_0 has a 50% chance of being in the $|1\rangle$ state before the CX gate is executed, q_1 has a 50% chance of flipping from $|0\rangle$ to $|1\rangle$ after the CX gate. In the end, all qubits are measured and each result of either $|0\rangle$ or $|1\rangle$ is written to the classical register c. Thus, after one execution, the classical register will contain either the state $|00\rangle$ or $|11\rangle$, both with probability 50%.

To find the probabilities for each possible state in a given circuit, the circuit must be run N times, where $N \gg |S|$ and S is the sample space of all possible outcomes; in our example $S = \{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. Every execution on a quantum device is called a *shot*. After executing the circuit with N shots, the probability of measuring the state $|x\rangle \in S$ is $n_{|x\rangle}/N$, where $n_{|x\rangle}$ is the number of samples where the state $|x\rangle$ is measured. This finally gives us the discrete probability distribution $P : S \to \{p \in \mathbb{R} \mid 0 \leq p \leq 1\}$ as the result of a quantum computation that can be used in quantum algorithms. Figure 2.4 shows the probability distribution we obtain when we run the example circuit with $N = 8192$ shots on a perfect, noise-free quantum simulator.

2.1.6 Physical Qubits and Noisy Quantum Hardware

Quantum computing has already been realized on various different platforms, e.g. with ultracold ion traps or superconducting hardware [13, 14]. These platforms are all considered as noisy intermediate-scale quantum (NISQ) devices since the qubits and quantum gates are still very noisy and therefore limit the size of the quantum circuits that can be run on these devices [5].

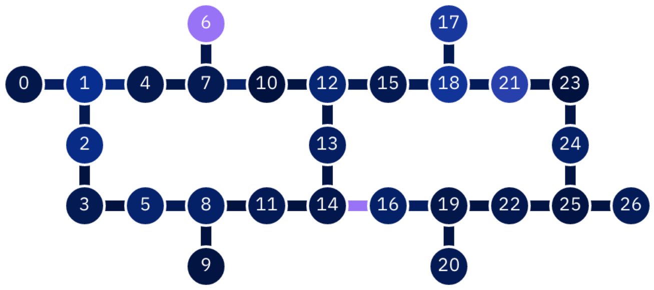


Figure 2.5: The physical qubit layout graph of the 27-qubit IBM Falcon r4 processor IBM Montreal. The darker the qubits or qubit connections, the less noisy are qubit measurements or binary gates between qubits.

This work focuses on the IBM superconducting qubit-based quantum computers that are accessible via the IBM Quantum Experience [2]. In the following, we briefly summarize the working principle of a superconducting quantum computer, as explained in [15].

Superconducting qubits are built out of so-called Josephson junctions and are arranged on a planar chip that has to be cooled down to approximately 10 mK. On this chip, the physical qubits are connected such that binary gates can be realized. The specific connectivity between the superconducting qubits leads to the characteristic qubit layout graph (Figure 2.5) of a quantum computer that indicates between which qubits the binary gates are implemented on the hardware level. Subsequently, binary gates can only be executed between two physical qubits that are connected on the hardware level.

A specific quantum circuit is processed in two main steps before it can be run on a specific quantum device. First, the quantum circuit is transpiled (Section 2.1.8), and secondly, the gates are translated to microwave pulses that directly control the qubits and implement the quantum circuit on the hardware level.

Superconducting hardware is considered as particularly promising because of comparatively fast gate times. This is crucial since states on physical qubits only have a finite life time of approximately $100 - 200\mu s$, before they loose the capability to be in a superposition state. The devices that we use in this work have error rates of approximately 10^{-4} for unary, 10^{-3} for binary gates and 10^{-2} for measurements, so every additional gate or measurement applied increases the error on the final result of a circuit execution. These exact errors vary for each physical qubit on a quantum device (Figure 2.5).

The noise of qubits and gates subsequently also effects the resulting probability distribution P of executing a circuit on a real quantum device, as shown in Figure 2.6. Here we execute our example circuit of Figure 2.3 again with 8192 shots, however, this time on the quantum device IBM Perth. In the result of the execution on this NISQ device, we can see that in some of the shots, the states $|01\rangle$ and $|10\rangle$ are measured, giving them a positive probability in the noisy probability distribution P_E of Figure 2.6. This is due to the noise of the gates or of the final measurements which produces a wrong output state in some occasions. Therefore, the noisy probability distribution P_E differs from the desired, perfect probability distribution P_S which can be retrieved from classical simulation. How much P_E differs from P_S is measured

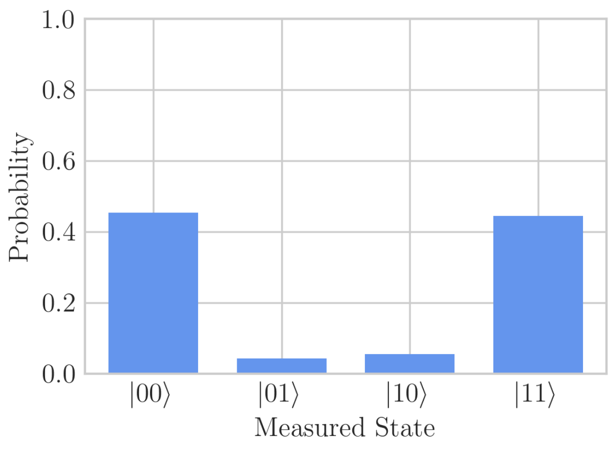


Figure 2.6: The noisy probability distribution P_E on all possible states that can be measured when running the circuit of Figure 2.3 with $N = 8192$ shots on the NISQ quantum device IBM Perth.

with the Hellinger fidelity, as defined in the next section.

2.1.7 Hellinger Fidelity

Running a quantum circuit on noisy quantum hardware produces a noisy probability distribution P_E (Figure 2.6). This noisy result is different from the desired perfect probability distribution P_S, which can be obtained by simulation with classical computation (Figure 2.4). As a measure of how noisy a probability is, we use the Hellinger fidelity, which is defined as follows.

Definition 2.1.1 (Hellinger Fidelity). *Given the perfect probability distribution P_S and the noisy probability distribution P_E of a circuit that both produces measurement results in sample space S. Then the Hellinger distance [16] $H(P_S, P_E)$ between the two distributions is defined as*

$$H(P_S, P_E) = \frac{1}{\sqrt{2}} \sqrt{\sum_{|x\rangle \in S} \left(\sqrt{P_S(|x\rangle)} - \sqrt{P_E(|x\rangle)} \right)}.$$

The Hellinger fidelity $F(P_E)$ of a noisy probability distribution is derived from the Hellinger distance [17] by using the perfect probability distribution as a comparison, and is defined as

$$F(P_E) = \left(1 - H(P_S, P_E)^2\right)^2.$$

The Hellinger fidelity $F(P_E)$ is a value in the range $[0, 1]$. The larger $F(P_E)$, the smaller the Hellinger distance from the perfect probability distribution and the less noisy the result. The value $F(P_E)$ can thus be used as a percentage indicator of how well a result of an execution on a quantum device resembles the desired, noise-free result, where 100% would mean that the result is perfect.

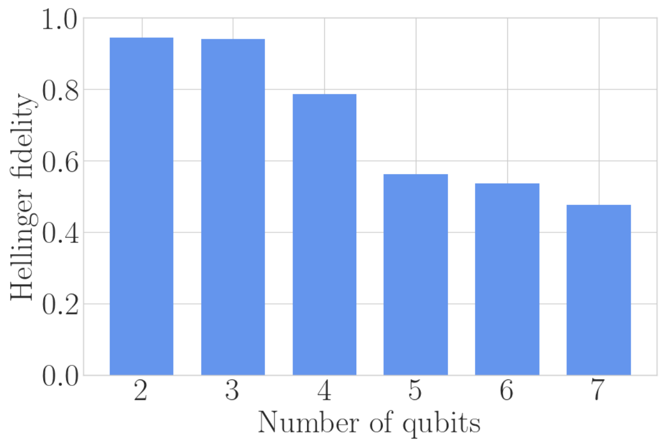


Figure 2.7: Hellinger fidelity (higher is better) of running QAOA-circuits [18] with different numbers of qubits on the 7-qubit IBM Falcon r5.11H processor IBM Jakarta.

How the Hellinger fidelity behaves when executing circuits with different sizes on a real quantum device can be seen in the graph in Figure 2.7. Here, a series of Quantum Approximate Optimization Algorithm (QAOA)[1] circuits (from the benchmark suite SupermarQ [18]) are executed on the 7-qubit quantum device IBM Jakarta with different sizes, ranging from 2 to 7 qubits. For each execution, the Hellinger fidelity is measured. The graph shows that executing circuits of smaller size produces a high-quality result with a fidelity of over 90%, but as the size of the circuit increases near the size of the device, the quality drops significantly to a fidelity of less than 50%.

2.1.8 Transpilation and Circuit-to-Hardware Mapping

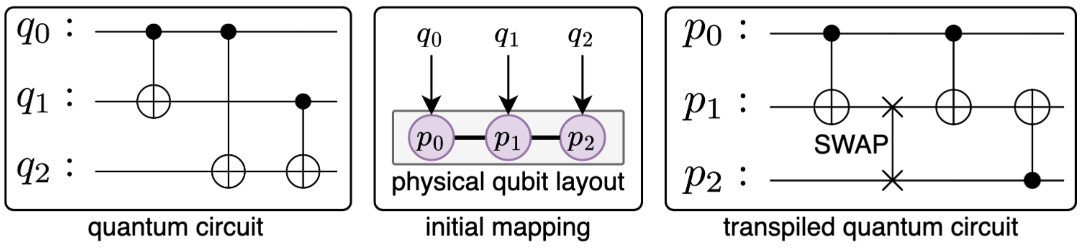


Figure 2.8: Transpiling a quantum circuit for a hardware-specific physical qubit layout.

[1]QAOA [19] is an algorithm to solve combinatorial optimization problems.

A transpiler is responsible for transforming a quantum circuit into an optimized representation that is directly executable on a given quantum device [20]. Among other optimization steps, this transformation includes translating all gates into the native gate set that the quantum device supports, as well as optimally mapping the logical qubits q_i onto the physical qubits p_i. After the logical-to-physical-qubit-mapping is decided, the binary gates have to be implemented using the restricted physical qubit connections that the quantum device has (Figure 2.5). However, this restriction often implies that binary gates can not be implemented on the hardware without *swapping* the logical qubits between physical qubits at runtime.

This is illustrated in more detail at the example in Figure 2.8 which shows the transpilation of a 3-qubit circuit where every qubit is connected via a CNOT gate with every other qubit. However, note that in the physical qubit layout of the quantum device used for executing the circuit, there is no physical connection between the physical qubits p_0 and p_2. In this example, we now map the logical qubits q_i to the physical qubits with $q_0 \mapsto p_0$, $q_1 \mapsto p_1$ and $q_2 \mapsto p_2$ for initializing the circuit execution. In the execution, the first CNOT gate ($q_0 \leftrightarrow q_1$) can be executed between the physical qubits p_0 and p_1. Conversely, the CNOT gate ($q_0 \leftrightarrow q_2$) cannot be executed with the current mapping, since the qubits q_0 and q_2 do not have a physical connection. Hence, we have to apply a SWAP gate between p_1 and p_2, which effectively assigns $q_1 \mapsto p_2$ and $q_2 \mapsto p_1$. With this new logical-to-physical-qubit-mapping, the two remaining CNOT gates can be executed on the physical qubit connections.

To implement a SWAP gate on the quantum hardware, a total of 3 CNOT gates have to be executed between the two physical qubits (Figure 2.2). Since every additional CNOT gate imposes additional errors in the circuit execution, it is imperative to minimize the number of SWAP gates. This can be done either by designing quantum circuits that fit well onto the given quantum hardware [21], or if such a design is impossible, by finding the optimal initial mapping and SWAP strategies for a circuit that does not fit on the given physical qubit layout. However, finding such optimal mappings can be a very hard and computationally expensive problem to solve [20].

2.2 Gate Virtualization

The term gate virtualization refers to a subfield of the recently introduced field of quantum circuit *cutting and knitting*, a set of divide-and-conquer techniques that generally allow certain parts of quantum circuits to be cut or virtualized to decompose the circuit into smaller subcircuits that can be executed more easily and with less noise on the current NISQ hardware. After executing the subcircuits, a classical algorithm merges the partial results to obtain the same result as if the original circuit had been executed on a large and less noisy quantum device.

Gate virtualization allows us to virtualize a binary gate by sampling it with unary operations instead and then using a simple knitting formula to obtain the probability distribution of the original circuit [9, 10]. In this section, we describe gate virtualization as defined and proved in [9] using a virtual CZ gate as an example. Note that this virtualization technique can be applied in a similar way to other binary gates.

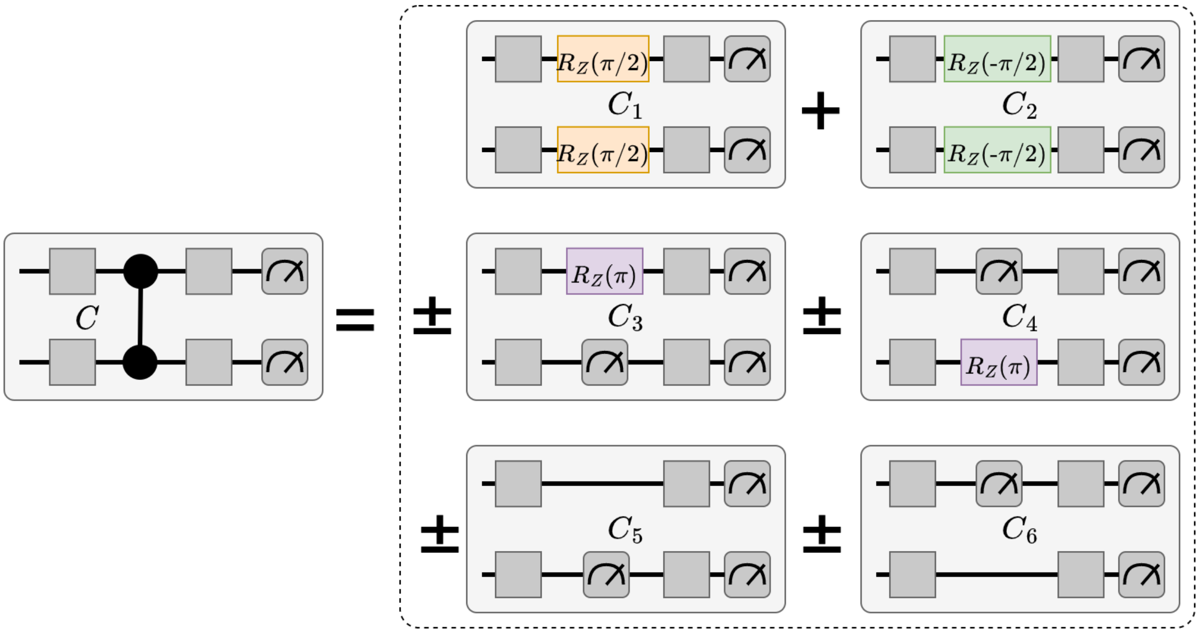


Figure 2.9: Schematic formula for virtualization of a CZ gate. Binary gates can be simulated by sampling a set of unary gates and mid-circuit measurements in the samples C_i, whose probability distributions P_i can be used to calculate the probability distribution P of the original circuit C.

2.2.1 Circuit Sampling

To virtualize a CZ gate, the circuit is first sampled with unary gates and measurements. These samplings result in a total of 6 different circuit *configurations* C_1-C_6, as depicted in Figure 2.9. The probability distributions P_1-P_6 of the corresponding configured circuit are required for the subsequent knitting process. In each of the configurations, the CZ gate is replaced with either a mid-circuit measurement or a unary gate at each of its endpoints. This means that for each configured circuit C_i, the dependency between the two qubits of the former CZ gate is removed, effectively creating two independent circuit fragments for each configuration, which can be executed independently as the circuits C_i^1 and C_i^2. The result P_i of executing the configured circuit C_i can then be obtained with the fragment results P_i^1 and P_i^2 by applying the rule of stochastically independent events [22] and multplying the probability of each outcome-state $|x\rangle$: $P_i(|x\rangle) = P_i^1(|x\rangle) \cdot P_i^2(|x\rangle)$.

2.2.2 Knitting

The knitting process computes all final probabilities $P(|x\rangle)$ of the original circuit C by applying the following formula, where $|x\rangle \in S_C$ lies in the sample space of C. The expression $P_i(|x\rangle, |m\rangle)$ with two arguments represents the probability of the state $|x\rangle$ with a mid-circuit measurement of $|m\rangle$.

$$
\begin{aligned}
p_1 &= P_1(|x\rangle) \\
p_2 &= P_2(|x\rangle) \\
p_3 &= P_3(|x\rangle, |1\rangle) - P_3(|x\rangle, |0\rangle) \\
p_4 &= P_4(|x\rangle, |1\rangle) - P_4(|x\rangle, |0\rangle) \\
p_5 &= P_5(|x\rangle, |0\rangle) - P_5(|x\rangle, |1\rangle) \\
p_6 &= P_6(|x\rangle, |0\rangle) - P_6(|x\rangle, |1\rangle)
\end{aligned}
$$

The full probability distribution for the original circuit can then be reconstructed in the classical post-processing step by summing over the outputs p_1 to p_6:

$$P(|x\rangle) = \frac{1}{2}(p_1 + p_2 + p_3 + p_4 + p_5 + p_6)$$

2.2.3 Virtualizing Multiple Binary Gates

We virtualize multiple binary gates by recursively sampling the circuit configurations and thus also recursively performing the knitting process. For example, consider the process of virtualizing two CZ gates, g_1 and g_2. To create all the required circuit configurations, we start by creating configurations C_1-C_6 for sampling the virtual gate of g_1. Then, to virtualize g_2 as well, we consider the configured circuits C_i as normal circuits and create new configurations for each 6, namely C_{11}-C_{66}, for a total of $6^2 = 36$ configurations. This process would continue for more virtual gates. Thus, to virtualize k CZ gates, we need to create and execute 6^k configured circuits for the subsequent knitting process.

The knitting process then uses the results P_{ij} of the recursively configured circuits to recursively knit the results in the opposite direction by generating each result P_i using the results P_{i1}-P_{i6}. This means that the knitting process also has an exponential overhead of 6^{k-1} individual knitting calculations for k virtual gates, causing a total recursion depth of k.

In summary, virtualization of k binary gates incurs an exponential computational cost $\mathcal{O}(6^k)$ for both quantum and classical computation. See Section 6 for a more detailed description of how DQS executes virtual gates and how the process can be parallelized to reduce the exponential runtimes for virtualizing a handful of binary gates.

3 Overview

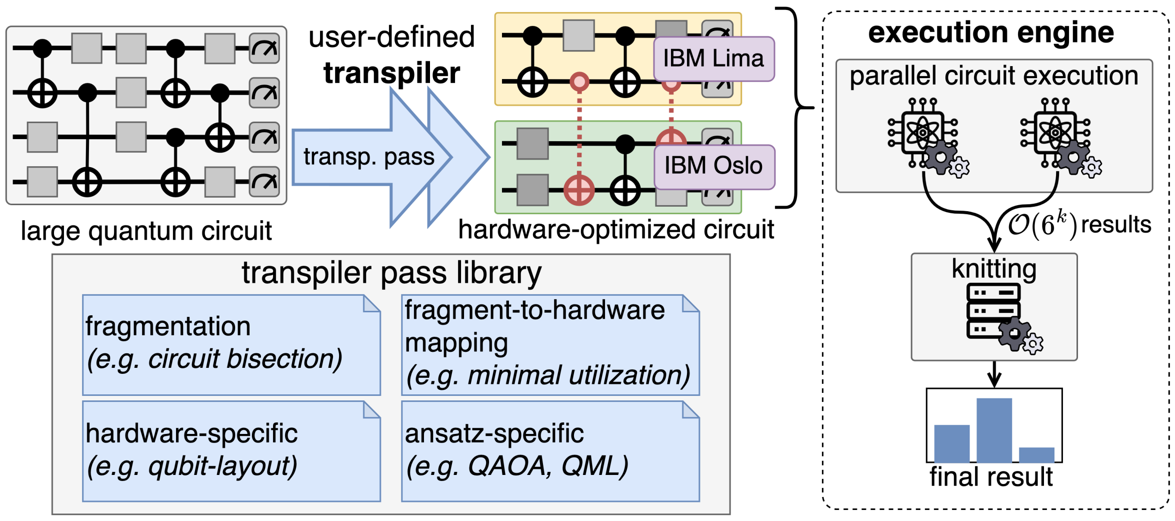


Figure 3.1: Overview of the DQS infrastructure.

For an overview of the components and workflow of the DQS system, see Figure 3.1. DQS is divided into two parts altogether: The **distributed transpiler** toolbox (left) allows users to define their custom circuit optimizations as distributed transpilers. A distributed transpiler is a sequence of transpiler passes that optimize the circuit using gate virtualization, circuit decomposition, or by assigning fragments to specific quantum devices. The application of a distributed transpiler creates the new abstraction of a fragmented, hardware-optimized **virtual circuit** running on a set of distributed quantum devices. The **execution engine** efficiently executes virtual circuits by running the fragments in parallel on the quantum devices and then composing (knitting) the result to the final result of the original circuit. In developing this system, DQS adheres to the following three design goals.

3.1 Design Goals

3.1.1 Programmabilty

To allow users and researchers to define their own circuit optimizations using gate virtualization, we need a unified API capable of automatically executing a custom transpiler. With this API, the user should have full freedom to optimize a circuit in every aspect that gate virtualization has to offer, including

- defining how binary gates are virtualized, e.g. to create approximate virtualizations (Section 5.4) or to virtualize user-defined non-standard binary gates,
- the virtualization of gates in a given circuit to reduce overall noise and allow the decomposition of the circuit into fragments,
- management of fragment decomposition and individual selection of the backends on which fragments are executed, and
- optimization of fragments for execution on each backend, using both gate virtualization and conventional optimization techniques.

By reusing concepts from state-of-the-art transpiler interfaces [20], this new transpiler API should be easy to use and allow straightforward integration of existing circuit optimization algorithms. To implement such an API, this work presents the novel abstraction of a **distributed transpiler** based on a **virtual circuit**, as explained in Section 4.

3.1.2 Transpiler Design for Optimal Gate Virtualization

The new transpiler API now allows researchers to create novel circuit optimizations. However, end users, such as quantum algorithm developers, should already have access to a collection of predefined transpiler primitives (or *passes*) that can be easily combined to create a transpiler for the user's specific needs. These primitives include

- **decomposition** transpiler passes that optimally decompose a given circuit into fragments (Section 5.1),
- **fragment-to-device mapping** transpiler passes which optimally map fragments to available quantum devices (Section 5.2), and
- **qubit layout** transpiler passes that optimize the logical-to-physical qubit mapping of individual fragments (Section 5.3).

In this work, we give a general overview of this transpiler design and show how to implement a set of passes as part of a transpiler library in Section 5. These transpiler primitives allow users to combine them for their use cases, e.g., to optimize circuits used in specific quantum algorithms.

3.1.3 Scalability and Performance in Circuit Execution

Current techniques for gate virtualization (Section 2.2) incur a computational cost of $\mathcal{O}(6^k)$ actual circuits that must be executed to virtualize k gates. Subsequently, the computational cost of constructing the final result from the $\mathcal{O}(6^k)$ execution results also grows exponentially with the number of virtual gates and the size of the circuits. To scale the number of gate virtualizations that can be executed in a reasonable amount of time, the following two techniques are used:

- **distributed and parallel computing** both in the execution of the circuits, by using multiple quantum devices simultaneously, and in the classical post-processing, by splitting the computation of the final result into a large number of sub-computations that can be executed in a distributed cluster (Section 6), and
- **approximation techniques**, which can reduce the computational cost to as little as $\mathcal{O}(2^k)$ by having little to no loss of accuracy for certain circuits. With its generic API, DQS allows empirical investigation of such approximation techniques (Section 5.4).

3.2 Programming Model

```
 1 from dqs import execute, transpile
 2 from dqs.backends import IBMQJakarta, IBMQOslo, Simulator
 3 from dqs.transpiler_library import BisectionPass, MinimalUtilizationPass
 4 
 5 # Define a set of quantum devices or simulators that DQS can use
 6 backends = {IBMQJakarta(), IBMQOslo(), Simulator()}
 7 # define a sequence of transpiler passes:
 8 # optimal decomposition pass
 9 frag_pass = BisectionPass(max_cuts=5)
10 # mapping pass that maps circuit fragments to quantum devices
11 mapper_pass = MinimalUtilizationPass(backends)
12 # a user-defined transpiler pass
13 custom_pass = CustomPass()
14 # define the custom transpiler as a sequence of the transpiler passes
15 transpiler = [frag_pass, mapper_pass, custom_pass]
16 
17 # transpile and run the transpiled circuit
18 circuit = QuantumCircuit(...)
19 transpiled_virtual_circuit = transpile(circuit, *transpiler)
20 result = execute(transpiled_virtual_circuit)
```

Listing 3.1: Example code of a typical user program with DQS.

The simplicity of programming with the DQS framework, including creating a transpiler, transpiling a circuit, and executing the resulting virtual circuit, is demonstrated with an example in Listing 3.1. Here, a sequence of three distributed transpiler passes is defined: (1) a bisection pass that decomposes a given circuit into two even fragments (line 9), (2) a mapper pass (line 11) that maps fragments to given backends (line 6) by minimizing the utilization on each backend, and (3) a pass that is defined by the user (line 13). This pass sequence is used to create the user-defined distributed transpiler (line 15). The transpiler can now be used to transpile a circuit into a virtual circuit (lines 18-19), and finally the execution engine of DQS executes the circuit to get the final noise-mitigated result.

4 The Distributed Transpiler Abstraction

In this section, we introduce the novel abstraction of distributed transpilation, which extends the scope of traditional circuit optimization by allowing gate virtualization . Before we can create this new abstraction, we also need to extend the scope of the quantum circuit object, by creating the abstraction of a virtual circuit.

4.1 Virtual Circuit

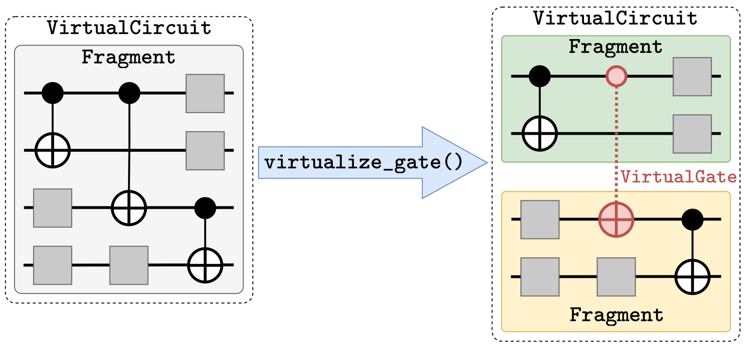


Figure 4.1: Visual representation of the abstraction of a virtual circuit.

As Figure 4.1 shows, virtual circuits can now manage virtual gates as part of the circuit, and view themselves as a collection of distinct fragments instead of one single circuit that is meant to run on a single device. To implement this, DQS proposes the three following abstractions of `VirtualGate`, `VirtualCircuit` and `Fragment`.

4.1.1 Virtual Gate

The `VirtualGate` interface (Listing 4.1) is used to implement the virtualization of a binary gate, as described in Section 2.2. Each `VirtualGate` is a special `BinaryGate` that implements two additional methods: The `configurations()` method is used to get a list of all configurations that must be executed to virtualize the gate. Each configuration is a list of unary operations that need to be inserted in place of the real gate. The `knit()` method implements a knitting formula for the virtual gate by calculating the probability of a given state. To do this, the method uses a list of results, where the result at index i is the result of the circuit execution with configuration i given by the `configuration()` method.

Since DQS provides this generic interface, users can create their own virtual gates for their optimized transpiler passes. In addition, DQS already provides the implementation of three common virtual gates in accordance to the explanation in Section 2.2: `VirtualCZ`, `VirtualCX`, and `VirtualRZZ`. For an example implementation of `VirtualCZ` (cf. Figure 2.9), see Listing 4.1, lines 12-19.

```
 1 Configuration = list[Union[UnaryGate, Measurement]] # a list of unary operations
 2 Result = dict[str, float] # probability distribution: measured state -> probability
 3 
 4 # Interface used to implement the virtualization for a specific binary gate
 5 class VirtualGate(BinaryGate):
 6     def configurations() -> list[Configuration]:
 7         # Provides all configuration that need to be executed as a replacement for the gate
 8     def knit(results: list[Result], state: str) -> float:
 9         # Knits the probability for a specific state by using the results of all configurations
10 
11 # Example implementation of the virtual CZ gate
12 class VirtualCZ(VirtualGate):
13     def configurations() -> list[Configuration]:
14         return [
15             [RZGate(qbit0, pi/2), RZGate(qbit1, pi/2)],
16             [RZGate(qbit0, -pi/2), RZGate(qbit1, -pi/2)], ...]
17 
18     def knit(results: list[Result], state: str) -> float:
19         return 1/2 * (results[0][state] + results[1][state] + ...)
```

Listing 4.1: The interface for virtual gates.

4.1.2 Virtual Circuit and Fragments

```
class Fragment:
    circuit: QuantumCircuit
    qubits: set[Qubit]

STANDARD_VIRT_GATES = {'cz': VirtualCZ, 'cx': VirtualCX, 'rzz': VirtualRZZ}

class VirtualCircuit(QuantumCircuit):
    fragments: set[Fragment]

    def virtualize_gate(real_gate: BinaryGate, virtual_gate: VirtualGate):
        # virtualize a gate by replacing it with the corresponding
        # implementation of a virtual gate
    def virtualize_connection(qubit1: Qubit, qubit2: Qubit, v_gates = STANDARD_VIRT_GATES):
        # virtualizes every binary gate between qubit1 and qubit2, using the given vgates
    def decompose():
        # decompose the circuit into the maximum amount of fragments
    def merge(frag1: Fragment, frag2: Fragment):
        # merge two fragments into one logical fragment
```

Listing 4.2: Simplified definitions of a virtual circuit and fragment.

To manage virtual gates and the decomposition of quantum circuits into multiple fragments, DQS introduces the abstraction of `VirtualCircuit` (Listing 4.2), a specification of the traditional `QuantumCircuit` object. In addition to traditional operations, a `VirtualCircuit` may also contain virtual gates. A virtual gate is inserted by replacing the real binary gate with the implementation of the corresponding virtual gate using the `virtualize_gate()` method. Often transpiler passes must remove the entire connection between two qubits using the `virtualize_connection()` method, which virtualizes each binary gate between two given qubits using the default or user-defined virtual gates.

Since virtualizing gates removes dependencies between qubits, a `VirtualCircuit` now consists of a set of `Fragments`. A fragment is a view on a qubit sub-set of a quantum circuit that has no *real* gates with the qubit set of other fragments and can therefore be executed independently. Note that virtualizing a gate does not change the state of the fragment set inside a virtual circuit. This is because a transpiler might still consider a fragment that could technically be further decomposed as a circuit, or decomposition might not be possible due to transitive dependencies between qubits. However, to manage fragmentation, the virtual circuit provides two methods, `decompose()` and `merge()`, which only change the state of the fragment set and not the state of virtual gates. The `decompose()` method decomposes the circuit into as many fragments as possible, which means that the qubits of each fragment are connected only to *real* binary gates. In contrast, the `merge()` method merges two fragments into a larger fragment that should run as a whole on a given quantum device.

4.1.3 Connectivity Graph

An important part of distributed transpilation is to obtain information about how the qubits of a virtual circuit are connected. For this purpose, we can represent a circuit as a connectivity graph, as shown in Figure 4.2. The connectivity graph is an undirected graph $G_C = (V, E)$ consisting of qubits as vertices $q_i \in N$ and edges $e_j(w) \in E$ representing the connection between two qubits in the circuit. Each edge $e_j(w)$ has a weight w, which represents the number of binary gates between the two qubits.

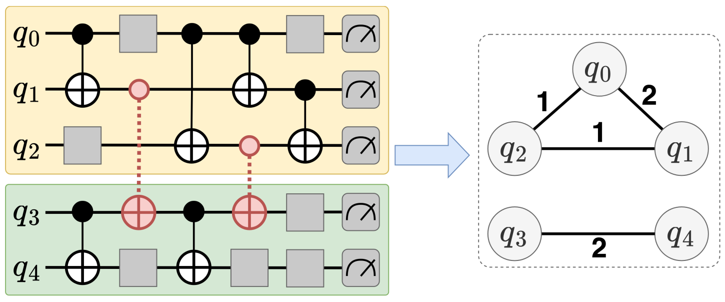


Figure 4.2: Example of representing a virtual circuit as a qubit connectivity graph.

4.2 Distributed Transpilation

```python
class TranspiledFragment(Fragment):
    backend: Backend
    transpiled_circuit: QuantumCircuit

class TranspilerPass:
    def run(virtual_circuit: VirtualCircuit) -> VirtualCircuit:
        # method that is called to optimize the circuit with the pass

def transpile(vc: VirtualCircuit, *transpilers: TranspilerPass) -> VirtualCircuit:
    for transp in transpilers:
        vc = trans_pass.run(vc)
    return vc
```

Listing 4.3: The distributed transpiler abstraction. A transpiler is a sequence of transpiler passes, each optimizing the circuit in their `run()` methods.

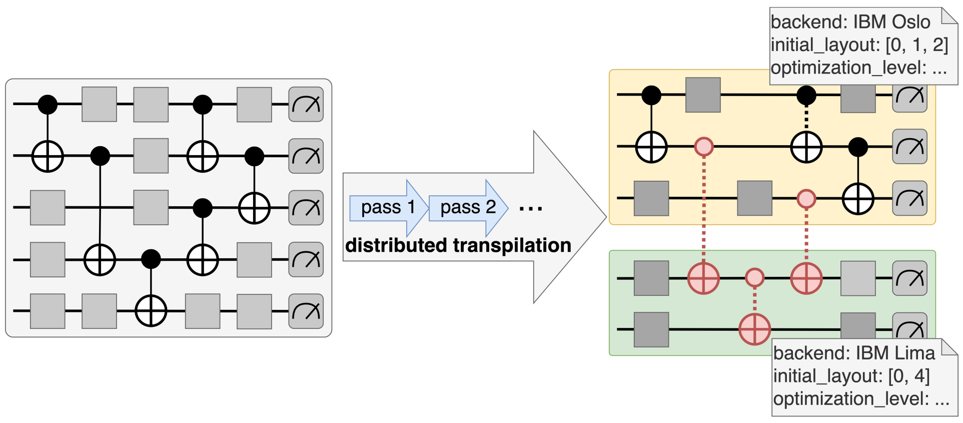


Figure 4.3: The design of distributed transpilation. A virtual circuit goes through a number of transpilation passes to fully optimize the circuit and its circuit fragments.

With the new virtual circuit abstraction, we can now design the generic interface for distributed transpilers (Figure 4.3), which enables the definition of a circuit optimization with the new capabilities of gate virtualization. Similar to traditional transpilers, a distributed transpiler is a chain of transpiler passes, each of which optimizes the virtual circuit in a particular way. Each pass has to implement the `run()` method, which receives the current virtual circuit as an argument and returns the further optimized virtual circuit. During transpilation, each run can generally perform one or more of the following optimization steps:

1. Virtualize gates by calling `virtualize_gate()` or `virtualize_connection()` on the circuit to allow further optimization steps. For this, the transpiler pass can use the standard or customized virtual gates.

2. Decompose the circuit into fragments and decide which distinct qubit-group is actually considered as a fragment that runs on a single quantum device by calling `decompose()`

and `merge()`. E.g., the green fragment in Figure 4.3 is considered as one fragment after transpilation although it could technically be decomposed further.

3. Map fragments to specific quantum backends that should be used for the execution.
4. Optimize the fragments to run on the respective quantum device by using traditional transpilation.

To perform steps 3 and 4, the fragments are replaced by the special `TranspiledFragment` object (Listing 4.3), which contains the backend and the already transpiled circuit fragment in addition to the normal fragment object. Note that while this individual fragment mapping and transpilation is very useful for distributed transpilers trying to virtualize binary gates to optimize them for specific quantum devices, these steps are completely optional for other transpilers. If a distributed transpiler does not fully transpile the fragments for specific backends, DQS automatically maps and transpiles the fragments using a just-in-time transpiler before execution, as described in Section 6.1.

This new design of distributed transpilation now allows users to define their own optimization techniques by taking full advantage of gate virtualization. In doing so, the abstraction does not stray far from the already familiar abstractions of single circuit transpilers and is also compatible with single circuit transpilers applied to specific fragments.

5 Design of Distributed Transpilers

We notice that a distributed transpilation pass can generally optimize a virtual circuit along three main axes. In the next sections, we introduce these three categories of transpilation passes. We then present an approximate virtualization technique that could reduce the computational cost of virtualizing gates by making a tradeoff with the overall noise mitigation.

5.1 Circuit Decomposition

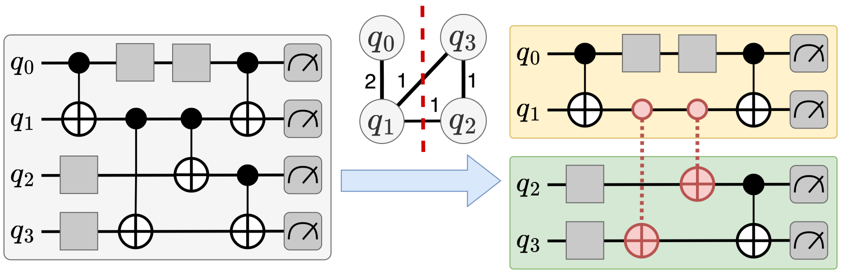


Figure 5.1: Example of optimizing a virtual circuit for decomposition by optimally dissecting its connectivity graph.

A circuit decomposition pass aims to virtualize gates to make a circuit decomposable into smaller fragments (Figure 5.1). In general, the pass should aim to decompose the circuit into 2 or more fragments of roughly equal size, with the least amount of binary gates virtualized. To this end, it is often useful to apply a node community detection algorithm, such as the Kernighan-Lin bisection [23], to the connectivity graph of the circuit. In this section, we present two implementations of such transpiler passes.

5.1.1 Bisection

The bisection pass is used to decompose an arbitrary circuit into two even fragments by virtualizing as few binary gates as possible. The algorithm of the pass runs as follows:

1. Apply the Kernighan-Lin bisection algorithm [23] to the connectivity graph $G_C = (V, E)$ (Section 4.1.3) of the virtual circuit. The algorithm returns two distinct vertex sets A and B such that the sum of the weights of all edges between A and B is minimal.

2. Determine all edges $E_{AB} = \{(q_i, q_j) \mid q_i \in A \wedge q_j \in B\}$ between the two vertex-sets.

3. Call `virtualize_connection(`q_i`,`q_j`)` on the virtual circuit for every $(q_i, q_j) \in E_{AB}$ to virtualize the connection between the two vertex-sets
4. Finally call `decompose()` to effectively create two distinct circuit fragments.

See Figure 5.1 for an example of applying this pass on a virtual circuit.

5.1.2 Ladder Decomposition

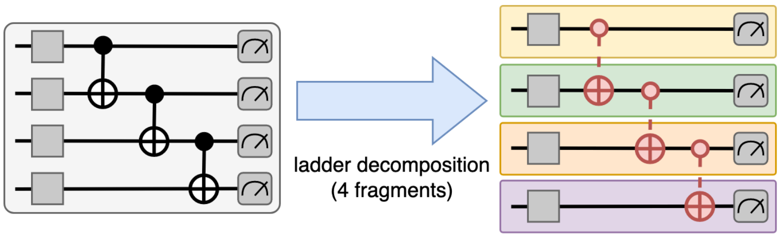


Figure 5.2: A ladder decomposition pass dissects a circuit consisting of binary gate ladders into multiple fragments.

In many circuits used in popular quantum algorithms we can observe one or more layers of so called ladders of binary gates going from the top two qubits to the bottom two qubits, so that every qubit is connected only to its nearest neighbor (Figure 5.2). When we know that a given circuit consists of ladders, the ladder deposition pass optimally decomposes a circuit with q qubits into n equal-sized fragments by simply calling `virtualize_connection(`q_i`,`q_j`)` on every $(q_i, q_j) \in \left\{(k \cdot s - 1,\ k \cdot s) \mid k \in \{1, ..., min(n, q)\} \wedge\ s = max\left(1, \lfloor \frac{q}{n} \rfloor\right)\right\}$.

5.2 Fragment-to-Device Mapping

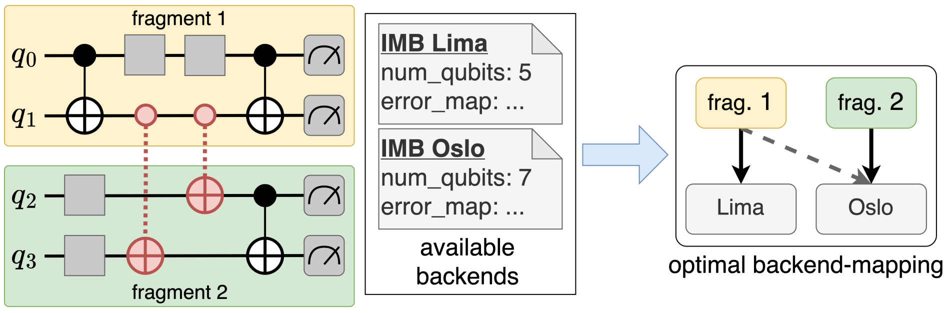


Figure 5.3: Mapping fragments to backends based on the backend properties.

With a virtual circuit decomposed into multiple fragments, a fragment-to-device mapping pass can now decide on which backends the fragments should be executed on by creating the function $m : F \to B$ from the set of fragments $F = \{f_1, ..., f_n\}$ to the set of available backends $B = \{b_1, ..., b_m\}$. The decisions that such a pass takes are based on the properties of the available backends, e.g. the number of physical qubits or the number of qubits that have readout errors below a certain threshold. The available backends B are passed to a fragment-to-devices mapping pass on initialization.

Given two fragments $\{f_1, f_2\} \subseteq F$ and two backends $\{b_1, b_2\} \subseteq B$ a fragment-to-device mapping pass generally has three options to create a mapping (see Figure 5.3):

- Map the fragments to distinct backends, e.g. $f_1 \mapsto b_1,\ f_2 \mapsto b_2$. The fragments are therefore executed entirely independent from each other, enabling parallelism in the circuit execution.
- Map the fragments to the same backend, e.g. $f_1 \mapsto b_1,\ f_2 \mapsto b_1$. This incurs that the fragments have to be executed serially.
- Merge fragments by calling the `merge()` method on the virtual circuit, so that the former fragments are viewed as one fragment $f_3 = f_1 \cup f_2$, to be executed as one circuit. Then, map the resulting fragment to a backend, e.g. $f_3 \mapsto b_1$. Every virtual gate in the new fragment is still virtualized, as `merge()` does not change the state of gates.

This pairwise mapping is easily extensible to create the entire mapping from fragments to backends, as we can see in the following three implementations of fragment-to-device mappings, all of which have linear time complexity with respect to the number of fragments and backends. Note that we assume that for any mapping $f_x \mapsto b_y$ decided by a pass, the constraint $|f_x| \leq |b_y|$ must hold ($|f_x|$ or $|b_y|$ denotes the number of qubits in the fragment or backend, respectively) such that the fragment can run on its assigned backend. If this condition is not met, a pass cannot be executed.

5.2.1 Minimal Utilization

The minimal utilization pass allocates fragments to backends by aiming to minimize the total percentage of utilized qubits per device while also parallelizing fragment execution. Minimizing utilization generally improves the ability to optimize fragments for each backend by selecting physical qubits with minimal noise, while keeping overall runtime low due to parallel execution. However, the drawback of this pass is that it occupies many underutilized devices that cannot be used for other computations. The pass runs as follows:

1. Dissect the virtual circuit into the maximum amount of fragments by calling `decompose()`.
2. Store both the fragments and the available backends into the sequences $F_S = (f_1, ..., f_n)$ and $B_S = (b_1, ..., b_m)$, both sorted in *descending* order by the number of qubits.
3. *Zip* the two sequences by mapping $f[i] \mapsto b[i]$ for all $i \in \{0, ..., m-1\}$.
4. If the number of fragments is larger than the number of backends ($n - m$ fragments are not mapped yet), set $F_S = (f_{n-m}, ..., f_n)$ and go to step 2.

5.2.2 Maximal Utilization

The maximum utilization pass aims to fit as many fragments as possible on a minimum number of devices, and is thus the counterpart of the minimum utilization pass. The problem this pass solves is basically a bin packing problem [24], where the bins are the backends and the items are the fragments. Both the capacity of the bins and the weight of the items correspond to the number of qubits in the backends and fragments, respectively.

We implement the following greedy approximation to the bin packing algorithm to solve the problem of the maximum utilization pass:

1. Store both the fragments and the available backends into the sequences $F_S = (f_1, ..., f_n)$ and $B_S = (b_1, ..., b_m)$, both sorted in *descending* order by the number of qubits.
2. Iterate through B_S, and for each backend $b[i]$:
    a) Set $f_{b[i]} = f[0]$ as a fragment that maps to $b[i]$ ($f_{b[i]} \mapsto b[i]$). Remove $f[0]$ from F_S.
    b) Iterate through each $f_x \in F_S$ in the descending order:
    If f_x still fits onto $b[i]$ with
    $$|b[i]| \geq |f_x| + \sum_{f_y \mapsto b[i]} |f_y|$$
    set $f_{b[i]} = f_{b[i]} \cup f_x$ (by calling `merge(`$f_{b[i]}$`, `f_x`)`) and remove f_x from F_S.
    c) If F_S is empty, return the complete mapping. If not, repeat step 2.

By assigning fragments to devices in this way, only a minimal number of backends need to be used, while still running the fragments in parallel as they are combined into larger fragments that maximize the utilization of the backends. This high utilization may seem counterintuitive at first, since a circuit that heavily utilizes the device generally generates noise (Figure 2.7). Nevertheless, execution can benefit from the virtual gates are in the merged fragments (Section 5.3).

5.2.3 Minimal Noise

The minimal noise pass aims to find the fragment-to-backend mapping that expects to have the least noise overall, while taking the disadvantage of a less parallelizable fragment execution. It runs at follows:

1. Dissect the virtual circuit into the maximum amount of fragments by calling `decompose()`.
2. Independently map each fragment $f_i \in F$ to the backend $b_i \in B$ that has the least expected noise when optimally mapping the logical qubits onto the physical qubits.

To execute Step 2, we use the mapomatic library [25] for the IBM quantum devices, which gives noise-scores (lower is better) to each available backend, given a circuit to run. This score factors in both the measurement error of each physical qubit and the CNOT-gate error between physical qubits.

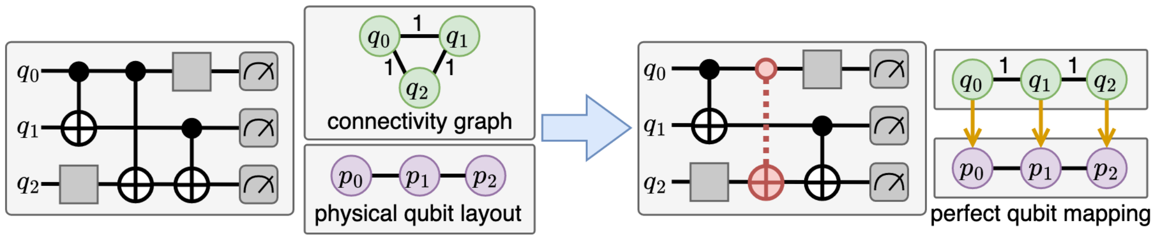


Figure 5.4: Using a qubit layout pass to allow a perfect mapping from logical qubits to physical qubits for a specific circuit or fragment.

5.3 Qubit Layout

Traditional transpilers aim to optimally map the logical qubits q_i of a circuit to the physical qubits p_j of a quantum device, i.e., with the least possible number of SWAP gates, as we described in Section 2.1.8. Mapping without SWAP gates is only possible if the circuit connectivity graph G_C is a subgraph of the physical qubit layout graph G_P. If the circuit's connectivity does not allow mapping without SWAP gates, a qubit layout pass can help bring a virtual circuit to a state where fewer or no SWAP gates are needed to run it on a given quantum device.

The idea behind qubit layout passes is illustrated in Figure 5.4, where such a pass is applied to a circuit similar to the example in Figure 2.8. Since each qubit is connected to every other qubit in the original circuit, each initial mapping $q_i \mapsto p_j$ requires at least one SWAP gate to execute the circuit on the physical qubit layout. However, if we now virtualize the connection between two qubits (here $q_0 \nleftrightarrow q_2$), we no longer need a SWAP gate because G_C is now a subgraph of G_P after removing the edge. Subsequently, we now have a perfect mapping from logical to physical qubits, since each logical qubit can be mapped to a fixed physical qubit for the entire runtime of the circuit.

Note that while perfect qubit mapping is definitely the desired outcome of a qubit layout pass, it may not be feasible given the computational overhead of virtual gates and the complexity such a pass introduces for larger circuits and devices. Below, we present two qubit layout passes that aim to heuristically minimize the number of SWAP gates required to execute a circuit on a given device.

5.3.1 Reduce Qubit Connections

The reduce qubit connections pass is a simple heuristic algorithm to reduce the average degree of qubit connections in a given circuit. This is particularly useful given the average degree d_P of physical qubit connections of a quantum device, e.g., in IBM's superconducting devices, most physical qubits are connected only to their nearest neighbor (Figure 2.5) and thus have an average degree of $d_P \approx 2$. The reduce qubit connections pass, which reduces the degree of each vertex to a maximum of d_P, proceeds as follows:

Iterate through every vertex $q_i \in E$ of the circuit's connectivity graph $G_C = (V_C, E_C)$. If

$deg(q_i) > d_P$, execute the following loop until $deg(q_i) = d_P$:

1. Determine the neighbor vertex $q_n = max_{deg}(\mathcal{N}(q_i))$, that has the maximum degree of all neighbor vertices of q_i. If multiple neighbor vertices have the same maximum degree, choose any of these vertices.
2. Virtualize the connection between q_i and q_n by calling `virtualize_connection(`q_i`, `q_n`)` on the virtual circuit, effectively removing the edge (q_i, q_n) in the connectivity graph and decrementing the degree of both vertices.

5.3.2 Minimal SWAP Gates

A minimal SWAP gates pass inspects and improves an already transpiled circuit, by effectively removing the need to apply SWAP gates to implement a binary gate between two logical qubits $q_i \in V_C$ of the connectivity graph $G_C = (V_C, E_C)$. The pass removes the logical connection between two qubits that has been mapped to non-neighboring physical qubits $p_k, p_l \in V_P$ of the physical qubit layout graph $G_P = (V_P, E_P)$. For this, it considers only the qubits p_k, p_l that have a distance $dist_{G_P}(p_k, p_l) \geq d$, where d is a user-defined parameter larger than 1.

1. Transpile the circuit for the device by using a state-of-the-art transpiler which creates an optimal initial qubit mapping $m_q : V_C \rightarrow V_P$.
2. For every edge $(q_i, q_j) \in E_C$, apply the mapping m_q to see how the connection is implemented on the hardware: $(q_i, q_j) \mapsto (p_k, q_l)$.
3. If $dist_{G_P}(p_k, p_l) \geq d$, call `virtualize_connection(`$q_i$`, `$q_j$`)` to remove the connection between the two qubits, so that no SWAP gates have to be applied for this connection.

5.4 Approximate Virtualization

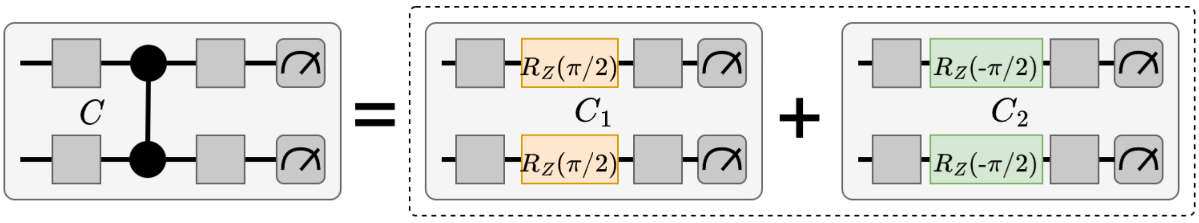


Figure 5.5: Schematic formula used to *approximately* virtualize a CZ-gate.

As we discussed in Section 2.2.3, virtualizing binary gates incurs a computational cost of $\mathcal{O}(6^k)$ for k virtual gates. To reduce this overhead, we argue that for specific circuits it might be unnecessary to execute all 6 configurations of the virtualization technique, and it is often a good approximation to just execute the two first configurations of the technique, as depicted in Figure 5.5. The approximate knitting formula to get the final probability distribution P (cf. Section 1.1) is then simply

$$P(|x\rangle) = \frac{1}{2}(P_1(|x\rangle) + P_2(|x\rangle)).$$

Subsequently, we now mitigate the computational overhead to just $\mathcal{O}(2^k)$, while loosing some accuracy in the result, depending on the properties of the circuit. Using such approximation, and not actually running the *accurate* virtualization of [9] has also already been discussed in prior works [26].

DQS implements the approximation techniques for the three standard binary gates (cf. Section 4.1.1), namely `ApproxVirtualCX`, `ApproxVirtualCZ` and `ApproxVirtualRZZ`. While these approximation techniques certainly need more investigation, we already apply them on commonly used circuits in our evaluation (Section 7), and see that for specific circuits there is almost none or very little loss in fidelity. Our approximation technique can especially be beneficial, since no circuit configuration is using any additional mid-circuit measurements, which in general can add large amounts of noise, because of the relatively large errors that measurements incur.

6 Execution Engine

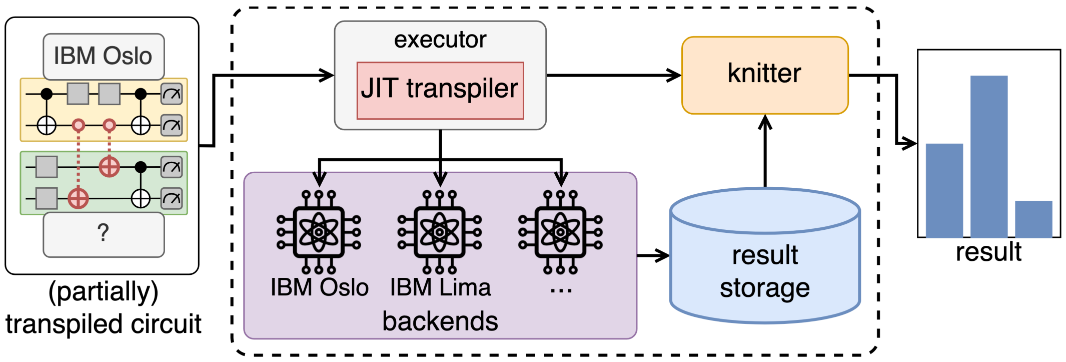


Figure 6.1: The execution engine of DQS.

In this chapter, we present the design of our execution engine, which is used to efficiently execute the new abstraction of a virtual circuit. The components and workflow of the execution engine of DQS are shown in Figure 6.1. To execute a transpiled virtual circuit, the circuit is passed to the executor. If fragments of the circuit are not fully transpiled for a particular backend, the executor fully transpiles the circuit using a just-in-time transpiler that assigns a backend to each fragment and transpiles the fragment for the backend. The executor then executes the fragments in all their configurations on the backends, which store their results in the result storage. Finally, when the execution is complete, the knitter retrieves all the results and merges them to compute the final result.

6.1 Just-in-Time Transpiler

A just-in-time (JIT) transpiler is used whenever a virtual circuit is not yet fully transpiled, i.e., when one or more fragments have not yet been assigned a backend. This is the case when a user has transpiled the circuit so that it is hardware independent for some fragments. Therefore, any JIT transpiler is a distributed transpiler with the additional constraint of creating a complete mapping between fragments and backends. If this constraint is not met, the scheduler returns an error. While users can define any transpiler that satisfies this condition as a JIT transpiler when deploying the execution engine, it is recommended to use a JIT transpiler that is time-efficient and simply allocates and transpiles fragments for available backends without performing further, time-consuming circuit optimizations, including gate virtualizations. For example, a JIT transpiler could include a full fragment-to-device mapping pass that guarantees that each fragment is mapped to one of the available backends (Section

5.2). It is particularly useful here that the JIT transpiler has a real-time view of the available backends and can therefore make the fragment-to-device mapping decision based on the current noise model or the current availability and expected queue times [27] of the backends.

6.2 Executor

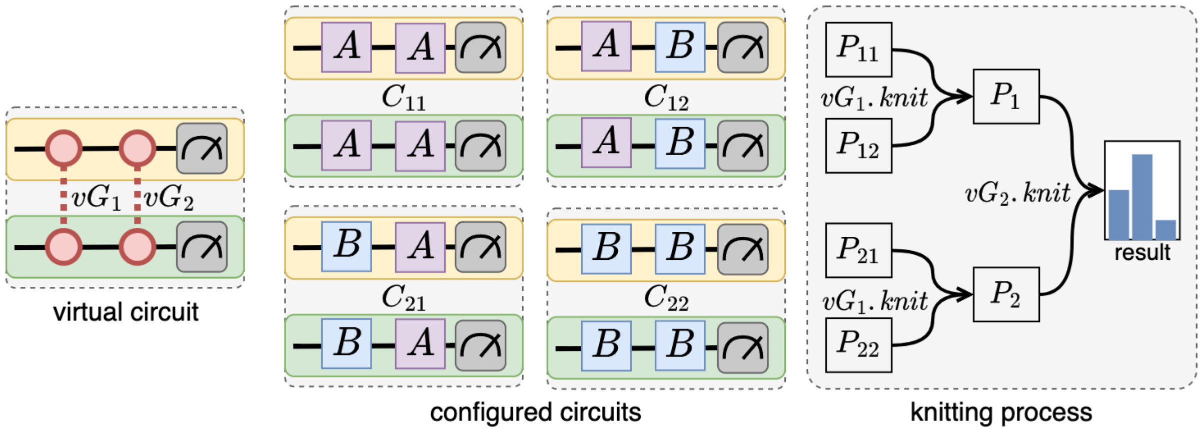


Figure 6.2: Example of creating all circuit fragments to execute and to knit the results of all configurations. Here, two gates (equal) are virtualized configuring the circuit with A gates as the first configuration and B gates as the second configuration on both qubits (similar to an approximate virtual gate, Figure 5.5).

```python
class Executor:
    JIT_transpiler: DistributedTranspiler

    def execute(virtual_circuit: VirtualCircuit):
        # transpile the circuit fully with the JIT transpiler if necessary
        if fragments_without_backend(virtual_circuit):
            virtual_circuit = JIT_transpiler.run(virtual_circuit)
        # abort the execution if the JIT tranpiler did not fully map the fragments
        if fragments_without_backend(virtual_circuit):
            raise Exception()

        # create backend jobs for all circuits required to execute
        jobs: dict[Backend, list[QuantumCircuit]] = {}
        for fragment in circuit:
            circuits_to_execute = fragment.configured_circuits()
            jobs[fragment.backend] += circuits_to_execute

        # run all jobs and store the result in the result storage
        execute_in_parallel(jobs)
```

Listing 6.1: Algorithm for the DQS executor

DQS proposes a simple algorithm for executing all the required circuits for virtualizing each virtual gate (Listing 6.1): First, the executor uses its JIT transpiler to fully transpile the virtual circuit if it is not yet fully transpiled, and throws an exception if at least one fragment is not yet mapped to a backend (lines 6-10). Next, the jobs to be executed are created that contain the configured, non-virtual circuits. To do this, the executor recursively calls the `configurations()` method of the virtual gates to apply all possible combinations of configurations to the fragments, as exemplified in Figure 6.2. Each fragment running on the same backend is combined into one job, which is executed one right after the other (lines 13-16). Finally, all jobs are executed in parallel and the results are stored in the result storage.

6.3 Result Storage

The result storage is a key-value store used to share all execution results between system components, in particular to pass the results of the execution of all fragments from the backends to the knitter. The key pointing to each result consists of three components: (1) a unique circuit-id for the virtual circuit, (2) the index of the fragment, and (3) a tuple t indicating the configuration in which the fragment was executed. Each element of this tuple t_i specifies the index of the configuration of the i-th virtual gate in the circuit, similar to the result P_x at the beginning of the knitting process in Figure 6.2.

6.4 Knitter

The knitter is responsible for constructing the final result of the virtual circuit using the results of all circuit configurations. For this, the knitter runs two steps:

1. The results of all fragments in a specific configuration are **merged** together to the result P_x of the configured circuit C_x (figure 6.2). For this, the results of all fragment executions are simply multiplied to one another, because the results of fragments are independent from each other. For example, consider the circuit C_{11} in Figure 6.2 that consists of two fragments C_{11}^1 and C_{11}^2. The probability of measuring the state $|00\rangle$ when executing C_{11} is therefore
$$P_{11}(|00\rangle) = P_{11}^1(|0\rangle) \cdot P_{11}^2(|0\rangle),$$
where $P(s)$ is the probability of measuring state s when executing C.

2. The results P_x are **knitted** to the final result of the virtual circuit by recursively calling the `knit()` method on the virtual gates (Section 4.1). This procedure is shown by example in Figure 6.2: At the maximum recursion depth, the first virtual gate vG_1 is knitted by calling `knit` on the results of P_{1x} and P_{2x} to get the new result of P_1 and P_2. This procedure carries on up to the last virtual gate (here vG_2), whose `knit` method finally calculates the end-result. In general, the procedure calculates along a recursion-tree where the root represents the final result. The depth of this recursion-tree

is the number of virtual gates in the circuit, and the branching factor at each depth i is the amount of configurations of the i-th virtual gate.

When virtualizing multiple gates with up to 6 configurations, the number of results to merge and knit explodes due to the exponential complexity which is already recognizable in the small example of figure 6.2. To allow more virtualizations to be calculated in reasonable amounts of time, the knitter process can be parallelized in three different ways. First, the merging for each configured circuit C_x is independent from the merging of other circuits, and can therefore trivially be parallelized. Second, the recursive knitting can be parallelized by independently knitting sub-trees of the recursion-tree, whose result is then combined to the final result. By the example of Figure 6.2, the sub-results P_1 and P_2 can be calculated in parallel. And third, for both merging and knitting, the calculation can be further parallelized along the probability distributions of the results, meaning that the merged or knitted probability for each state can be calculated independently, making the entire calculations parallelizable into the smallest units.

Note, that although we recognize that the knitter can be easily parallelized and scaled in a distributed computing setting, building such a distributed system is not the focus of this work; however, we will be present a scalable system in future projects.

7 Evaluation

In this chapter we evaluate our implementation of the DQS framework. For this we run experiments with commonly used quantum circuits on real quantum hardware and compare the fidelities of circuit executions of different distributed transpilers with the state-of-the-art transpilers.

7.1 Implementation

We implement a prototype of DQS from the ground up as a Python framework. As the basis for the virtual circuit and the distributed transpiler, we implement a simple quantum circuit object that represents the circuit as a subset of the OpenQASM representation [28]. This makes DQS easily compatible with existing quantum computing frameworks such as Qiskit or Cirq to use their quantum algorithm libraries [29, 30]. On this circuit object, we implement our new virtual circuit and distributed transpilation abstractions as described in Section 4, using the networkx library for representation and computations on the circuit's connectivity graph [31]. The implementations of our distributed transpilers with less than 15 lines of Python code for each transpiler shows that implementing a custom transpiler is clear and simple for an end-user.

For circuit and fragment execution in the execution engine, we rely mainly on the Qiskit framework [29], both for circuit simulation and for circuit execution on the superconducting quantum devices provided by the IBM Quantum Experience [2]. To parallelize the circuit execution and knitting process, we use the Ray distributed framework [32], which allows the user to easily scale the execution engine in a cloud environment.

7.2 Methodology

7.2.1 Hardware

We perform our experiments using the superconducting Falcon r5.11H devices offered by the IBM Quantum Experience [2]. The Falcon r5.11H devices have 7 physical qubits arranged in the layout shown in Figure 7.1. We perform all of our experiments on three of the five currently available Falcon r5.11H devices, namely IBM Nairobi, IBM Perth, and IBM Lagos.

7.2.2 Metrics and Benchmarks

To measure the quality of an execution result of an optimized circuit, we use the Hellinger fidelity as defined in Definition 2.1.1. The Hellinger fidelity is a percentage indicator of how

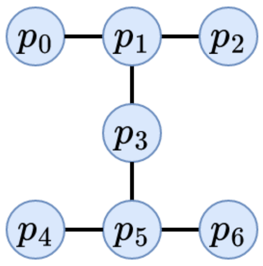


Figure 7.1: The physical qubit layout of an IBM Falcon r5.11H quantum device.

close the probability distribution obtained from an execution on noisy hardware is to the perfect probability distribution obtained by a classical simulation.

To benchmark the transpilers described in Section 5, we compare the result fidelity of the baseline (without any virtualization) with the result fidelity of an optimized circuit using the distributed transpilers. To this end, we perform experiments on the following three types of circuits used in popular quantum algorithms. These circuits are generated using the SupermarQ benchmark suite [18]; a more detailed description of how the circuits are used in applications can be found in this paper. This benchmark suite can be used to generate common circuits of varying length and number of qubits.

GHZ

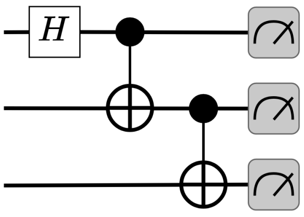


Figure 7.2: A 3-qubit GHZ circuit.

A GHZ circuit is used to generate the GHZ state that maximally entangles all qubits in a circuit [33] . Measuring the qubits in the GHZ state gives a 50 percent probability for the $|00..0\rangle$ state and a 50 percent probability for the $|11...1\rangle$ state. The GHZ circuit for 3 qubits is shown in Figure 7.2. It consists of a Hadamard gate in the first qubit and then a ladder of CNOT gates from the upper to the lower qubits. The circuit can be easily created with a different number of qubits by extending this CNOT gate ladder.

Hamiltonian Simulation

A Hamiltionian simulation is a quantum algorithm for simulating the time evolution of quantum systems [34]. The particular Hamiltonian simulation used in the SupermarQ benchmark suite performs quantum circuits similar to the circuit shown in Figure 7.3. The

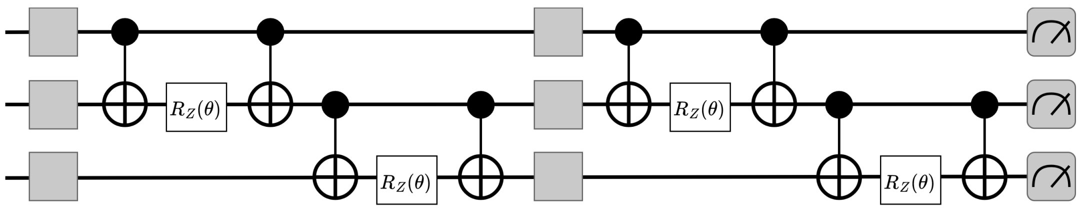


Figure 7.3: A 3-qubit Hamiltonian simulation circuit with 2 layers.

circuits consist of a variable number of layers, with each layer consisting of unary gates and a ladder of (CNOT, RZ(θ), CNOT) sequences. Similar to the GHZ circuit, the circuit is created with more qubits by expanding each layer in both the unary gates and the binary gate ladder. Note that a sequence of (CNOT, RZ(θ), CNOT) is equal to the single binary RZZ(θ) gate (Figure 2.2). Therefore, a simple first "optimization" used in our benchmarks is to convert the sequences into RZZ gates, halving the number of total binary gates. This conversion is realized by a simple distributed transpiler pass.

VQE

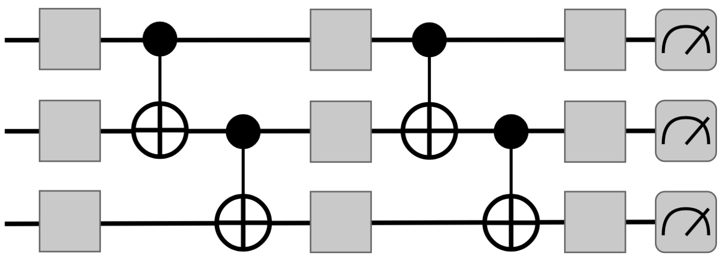


Figure 7.4: A 3-qubit VQE circuit with 2 layers

The Variational Quantum Eigensolver (VQE) [35] is a quantum algorithm used to determine the eigenvalues of a given matrix. The specific SupermarQ circuit is generated similarly to the Hamiltionian simulation circuit, i.e., it consists of several layers of binary gate ladders (with CNOT gates) and sequences of unary gates between the ladders.

7.2.3 Benchmark Execution and Fairness

Every circuit or fragment executed in our evaluation (both baselines and virtual circuits) is executed with the following three properties:

- Maximum amount (20000) of available shots.
- Highest possible Qiskit optimization level (3) to transpile each circuit for a device [20].
- Choosing the optimal physical qubits for both, measurement error and CNOT-Gate error, using the mapomatic framework [25].

Since the fragments of a virtual circuit are executed up to $\mathcal{O}(6^k)$ times for k virtual gates, the execution of the virtual circuit runs up to $6^k \cdot 20000$ shots in total and therefore might have an unfair advantage in the accuracy of the result when compared to the result of executing the baseline once. To keep the comparison fair, we execute the baseline exactly as many times as fragments are executed in the virtual circuit, and take the average result of all executions as the baseline.

7.3 Decomposition Transpilers

In this section, we evaluate the use of decomposition transpilers (Section 5.1) by applying the bisection pass to the three benchmark circuits. We then show how we scale the Hamiltion simulation circuit to 16 qubits using only the 7-qubit devices.

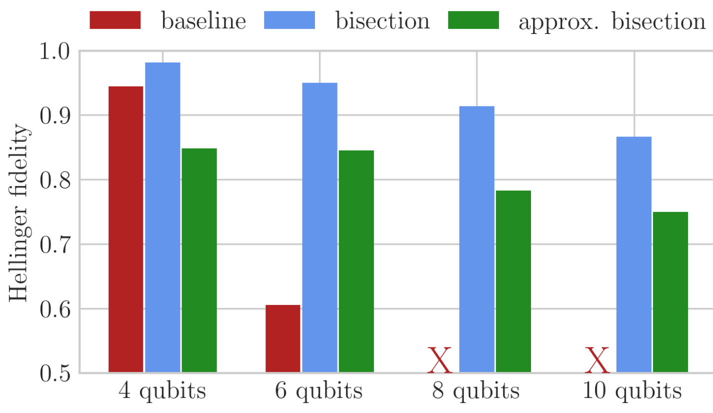
7.3.1 Bisection

The results of evaluating the execution of the benchmark circuits with the bisection pass are shown in Figure 7.5. Each benchmark circuit is executed with 4, 6, 8, and 10 qubits on the IBM Perth device, with fragments of a bisected circuit executed sequentially on the device. We execute the circuit in the following three variants:

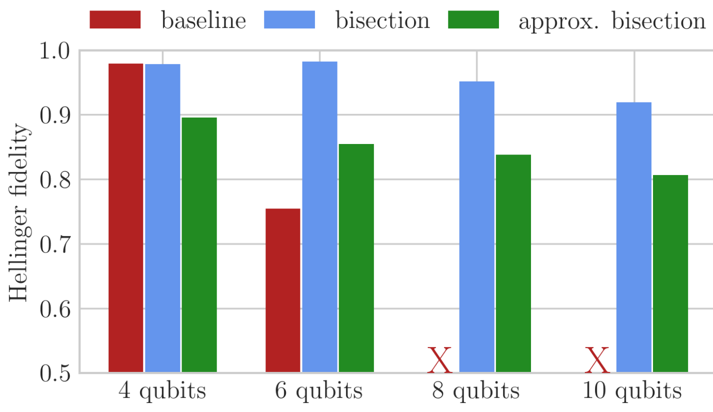
1. The baseline circuit without any virtualization (red), which can only be executed with up to 6 qubits because of the maximum number of qubits of the quantum device.
2. The bisection pass using the standard virtual gates (blue).
3. The bisection pass using the approximate virtual gates (Section 5.4), which incur less computational overhead (green).

For the Hamiltonian simulation and the VQE benchmark, the fidelty of the results and the ability to perform larger circuits improve significantly (Figures 7.5 a and b): For the 4-qubit circuits, we observe high fidelity of both the baseline and the standard bisection at 95% or more. For the 6-qubit circuits, on the other hand, the fidelity of the standard bisection increases by a factor of 1.25 - 1.5$\times$ compared to the baseline, and the approximate bisection also improves the accuracy by a factor of 1.4$\times$ for the Hamiltonian simulation and a factor of 1.1$\times$ for the VQE benchmark. Using the standard bisection for the 8- and 10-qubit circuits still produces high-quality results with 86%-95% fidelity, even though circuits of this size are too large to run on the hardware available to us. Also, the approximate bisection for these two circuits leads to relatively high fidelity with an absolute loss of 10% fidelity on average.

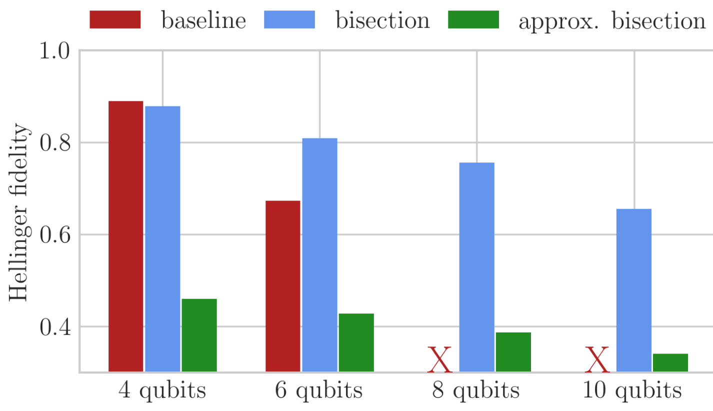
The GHZ baseline generally produces noisier results than the other two benchmarks (Figure 7.5c). Therefore, we can also see worse results in fidelity for the standard bisection, as the executed fragments also provide noisy results that add up to a noisy final result. This is also very evident in the low fidelity values measured in approximate bisection. Here, the noise of the fragment execution and the error caused by the approximation add up to a highly noisy result.



(a) Hamiltonian Simulation.



(b) VQE.



(c) GHZ.

Figure 7.5: Using the bisection pass to optimize the benchmark circuits on IBM Perth. Both cases, using accurate (blue) and approximate (green) virtual gates are compared with the baseline (red, without virtualization). The baseline is limited by the number of qubits on the device (red crosses).

From this experiment, we conclude that standard bisection improves the fidelity of the results compared to running the circuit without virtualization and with using only traditional circuit optimizations. We also see that the use of approximate virtual gates is only beneficial when the fragments produce high quality results.

7.3.2 Scaling Maximum Circuit Sizes with Ladder Decomposition

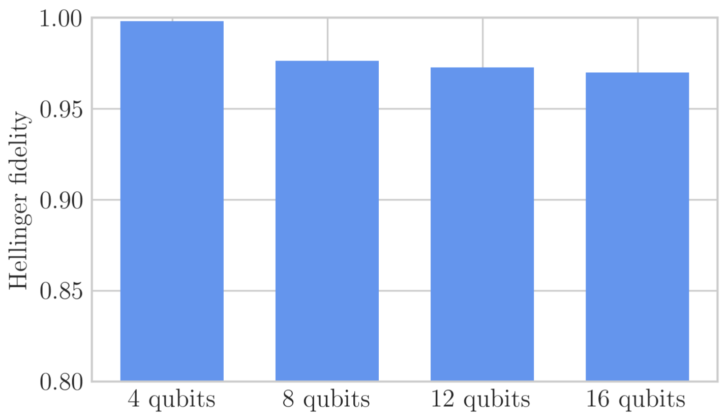


Figure 7.6: Scaling the circuit size of Hamiltonian simulation to more than double the size of the quantum device while loosing only 3% fidelity on IBM Lagos.

In this experiment, we scale the circuit size of the single-layer Hamiltonian simulation circuit to more than twice the size of the quantum device on which we run the virtual circuit, as depicted in Figure 7.6. To do this, we use the ladder decomposition (Section 5.1.2) to decompose each circuit of sizes 4, 8, 12, and 16 qubits into 4 fragments each and execute the fragments sequentially on the IBM Lagos device.

As the graph shows, when we execute the circuit with 4 fragments of 1 qubit to execute the virtual circuit of 4 qubits, we get a perfect accuracy of 99.8%. If we increase the size of the circuit by 4× (4 fragments of 4 qubits each) to a total circuit size of 16 qubits, which is 2.3× the size of the IBM r5.11H device, we still get a near perfect fidelity of 97%.

7.4 Hardware Mapping Transpilers

For comparison of the fragment-to-device mapping passes (Section 5.2), we run the passes on Hamiltonian simulation circuits of sizes 4, 8, and 12 qubits. Each of the virtual circuits was decomposed into 4 fragments using the ladder decomposition pass before being passed to the respective fragment-to-device mapping pass. The devices available to run the fragments are IBM Lagos and IBM Nairobi.

As expected, the maximum utility pass yields the worst results with up to 0.08× lower fidelity than the minimal noise pass, because it merges the fragments to the maximum

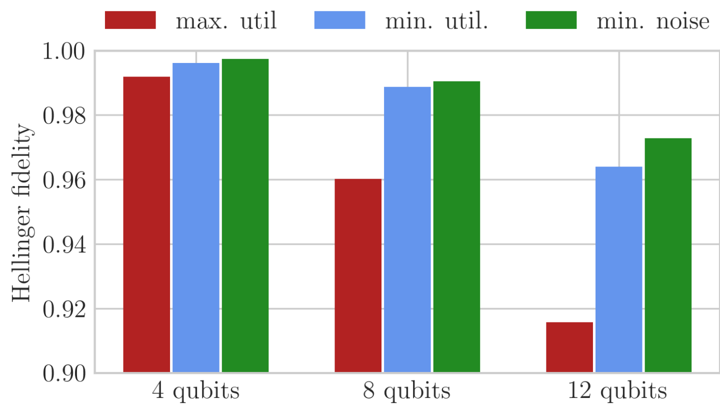


Figure 7.7: Comparing the circuit-to-device -mapping transpilers. Every transpiler has the backends IBM Lagos and IBM Nairobi as their available backends that can be mapped to.

possible size, which must then also use run on physical qubits with high error rates. With minimal utility and minimal noise passes, we obtain similar high fidelities of >96% since they use the same, minimal sizes for their fragments. However, the fidelity of the minimal noise pass is slightly higher because it maps all fragments to the backend with the best quality (at this time, IBM Lagos) and the minimal utilization pass would run fragments on both devices in parallel; therefore, the slightly noisier IBM Nairobi device is also used.

8 Related Work

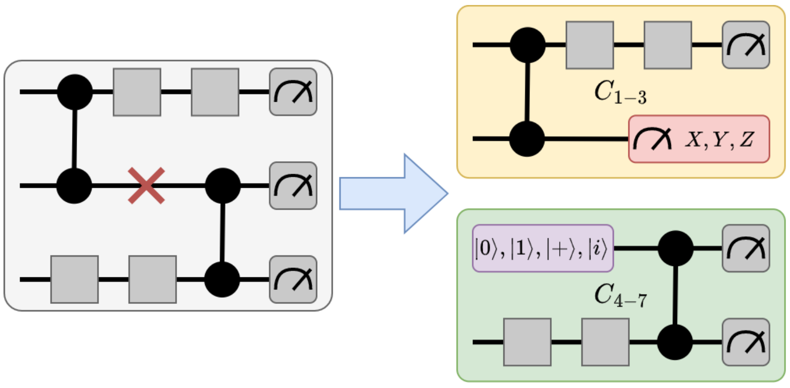


Figure 8.1: Cutting and simulating a qubit wire by sampling measurements in various bases before the cut, and multiple input states after the cut.

In the area of circuit cutting and knitting, much work has focused on the technique of wire cutting, which, unlike gate virtualization, allows the simulation of qubit wires. A qubit wire is the time-evolution of a qubit. Wire cutting is therefore a time-wise cutting technique, while the binary gate virtualization is a space-wise cutting technique. We briefly describe wire cutting as described in the work of Peng et al. [10] and Tang et al. [11]. These papers provide a detailed mathematical description of the technique. The idea of wire cutting, is shown in Figure 8.1: The wire (= the time-progress) of the second qubit in the original circuit is cut at the red cross, creating two smaller circuit fragments that can be executed independently. To calculate the resulting probability distribution of the original circuit using only the fragments, the fragments must be sampled and then the results of the sampling are combined in a knitting process to produce the final probability distribution. To sample the circuit, the fragment *before* the cut (yellow fragment) is sampled by measuring in the three bases X, Y, and Z at the point of the cut (red measurement node), resulting in the circuit configurations C_{1-3}. The fragment *after* the cut (green fragment) is sampled by preparing the four input states $|0\rangle, |1\rangle, |+\rangle, |i\rangle$, resulting in the circuit configurations C_{4-7}. To then knit the results, a knitting formula is used that informally uses the results P_{1-3} to weight the results of P_{4-7}'s different input states. Since we have have to create a maximum of 4 samples for each cut on each fragment, the sampling overhead of fragments to execute and the knitting overhead is $\mathcal{O}(4^k)$ for k cuts in a given circuit [11].

Compared to gate virtualization, wire cutting has the disadvantage that it can only be used to decompose a circuit into multiple fragments and therefore cannot reduce the connectivity of the resulting fragments by removing binary gates. This results in the need to break the circuit into fragments to benefit from wire cutting, while gate virtualization can be used to virtualize any binary gate without additional constraints to reduce noise. While wire cutting can help optimize a circuit to run on the small NISQ devices with less overall noise, it cannot help improve qubit layout and reduce SWAP gates to optimize a circuit to run on a specific device, as we have demonstrated in our work with qubit layout transpilers. To summarize, gate virtualization provides more flexibility in finding circuit optimizations.

CutQC [11] is a framework that, similar to DQS for gate virtualization, aims to automate the workflow of wire cutting and executing fragments. To automate the cutting of a given large quantum circuit, CutQC uses a mixed-integer programming (MIP) solver that determines the optimal cuts depending on parameters such as the maximum number of fragments into which the circuit should be cut. This automatic cutting of circuits is similar to a distributed transpiler in DQS, but the MIP cut searcher is much less flexible because users can only switch optimization parameters and cannot exploit certain patterns in certain quantum circuits. So, our distributed transpilers can offer a wider range of optimization passes that can be combined to a fully customizable transpiler. However, a MIP cut searcher is a very suitable technique to optimally decompose an arbitrary circuit, and thus can also be implemented as a distributed transpiler pass as part of DQS to find optimal gates for virtualization. Similar to DQS knitting in the execution engine, the kitting process in CutQC is also highly parallelizable, allowing CutQC to scale in a distributed cluster. To further reduce the computational cost of knitting, CutQC also introduces a useful *dynamic definition* algorithm to detect the initial states that have probability > 0 in the final probability distribution P, so that not all 2^q (for q qubits) initial states in the state space of the original circuit C need to be knitted to construct the entire probability distribution. Such a technique could also be applied to DQS to cooperate with the overhead mitigation of our approximate virtualization technique.

In summary, wire cutting and gate virtualization have both advantages and disadvantages, e.g., because gate virtualization is more flexible, while circuit cutting has a lower computational cost. In addition, circuit decomposition using one of the two techniques may be more advantageous than the other when applied to a particular circuit in terms of the number of cuts required or of overall noise reduction. Thus, both techniques need to be further investigated and compared. Ideally, a system will be developed that would combine both gate virtualization and wire cutting to take advantage of both techniques.

9 Conclusion and Outlook

This work serves as the first proof-of-concept for the novel concepts of virtual circuits and distributed transpilers that allow transparent optimization of circuits using virtual gates [9, 10]. These new concepts are intended to allow users to create automated transpilers that leverage this new strategy to optimize circuits in any way possible, including specifying the exact technique for virtualizing a binary gate, selecting the specific binary gates in a circuit to be virtualized, and managing the decomposition of the circuit into fragments. The abstractions exposed by DQS can be easily integrated into existing quantum computing frameworks, as they generically extend and reuse state-of-the-art circuit optimization concepts.

Using our distributed transpiler framework, we identify the three main pillars of distributed transpilation: (1) *circuit decomposition* transpilers that use community detection algorithms to decompose circuits into multiple fragments of equal size, (2) *fragment-to-device-mapping* transpilers that automatically decide on which quantum device to execute each fragment, e.g., to minimize the expected total noise, and (3) *qubit layout* transpilers that assist traditional transpilers in finding the optimal logical-to-physical qubit allocation by reducing the amount of required SWAP gates.

To execute the new virtual circuit abstraction, we design and implement an early prototype execution engine that uses a just-in-time transpiler to map each fragment to a device just before execution, and that can parallelize the quantum execution and knitting process. Scaling the execution engine in a distributed system allows us to do a handful more gate virtualizations, but the exponential cost of $\mathcal{O}(6^k)$ for k virtual gates will still severely limit the number of gates we can virtualize in a large circuit. To reduce the computational cost, we also discuss and evaluate the use of approximation techniques that have a cost of *just* $\mathcal{O}(2^k)$.

As part of this work, we have also presented the implementation of a series of distributed transpilers that, in addition to their straightforwardness, demonstrate their effectiveness in mitigating noise and executing larger circuits on smaller quantum devices. Specifically, our evaluation on three commonly used quantum circuits shows that, compared to state-of-the-art transpilers, we can achieve up to 1.5× higher fidelity and scale the size of the circuits up to 2.3× the size of the quantum device on which the circuit fragments run.

In future work, the DQS framework can be improved and extended in several ways. Our collection of distributed transpilers should be extended with various new transpiler passes to find new circuit optimizations for different types of quantum circuits. These new distributed transpilers can either uncover patterns in quantum circuits or find a unified transpiler that finds the optimal gates for virtualization, similar to the mixed integer programming optimizer presented in the work of CutQC [11]. Newly created and existing distributed transpilers should be studied extensively to find optimal transpilers for common quantum algorithms.

The scalability of DQS, both in terms of circuit size and number of gate virtualizations, can be further improved by efficient scheduling of quantum computational tasks and distributed classical computing, as well as by further investigation of approximation techniques. Finally, it would be beneficial to integrate wire cutting into the DQS framework, as it could help to decompose certain circuits more efficiently. For this, parts of the CutQC framework can be reused and adapted to DQS's distributed transpiler toolbox.

In the further future, DQS may also leverage new quantum hardware that will be developed in the coming years [7]. This new hardware is anticipated to enable classical communication between quantum devices, allowing gate virtualizations during the runtime of a quantum circuit whose fragments are assigned to each of the distributed quantum devices [26]. Such a technique could significantly reduce the computational cost of gate virtualization and, by providing a simulation, DQS could allow researchers to investigate this technique before the actual hardware becomes a reality.

Bibliography

[1] "40 years of quantum computing". In: *Nature Reviews Physics* 4.1 (2022), pp. 1–1. DOI: 10.1038/s42254-021-00410-6. URL: https://doi.org/10.1038/s42254-021-00410-6.

[2] *IBM Quantum*. https://www.ibm.com/quantum-computing/. Accessed: 2022-04-11.

[3] *Google Cirq*. https://quantumai.google/cirq. Accessed: 2022-04-11.

[4] *AWS Bracket*. https://aws.amazon.com/braket/. Accessed: 2022-04-11.

[5] J. Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331/q-2018-08-06-79.

[6] A. Wack, H. Paik, A. Javadi-Abhari, P. Jurcevic, I. Faro, J. M. Gambetta, and B. R. Johnson. "Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers". In: *arXiv preprint arXiv:2110.14108* (2021).

[7] *IBMQ roadmap*. https://research.ibm.com/blog/ibm-quantum-roadmap-2025. 2022.

[8] M. Suchara, J. Kubiatowicz, A. Faruque, F. T. Chong, C.-Y. Lai, and G. Paz. "QuRE: The Quantum Resource Estimator toolbox". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 419–426. DOI: 10.1109/ICCD.2013.6657074.

[9] K. Mitarai and K. Fujii. "Constructing a virtual two-qubit gate by sampling single-qubit operations". In: *New Journal of Physics* 23.2 (2021), p. 023021.

[10] T. Peng, A. W. Harrow, M. Ozols, and X. Wu. "Simulating large quantum circuits on a small quantum computer". In: *Physical Review Letters* 125.15 (2020), p. 150504.

[11] W. Tang, T. Tomesh, M. Suchara, J. Larson, and M. Martonosi. "Cutqc: using small quantum computers for large quantum circuit evaluations". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 473–486.

[12] F. Bloch. "Nuclear induction". In: *Physical review* 70.7-8 (1946), p. 460.

[13] J. I. Cirac and P. Zoller. "Quantum computations with cold trapped ions". In: *Physical review letters* 74.20 (1995), p. 4091.

[14] J. M. Gambetta, J. M. Chow, and M. Steffen. "Building logical qubits in a superconducting quantum computing system". In: *npj quantum information* 3.1 (2017), pp. 1–7.

[15] D. D. Stancil and G. T. Byrd. *Principles of Superconducting Quantum Computers*. John Wiley & Sons, 2022.

[16] E. Hellinger. "Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen." In: *Journal für die reine und angewandte Mathematik* 1909.136 (1909), pp. 210–271.

[17] *Qiskit Hellinger fidelity*. `https://qiskit.org/documentation/stubs/qiskit.quantum_info.hellinger_fidelity.html`. Accessed: 2022-04-11.

[18] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Viszlai, X.-C. Wu, N. Hardavellas, M. R. Martonosi, and F. T. Chong. "Supermarq: A scalable quantum benchmark suite". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 587–603.

[19] E. Farhi, J. Goldstone, and S. Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. DOI: `10.48550/ARXIV.1411.4028`. URL: `https://arxiv.org/abs/1411.4028`.

[20] *Qiskit transpiler*. `https://qiskit.org/documentation/apidoc/transpiler.html`. 2022.

[21] H. L. Tang, V. Shkolnikov, G. S. Barron, H. R. Grimsley, N. J. Mayhall, E. Barnes, and S. E. Economou. "qubit-adapt-vqe: An adaptive algorithm for constructing hardware-efficient ansätze on a quantum processor". In: *PRX Quantum* 2.2 (2021), p. 020310.

[22] W. Feller. *An introduction to probability theory and its applications, vol 2*. John Wiley & Sons, 2008.

[23] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". In: *The Bell system technical journal* 49.2 (1970), pp. 291–307.

[24] S. Martello and P. Toth. "Bin-packing problem". In: *Knapsack problems: Algorithms and computer implementations* (1990), pp. 221–245.

[25] P. Nation, M. Treinish, and C. Possel. *mapomatic*. `https://github.com/Qiskit-Partners/mapomatic`. 2022.

[26] C. Tüysüz, G. Clemente, A. Crippa, T. Hartung, S. Kühn, and K. Jansen. "Classical Splitting of Parametrized Quantum Circuits". In: *arXiv preprint arXiv:2206.09641* (2022).

[27] *IBMQ fairshare queuing*. `https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/queue/`. 2022.

[28] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. "Open quantum assembly language". In: *arXiv preprint arXiv:1707.03429* (2017).

[29] M. S. ANIS, Abby-Mitchell, H. Abraham, et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2021. DOI: `10.5281/zenodo.2573505`.

[30] C. Developers. *Cirq*. Version v0.14.1. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors. Apr. 2022. DOI: `10.5281/zenodo.6599601`. URL: `https://doi.org/10.5281/zenodo.6599601`.

[31] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. Pasadena, CA USA, 2008, pp. 11–15.

[32] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. “Ray: A distributed framework for emerging {AI} applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[33] D. M. Greenberger, M. A. Horne, and A. Zeilinger. “Going beyond Bell’s theorem”. In: *Bell’s theorem, quantum theory and conceptions of the universe*. Springer, 1989, pp. 69–72.

[34] R. P. Feynman. “Simulating physics with computers”. In: *Feynman and computation*. CRC Press, 2018, pp. 133–153.

[35] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O’brien. “A variational eigenvalue solver on a photonic quantum processor”. In: *Nature communications* 5.1 (2014), pp. 1–7.