SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hardware-Dependent Visualization of Quantum Circuits

Donia Fouzri

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hardware-Dependent Visualization of Quantum Circuits

Hardware-Abhängige Visualisierung von Quanten-Schaltkreisen

Author:	Donia Fouzri
Supervisor:	Prof. Dr. Helmut Seidl
Advisors:	M.Sc. Yannick Stade and M.Sc. Yanbin Chen
Submission Date:	15.05.2023

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

Munich, 15.05.2023 Donia Fouzri

Acknowledgments

I would like to express my gratitude to my supervisors, Yannick Stade and Yanbin Chen, without whom this work would not have been possible. Their continuous support and constructive feedback have been of great help during the period of this thesis. Our multiple discussions and exchange of ideas have only made this work a more valuable and enriching undertaking.

I would also like to thank Dr. Johannes Zeiher, at the Max Planck Institute of Quantum Optics, for providing and discussing information contributing to this work.

Abstract

Quantum compilers play a key role in quantum computing. They adjust quantum software and adapt it to the constraints of the quantum platform. However, the black-box approach of these compilers gives little control to the programmer over the compiling process and the changes made to the quantum code. It can jeopardize the optimization techniques used by the programmer before compiling. As a solution, we present an implementation of a tool that connects quantum hardware and software and allows the developer to take a closer look into the hardware. It will support hardware compatibility testing and both manual and built-in code adjustment. It offers a step by step compiling process, controlled by the programmer.

Contents

1 Introduction

With a billion times faster calculations and potentially unbreakable cryptography protocols[7], quantum computing is indeed a revolutionary technology. Using quantum entanglement and superposition[7], two of the most important principles in quantum mechanics, quantum platforms can achieve high levels of parallelism and perform complex computations, previously unattainable with classical computing.

These opportunities that quantum computing promises, have lead to a race towards the newest most efficient and error-free platforms. Research centers and tech-companies are also developing different programming languages, frameworks and kits, at a fast pace, to support this new field of computations and bring it closer to the user.

Given that quantum computers are physical systems run by the laws of quantum mechanics, quantum algorithms need to be dissected and transformed into simple machine instructions executable on a low hardware level. These intermediate tools, connecting quantum code and low level platforms, are quantum compilers and there is a variety of them available today, for different quantum backends[17].

Additionally, every quantum computer imposes several constraints on quantum code[17], which need to be respected in order for the code to run. These constraints can change from one computer to another[17], and it is the job of the compiler to apply the necessary adjustments to the algorithm before it is executed.

Our main problem, however, is that quantum compilers have a black-box approach. They run in the background after the developer starts the code simulation. Compilers will transform the code, apply the necessary changes and send it to the hardware. This process leaves the programmer in the dark concerning the compiling process and the state of the code which will be sent to the backend. This would not be a problem, if it was not for the critical changes happening to the quantum code, concerning its hardware compatibility.

In order to fully understand why changing the quantum code in the compilation can be problematic, we need to observe the limitations of quantum computing. Unfortunately, despite their immense potential, these platforms are still limited due to the occurring error which corrupts the output results. Error being a central object of concern in quantum computing[7], programmers use different optimization techniques in their code to minimize the error[8]. These optimizations will not necessarily be preserved in the compiling process, due to the divergent priorities of the programmer

and the compiler.

To solve this problem, we need to fill the gap between quantum software and hardware while allowing developers more control over the transformations happening to their code in the compiling process. In other words we will implement a white-box approach of a quantum compiler.

We develop a tool where the user has direct access to the quantum hardware characteristics and constraints. For this, we need to find a universal data structure with which we can represent any quantum computer regardless of its nature. We then offer the possibility to test the compatibility of the code with the chosen hardware. After running the tests, we will provide the user with detailed information concerning any potential problems that might need adjustments. The user can rely on this feedback to manually make the necessary changes to the code and, therefore, adjust the algorithm without ruining the previous optimization efforts. This is a user-controlled compiling process, tailored to the priorities of the programmer with respect to the hardware specifications. Moreover, we will provide built-in solutions for the occurring issues in case the user chooses not to solve them manually. We will also add other features for a better usability and functionality of the tool.

In this work, we go through some background concepts of quantum computing and available tools that are similar to our implementation. Next, we will explain the tool we developed, its structure, key features and the different algorithms implemented. This will also be presented in fig. 3.1 with a detailed diagram. We will provide proof of correctness for complex algorithms and expose their limitations. Finally, we will suggest possible improvements to be performed in the future.

2 Background

In this Chapter, we explain the concepts of quantum computing that are relevant to our work. We will handle the concept of both, quantum hardware and quantum software. Additionally, we will introduce Qiskit due to its important role in this work. Finally, we discuss some other tools and frameworks that are related to our topic.

2.1 Quantum Computing

Quantum computing is a new computation technology, based on the laws of quantum mechanics and capable of exponentially reducing the runtime of many algorithms and complex calculations. It has different characteristics from classical computing which we will explore in this section.

2.1.1 Qubits and Quantum State

Instead of classical bits, quantum computers use quantum bits usually referred to as *qubits*[7]. Qubits can take the values 0, 1 or a superposition of these two. This is a key to the efficiency of this technology. The quantum state of a qubit can be represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α^2 and β^2 are respectively the probabilities of the states $|0\rangle$ and $|1\rangle$ after measurement. Additionally, α and β have to verify the condition[7]: $|\alpha|^2 + |\beta|^2 = 1$.

2.1.2 Quantum Gates

To operate on qubits we use unitary $2^n \times 2^n$ matrices in $\mathbb{C}$ with n the number of qubits involved in the operation. Such matrices are also referred to as quantum gates. They are applied to the quantum state vector of the qubits to generate a new state vector[7, chapter. 3].

As an example, applying the *Not* operator, also known as the X gate, to the initial state $|1\rangle$ results in the following: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0+1 \\ 0+0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$.

An important gate, for this work, is the *Swap* gate. It switches the state of two qubits[7].

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

Applying *Swap* to $|10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$ results in the state vector: $|01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$.

2.1.3 Quantum Circuit

Quantum algorithms are modelled as quantum circuits. A circuit consists of a set of quantum gates applied to a set of qubits in a certain order [10]. Aside from quantum gates, a circuit can also include barriers and measurements. A barrier delays the execution of gates subsequent to it, until all prior operations are terminated.

Each quantum circuit has specific semantics depending on the nature and order of the gates and the initial state vector.

The depth of a circuit is also an important feature. This is the size of the longest path of non-parallel gates acting on a common qubit[10].

In order to visualize a quantum circuit we represent a qubit with a line also called a wire. On the other hand, we denote most unary and binary gates with a box containing their name placed on the wires of the corresponding qubits as presented in fig. 2.1.

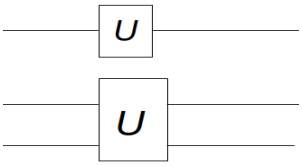

Figure 2.1: Unary and binary gates representation in circuit visualization

2.1.4 Quantum Hardware

To create a qubit based system, we need to conduct a variety of physical experiments[7, chapter. 5]. Their goal is to produce single units of information behaving similar to quantum particles and adhering to the rules of quantum mechanics[7, chapter. 5]. These will represent the qubits of the platform.

There are different ways to build such platforms. Some popular approaches are: Neutral Atoms, Superconducting Qubits, Trapped Ions and Spin Qubits.[7, chapter. 5].

The physical implementation of quantum platforms influences some key characteristics of the hardware such as error and connectivity.

Error handling plays an important role in quantum computing, and errors can arise due to external and internal factors. One form of error that is relevant to our work is crosstalk. Crosstalk happens when applying two or more gates simultaneously, usually acting on qubits within a short distance from each other[25]. In this case, the operations are not isolated, and they influence each other causing inaccurate results[7, chapter. 5]. Depending on the hardware, the qubits that can potentially have crosstalk between them can vary.

Connectivity is also a key feature, because multi-qubit gates can only be applied to connected qubits. It is dependent on the physical architecture of the qubits and their spatial location[7, chapter. 5].

Later in this work, we will mention other important quantum hardware features, that are linked to the nature of the platform.

2.2 Qiskit

Qiskit is one of the various quantum software development kits available today. It is an open-source kit created by IBM. It offers multiple libraries to manipulate and simulate quantum algorithms[19].

Some other useful functionalities offered by this tool are quantum circuits visualization, support of Quantum Assembly (QASM) code[4] and running quantum algorithms on IBM's quantum computers via API tokens[19].

To handle quantum circuits in the background, qiskit transforms them into directed acyclic graphs[16] and then generates a layered data structure of the gates[16]. These layers will then be used by the built-in compiler to process and adjust the code to the backend of choice[20].

Each layer is a list containing parallel gates only, with a limit of one gate per qubit. The layers are then collected in a list. The indices of the layers inside the outer list indicate their order of execution. So the gates in $Layer_i$ are executed before those in $Layer_{i+1}$.

At the end we obtain a similar data structure:

$$Layers = [[gate_0, gate_1, ...]; [gate_j, gate_{j+1}, ...]...]$$

This data structure conserves the topological order of the gates while making the distinction between parallel and non-parallel gates.

In this work, many Qiskit libraries are used via function calls or using the initial code from the repository and making changes to it. Any code used from Qiskit has been taken from their open-source GitHub repository[15].

2.3 Related Work

With the growing focus on quantum compilers, we can find compilers allowing the user more flexibility and control over the compiling process. In this section we will explain the concept of a quantum compiler, and we will discuss some modern tools which share some common features with our work.

2.3.1 Quantum Compilers

Similar to classical computing, quantum compilers translate high-level code into machine instructions specific to the quantum hardware [10]. Considering the specific characteristics of the hardware, they transform the initial program into a set of low-level gates runnable on the platform[17].

Compilers also have different optimization algorithms which they run over the code for a more accurate output[17]. Some frequently used algorithms are: gate cancellation, gate fusion and pattern matching[21].

There is also a special part of the compiler usually referred to as the scheduler [9]. This component has the task of assigning to gates the order and time of their execution. It usually aims to minimize the global runtime of the circuit and the decoherence[9].

2.3.2 Related Compilers

Among the many compilers available in the field, some of them allow the user to have control over the different optimizations and adjustments applied to the quantum code. Compilers such as the Qiskit Transpiler[21], Cirq[5] and Quil[3] allow the user to choose the level and type of optimization to be applied. In addition, Quil and Cirq allow the user to define their own custom compilation passes.

In order to solve crosstalk, compilers such as the Qiskit Transpiler combine logical to physical qubit mapping algorithms with the insertion of barriers between every two problematic gates[20].

Another important factor is that each of these compilers is compatible with more than one quantum hardware. However, there is still no tool available that supports all types of quantum backends.

3 Description of the Tool and User Interface

In this chapter, we will explain how the tool works and go through the different features offered in the user interface. Next, We will focus on the hardware configuration files and the various types of circuit visualization.

We chose to leave testing the software-hardware compatibility for chapter 4 and the solution for the potential issues for chapter 5.

3.1 Functionalities

Our goal with this tool is to enable quantum programmers to have a closer look into the constraints of the quantum platforms they are using. Via this tool, we also want to give them more control on the overall process of mapping their quantum code to the hardware of choice.

In order to make this a user-friendly experience, we decided to implement this tool as a desktop application. The global structure of our app is shown in a diagram form in fig. 3.1. The figure also enumerates the files containing the code for each element of the app, which will be discussed in this chapter and in the next ones.

The application is implemented in Python using the framework PyQt6. This enabled us to use the Qiskit libraries with Python. We use those libraries to analyze the quantum code, represent quantum circuits, gates and registers and other concepts.

In the app, the user can choose the hardware they want to run their code on. All available quantum platforms are represented through configuration files. We display a summarized description of the chosen hardware along with a graph representing the qubit connectivity[23]. In fig. 3.2, we can see how the information of the chosen hardware is presented in the application.

After selecting the platform, the user needs to enter the quantum code to be processed. For this purpose we offer a built-in text editor, which supports code in Quantum Assembly Language (QASM 2.0)[4]. The user can write the desired code in there or upload a (.qasm) file from an external source. This code will be then displayed in the text editor. The user is allowed to edit the code at any time while using the app.

Now that the code is loaded, the developer can start using the different features in the app.

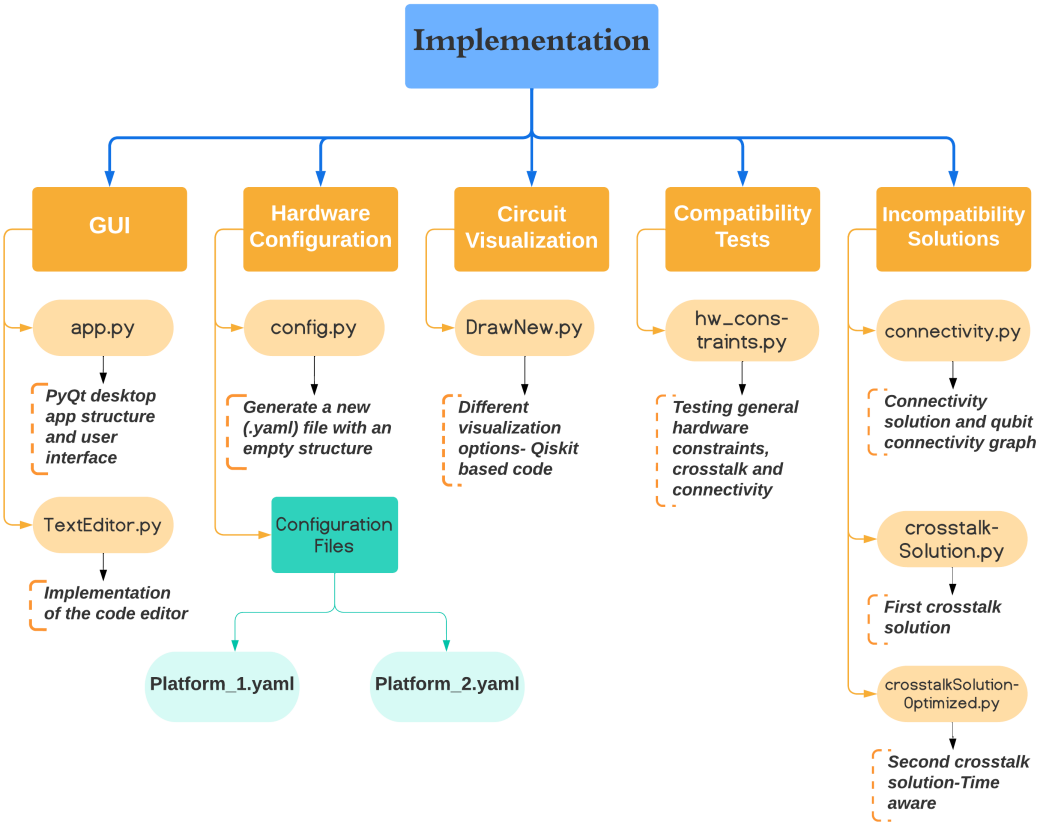

Figure 3.1: Diagram representing the main parts of the implementation with their respective code files

First, we offer three different circuit visualization options, which we will explain in section 3.3. These options are shown in fig. 3.3 along with the code editor, containing QASM code which would be visualized.

Next, the user can start testing the circuit for hardware compatibility. The tool offers different testing options and thorough feedback for the different issues (more about this in chapter 4). In fig. 3.4, we have different feedback boxes for the various tests. We can see the "Test" and "Solve" buttons in each box and how the feedback is formulated.

The user can choose to solve the suggested issues manually or apply the built-in solutions. The built-in solution will return new code with the same semantics as the input code. This will be discussed in chapter 5. At any point the user can change the code, test or solve the issues. There is no order imposed on how to run the different features.

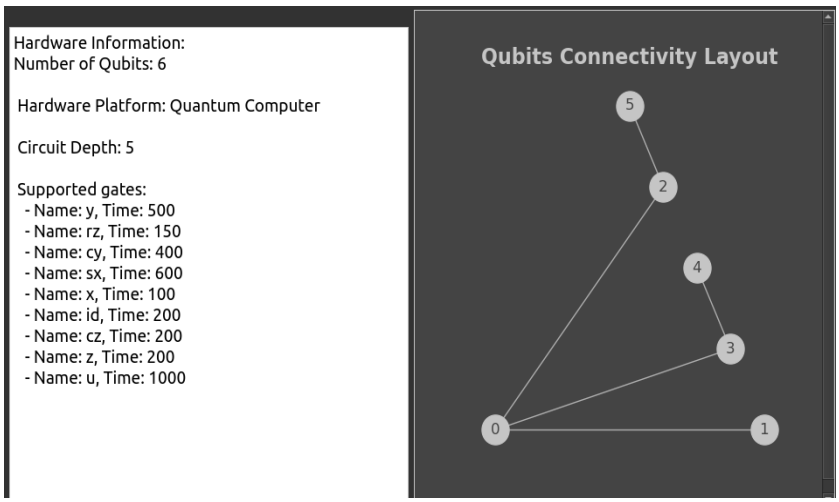

Figure 3.2: Screen capture, from the app, representing how the hardware information are displayed

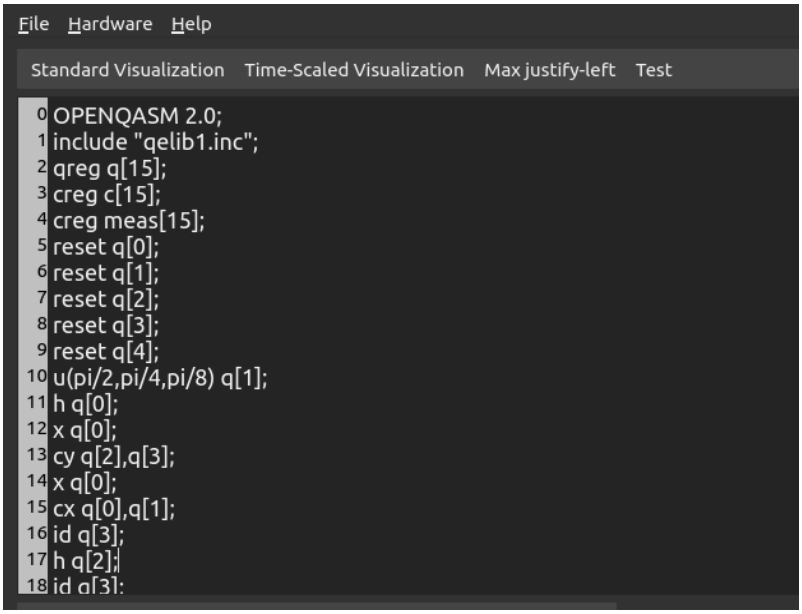

Figure 3.3: Screen capture, from the app, containing the code editor, visualization options and the toolbar for file management and hardware selection

Another important point is the ability to switch from one hardware to another at any point. This is particularly useful when observing the difference in the constraints that

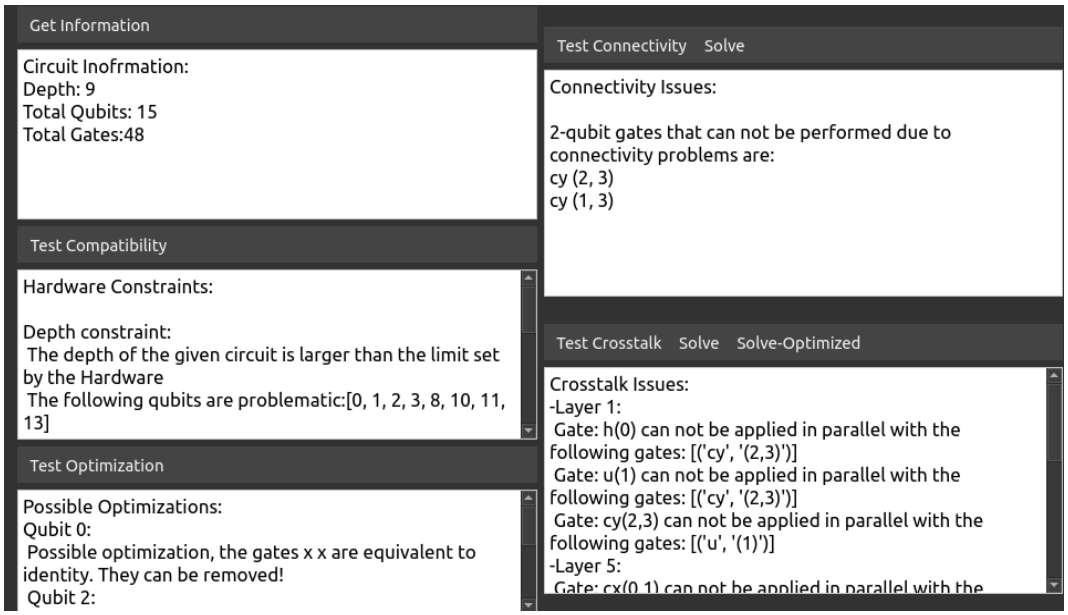

Figure 3.4: Screen capture, from the app, showing the different testing feedback and solution options

the platforms impose on the code.

If the platform does not give any feedback containing issues, then the QASM code can be run on the hardware of choice.

It is important to note that in this app, the tests and solutions support single qubit and two-qubit gates only. Gates operating on more than two qubits will be ignored.

3.2 Hardware Configurations

An essential part of our application, relies on the ability to store all relevant information of the quantum hardware. Finding a data structure that can store all those specifications and trying to make it as universal as possible was challenging. Our goal was to create a tool that can support almost all types of platforms. The problem we encountered is that each platform is built differently[22]. Some main differences are of course the qubits nature, layout and distance, which lead to other divergences between quantum computers such as qubit fidelity, error model, connectivity of the qubits and other characteristics[22].

We finally decided to store the information in a (.yaml) file or else known as "configuration file". Each (.yaml) file represents a hardware. The hardware characteristics are

contained in lists and dictionaries. We will now display the different hardware features we store in those files.

The first main characteristics [7, p. 54] are:

- Name of the platform
- Number of qubits supported
- Maximum circuit depth

Furthermore, at the beginning of this project we had access to information describing the $MQV\ Neutral - atoms\ platform$[1].

We noticed that it only supports certain sets of gates. This can also be the case for other platforms. To cover such cases, we added a list of supported gates to the configurations. Each gate in this list has the following attributes: *Name* and *Time*. The attribute *Time* represents the time it takes the hardware to execute the gate [14].

In our implementation, only *Name* and *Time* are being used. However, in the (.yaml) file, two additional attributes are generated: *Global* and $U3 - form$. These are possible features to be supported in the future. *Global* can be set to *True* or *False*. If it is set to *True*, this means this gate can be applied globally to many qubits at the same time. This is a special addition for the $MQV\ Neutral - atom\ platform$ where such gates are supported. For now, the value of this attribute does not influence any of the upcoming functionalities of the app. The $U3 - form$ is an attempt to generalize the gates that the app supports. Instead of checking the gates names to see if they are supported, we check their $U3 - form$. This can become useful if the hardware and the user are naming the gates differently. This is however not yet implemented, and the attribute is always set to *null*. The gate information will be presented as follows:

$$\textbf{List of gates: } \{gate : name \mid time \mid u3 - form \mid global\}$$

Moving on, we need to find a way to save the qubits layout in the form of a list with the ability to build the exact layout again, based on this list only. In order to do that, we create a dictionary called *Double Qubits*. The elements of this dictionary are the pairs of qubits of the quantum platform. For each pair of qubits, we save two main properties: *connectivity* [23] and *distance*. The *distance* is not relevant in our tool but the *connectivity* is a crucial constraint for code compatibility [7, p. 53-54]. We set *connectivity* to 1 if they are directly connected otherwise it is 0.

To these two attributes, we add the other most relevant criteria in our tool: *crosstalk* [25]. Each qubit or pair of qubits can have potential crosstalk partners [7, p. 53-54]. In

[1] These data stems are based on personal discussion with physicists from the Max Planck Institute for quantum optics

case a gate is applied to a qubit(s) and another gate is applied in parallel to one of their potential crosstalk partners, it results in a crosstalk condition [7, p. 53-54]. For each qubit pair we have two lists of crosstalk partners in *crosstalk*, *pairs* and *single*:

Double Qubits: $\{(index_1, index_2) : crosstalk : \{pairs : [],\ single : []\} \mid connectivity \mid distance\}$

We also need to implement a dictionary called *Single Qubit*. It contains the qubits individually and each has two attributes: $crosstalk(single\ and\ pairs)$ and $fidelity$

Single Qubit: $\{index : crosstalk : \{pairs : [],\ single : []\} \mid fidelity\}$

The $fidelity$ is not used in this tool, but it can be considered as a mean to minimize the error [7, Chapter 5].

Finally, we get a configuration file with all the relevant hardware information. It will enable us to figure out how to make quantum code work on the desired platform.

Note: You can generate an empty configuration file via calling the function $MakeConfig$ in the file $config.py$. This will generate a *Mock.yaml* file with all the lists and attributes. You can change the information inside it to fit your hardware.

3.3 Visualization

For a better understanding of quantum code, we can draw the quantum circuit and have a graphical representation of the gates as mentioned in section 2.1.3. For this purpose, the tool offers three visualization options:

- Standard Visualization
- Time Scaled visualization
- Max justify-left

Each of these options is based on the visualization of quantum circuits in Qiskit[19]. The original code is taken from the Qiskit source code[15]. However, it was largely modified by adding new features and options for the purpose of this tool.

3.3.1 Standard Visualization

The standard visualization is a way for the user to better visualize the gates that can be run in parallel or sequentially. This option does not take the execution time of the gates

into consideration and considers all gates the same. It displays the circuit similar to the standard Qiskit method. However, the standard visualization in Qiskit attributes to each gate box a length according to the text that it contains. Naturally some gates contain more text than others. An example would be the $U - Gate$. Besides its name, the gate-box contains its parameters. This makes it larger than an $X - Gate$. So in our tool, we want the standard visualization to have a unified width for all boxes. For that we set the width of all gate-boxes to the width of the largest gate in the quantum circuit. This will particularly make sense in the next visualization, where the width of the boxes is related to their execution time.

3.3.2 Time Scaled visualization

In this project, we decided to take the time of the gates into consideration when visualizing the quantum circuit. This gives the user an outlook over the runtime of the circuit, which can be useful when trying to optimize the code. Based on the times of the gates in the configuration file of the chosen hardware, a scaling dictionary will be produced. Each gate has a width proportional to its time, at the exception of gates which symbols do not include rectangular shaped boxes. In order to get the scaling dictionary, we implemented algorithm 1.

The function *GetScaleDictionary*, algorithm 1, returns the *scale dictionary* that will be used to set the width of the different gates in the quantum circuit. We get the width of a gate by multiplying the scale corresponding to the gate with the unified width explained in section 3.3.1.

Visual inconveniences might happen when there is a big difference between the times of the gates. This can lead to very large gate-boxes. We have a similar case in the $MQV\ Neutral - atoms\ platform$[2], where an R_z gate takes $100\,ns$ while an R_x gate can take $100,000\,ns$ to finish execution.

To avoid this problem we added the condition in line 13, which limits the scale to 8. It is, however, important to make it clear to the user that the width of such gates is not true to their real scale. We opted for the design that you can observe in the $U - gate$ in fig. 3.5. The blue middle part of the $U - gate$ indicates that its true width is larger than what is being visualized. It is also clear for the user, in fig. 3.5, that the $CY - gate$ takes more time in execution than the $X - gate$.

The next step is to set the width of each layer within the layers of gates to the width of the largest gate in that layer. This enables the different layers to contain all their gates without visually interfering with one another. Other changes also need to be

[2]These data stems are based on personal discussion with physicists from the Max Planck Institute for quantum optics

Algorithm 1 Get Scale Dictionary

Input: Configuration File
Output: $ScaleOfEachGate$
Initialize the scale dictionary: $ScaleOfEachGate \leftarrow \{\}$
if $TimeScale == false$ **then**
    $ScaleOfEachGate[''gateName''] \leftarrow 1$
else
    if $TimeScale == True$ **then**
        $minimumTime \leftarrow$ the minimum time of all gates in Configuration File
        for $gateName \in ConfigurationFile[''ListOfGates'']$ **do**
            Set the scale of each gate:
            $gateTime \leftarrow ConfigurationFile[''ListOfGates''][gateName][''Time'']$

$$GateScale \leftarrow \frac{gateTime}{minimumTime}$$

            if $GateScale > 8$ **then**
                $GateScale \leftarrow 8$
            end if
            $ScaleOfEachGate[''gatename''] \leftarrow GateScale$
        end for
    end if
end if

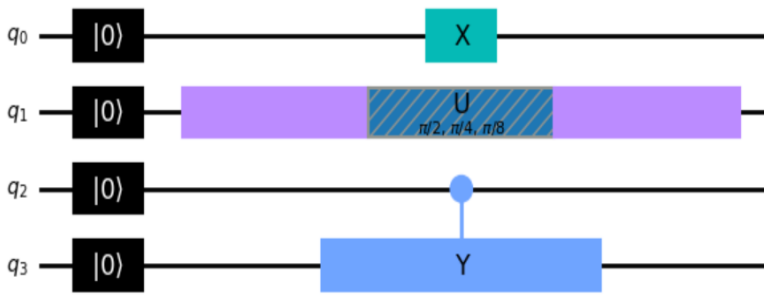

Figure 3.5: Time-Scaled Gate designs

made to the different functions of the file to adjust to the new method of visualizing the quantum circuit.

3.3.3 Max justify-left

The maximum justification is a way of emphasizing the parallelism of the gates by aligning them on the same vertical line, visually. This option can be applied to both standard and time-scaled visualizations. In order to achieve the desired visual outcome we move gates that can be executed in parallel into one layer.

To showcase the result of applying this feature, fig. 3.6 and fig. 3.7 represent the circuit before and after the application. You can notice in fig. 3.7 how the $CY-gate$, $I-gate$ and $H-gate$ are now on the same vertical line. However, in the same example, it can become unclear whether it is an $H-gate$ or a $CH-gate$ due to overlapping with the $CY-gate$. This is a clear disadvantage in this visualization. Therefore, it can not be used as a reference to identify clearly the different gates in the quantum circuit. Despite the shortcomings, it can be very useful in many cases. As an example, if the depth of the circuit is large, the user would benefit from making the visualization more compact.

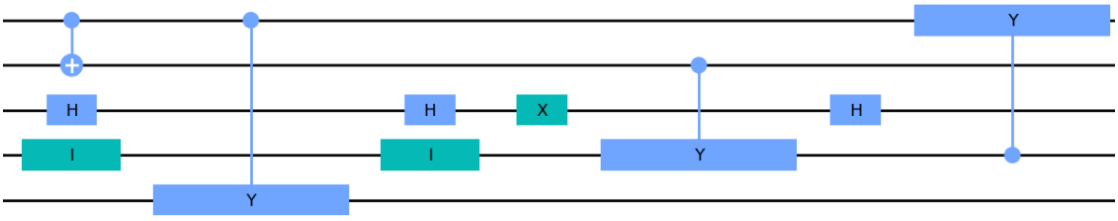

Figure 3.6: Gates before applying max justify-left

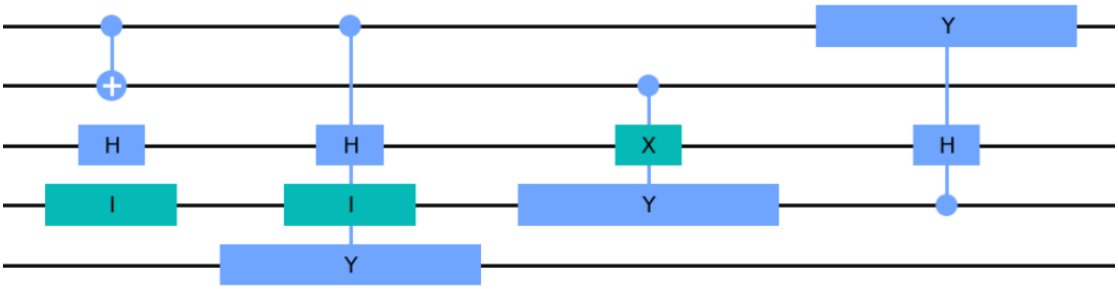

Figure 3.7: Gates after applying max justify-left

The function *MaximizeParallelism*, algorithm 2, has as input all the gates of the quantum circuit represented as Layers, see section 2.2. The loop in line 6 runs through the layers. For each gate in the layer on the right, we check if the qubits in the layer on the left are free. If the condition is met, we move the gate from the right, $Layer_{i+1}$, to the left, $Layer_i$. After removing gates from $Layer_{i+1}$, we check if it became empty and needs to be removed. The direction of the process, from right to left, is why the process is called left justification. We run an outer loop in line 3, which repeats the process until there is no change happening in the layers. This way, we ensure the maximized left-justification.

Despite the difference between the input and output layers, if we create a new quantum circuit out of the output, it will have the same semantics as the initial circuit. The reason is that we are only changing the order of gates which are independent of each other. We move a gate from one layer to another, only if the qubits on which the gate is acting are free. This ensures that the order of the gates acting on commun qubits does not change. Also, no gates are being removed from the circuit, they are only moved around. Therefore, the order and nature of the gates on each qubit is not compromised.

Algorithm 2 Maximize Parallelism

Input: Layers of gates
Output: Layers of gates after change
while *changed* **do**
    $changed \leftarrow False$
    $i \leftarrow len(Layers) - 2$
    while $i \geq 0$ **do**
        $Layer_i, Layer_{i+1} \in Layers$
        $occupiedQubits \leftarrow []$
        for $gate \in Layer_i$ **do**
            $occupiedQubits.add(gate.qubitsIndices)$
        end for
        for $gate \in Layer_{i+1}$ **do**
            if $gate.qubitsIndices \bigcap occupiedQubits = \emptyset$ **then**
                $move\ gate\ from\ Layer_{i+1}\ to\ Layer_i$
                $occupiedQubits.add(gate.qubitsIndices)$
                $changed \leftarrow True$
            end if
        end for
        if $Layer_{i+1} = \emptyset$ **then**
            $remove\ Layer_{i+1}\ from\ Layers$
        end if
        $i \leftarrow i - 1$
    end while
end while

4 Test Software-Hardware Compatibility

The main goal of this application is to enable the user to test the compatibility of their code with the chosen hardware. Therefore, we need to run a series of tests on the given QASM code and display the potential issues to the user. The main requirement for these tests is to be able to run them at any time, as many times as wanted and even after changing the code in the code editor. The tests can be run all at the same time through the option *Test* or separately via their own test options, which we will discuss next.

4.1 Circuit Information and General Compatibility

We start by collecting the circuit information and displaying it. The option *Get Information* fetches the depth, qubit count and number of gates of the quantum circuit. Some functions used for this purpose are: *qubitNumberCheck*, *circuitDepthCheck* and *getAvailibilityCheck*.

Now we can choose *Test Compatibility* to test the general compatibility of the input code with the hardware information (see section 3.2). We need to check if the depth of the circuit respects the maximum depth imposed by the hardware. If this constraint is not respected, we give the user a list of the qubits whose depth surpasses the limit.

Next we check if the number of qubits used in the circuit is below the limit. Last we go over the gates used in the code and make sure they are contained in the list of supported gates in the hardware. The user will get feedback containing the gates that are not supported.

4.2 Connectivity Test

A very essential condition for quantum code is that the multi-qubit gates need to be applied to connected qubits. Connected qubits are the pairs of qubits in the configuration file with the $connectivity = 1$. In this tool we only consider single-qubit and two-qubit gates. Once we choose the option *Test Connectivity*, a list of all qubit pairs that are connected is formed. Then it checks if the two-qubit gates use pairs from that list. The problematic gates, if there are any, are returned along with their corresponding qubits.

4.3 Crosstalk Test

Crosstalk detection is an important step in error correction and prevention in quantum computing. As mentioned in section 2.1.4, crosstalk occurs when gates are executed simultaneously on problematic qubits[24]. This condition depends on the qubits that are being used and their physical layout[24]. In this work we only detect crosstalk between two gates at a time.

To check whether two gates have crosstalk, we first check if they can be run in parallel, then we check if they are operating on qubits that can potentially have crosstalk between them.

There are different situations for crosstalk detection and each needs to be dealt with differently. We will explain them one by one using abstract examples where G_1 and G_2 are quantum gates and q_i, q_j, q_l *and* q_k are qubits.
For each example, if any of the conditions given as a list, is *True*, then the two gates are considered as crosstalk generators.

For the first situation, we have gate G_1 operating on qubits q_i and q_j, parallel to gate G_2 operating on q_k and q_l. The crosstalk conditions are the following:

- $(q_k, q_l) \in \text{DoubleQubits}[(q_j, q_i)][crosstalk][pairs]$
- $(q_j, q_i) \in \text{DoubleQubits}[(q_k, q_l)][crosstalk][pairs]$

The second case is when G_1 operates only on q_i and G_2 operates on q_k and q_l. The crosstalk requirements become as follows:

- $(q_k, q_l) \in \text{SingleQubit}[q_i][crosstalk][pairs]$
- $q_i \in \text{DoubleQubits}[(q_k, q_l)][crosstalk][single]$

The last case would be, G_1 operates on q_i and G_2 operates on q_k. The conditions for this situation are:

- $q_k \in \text{SingleQubit}[q_i][crosstalk][single]$
- $q_i \in \text{SingleQubit}[q_k][crosstalk][single]$

In our implementation, the function *CrosstalkCheck*, algorithm 3, returns a string containing all the information about the gates causing crosstalk in each layer. We start by calling the function *MaximizeParallelism*, algorithm 2. This call ensures that every layer of gates contains all the gates that can be run in parallel. This is important because we only detect crosstalk within the layer itself and not across layers.

The next step is to go through each layer separately. For each $gate_j$ inside the layer, we check if it has crosstalk with any other gate starting with $gate_{j+1}$. Restricting the

search to the latter part of the layer optimizes the search time. In case crosstalk is detected, the function $getFeedbackCrosstalk(gate_i, gate_j)$ is called. This function returns a string containing the information of $gate_i$ and $gate_j$. This information will be collected in the string *CrosstalkFeedback* which will be returned at the end. The purpose of collecting the information as a string is to be able to display it to the user when the *Test Crosstalk* option in the app is chosen.

Algorithm 3 Crosstalk Check

1: **Input**: the gates of the quantum circuit in form of Layers
2: **Output**: *CrosstalkFeedback*
3: $MaximizeParallelism(Layers)$ -algorithm 2-
4: $CrosstalkFeedback = ""$
5: **for** $Layer \in Layers$ **do**
6: $j \leftarrow 0$
7: **for** $gate_1 \in Layer$ **do**
8: $i \leftarrow j + 1$
9: **while** $i < len(Layer)$ **do**
10: $gate_2 \leftarrow Layer[i]$
11: **if** $(gate_1.qubitsIndices \in gate_2.potentialCrosstalkPartners)$ or $(gate_2.qubitsIndices \in gate_1.potentialCrosstalkPartners)$ **then**
12: $CrosstalkFeedback+ = getFeedbackCrosstalk(gate_1, gate_2)$
13: **end if**
14: $i \leftarrow i + 1$/watch
15: **end while**
16: $j \leftarrow j + 1$
17: **end for**
18: **end for**

4.4 Optimization Test

The Tool supports the detection of some gate patterns that can be optimized or removed. The user can test the circuit for potential optimizations by choosing *Test Optimization*. The following patterns[7, Chapter 14] can be detected by the app:

- $I - gate$ can be removed
- $HH \Leftrightarrow I - gate$
- $XX \Leftrightarrow I - gate$

- $ZZ \Leftrightarrow I - gate$
- $HXH \Leftrightarrow Z - gate$
- $HZH \Leftrightarrow X - gate$

The algorithm analyses all gates in their topological order and detects the previous gate sequences. The patterns found will then be returned as feedback. The optimizations, however, need to be done manually by the user.

5 Solve Software-Hardware Compatibility Issues

After having visualized and tested the quantum circuit for possible issues in chapter 3, the user now has to solve those problems in order to obtain a functional QASM code. The general compatibility problems described in section 4.1, such as gate types, number of qubits used and circuit depth, need to be solved manually by the user before moving on to the ones described in this chapter.

However, the user can choose not to solve those issues and still apply the built-in connectivity and crosstalk solutions available in the app. The algorithms take into consideration the possibility that the initial compatibility problems might not be solved and treat these special cases without generating errors.

5.1 Solve Connectivity Issue

The most common solution for connectivity problems is adding $SWAP-gates$ to the quantum circuit[12]. Applying $Swap(q_1, q_2)$ is equivalent to switching $|\psi\rangle$ and $|\phi\rangle$ which are respectively the current states of q_1 and q_2. We can see this switch of states in fig. 5.1 after applying the $SWAP-gate$.

After switching the values stored in the two qubits, we can operate on q_1 as if it were q_2 and vice versa. The mathematical implementation of the $Swap$ can be found in section 2.1.2.

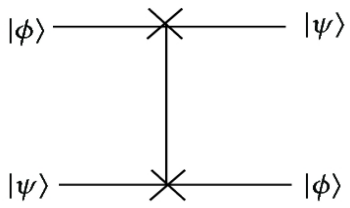

Figure 5.1: Swap gate [2]

5.1.1 Solution

Small Scale Solution

To better understand the logic of a *swap* and how it solves our connectivity problem, we will consider a small scale example. We have a two-qubit gate G that operates on q_1 and q_2. However, q_1 and q_2 are not connected, which results in a connectivity conflict. In this case, we need to find a bridge between the two qubits to establish a connection between them. A possible approach is to find a qubit q_3 that is connected to q_1 and q_2. The next step would be to apply $Swap(q_3, q_1)$. This results in storing the value collected in q_1 till now, in q_3. Finally, we can apply $G(q_3, q_2)$.

It is also very important to replace q_1 by q_3 and q_3 by q_1 in the rest of the circuit. We need to access the qubits of every gate that comes after the $Swap$ and change their indices in case they contain q_3 or q_1. The change of indices needs to be done manually because $Swap$ only affects the value stored in the physical qubits and does not switch the indices of the logical qubits in the circuit[1]. Adding a swap gate without switching the indices in the upcoming gates would result in corrupting the semantics of the initial circuit.

In conclusion:

$$G(q_1, q_2) \Leftrightarrow Swap(q_3, q_1),\ G(q_3, q_2)\ \textit{then switch}\ q_1\ \textit{and}\ q_3\ \textit{in the remaining circuit}$$

Another important point is that q_3 has to be connected to both q_1 and q_2 because the swap operation also imposes a connectivity constraint.

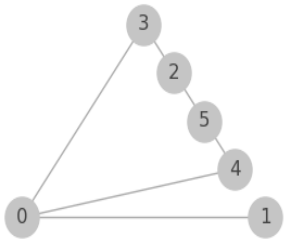

Figure 5.2: An example of an undirected connectivity graph of six qubits

Unfortunately, finding such a bridge, inter-qubit, may not be as simple as the previous example. In many cases there is no qubit which is directly connected to both q_1 and q_2. In fig. 5.2, we can observe an example of a qubit layout where this problem might occur. It visualizes the layout as a graph where the qubits are represented as nodes and

[1]To better understand the difference behind the logical and physical qubits check chapter 2

the edges are the existing connections between them. As a result finding a replacement is equivalent to finding a connected path from q_1 to q_2. We could also try to find a path from q_2 to q_1, and we will end up with the same results due to the connectivity graph being undirected. In our implementation, the starting qubit is the first index in $gate.qubits$, in this case it is q_1. We can see in fig. 5.2 that there is more than one path connecting q_1 and q_2:

1. $[1 \rightarrow 0 \rightarrow 3 \rightarrow 2]$
2. $[1 \rightarrow 0 \rightarrow 4 \rightarrow 5 \rightarrow 2]$.

Of course in order to optimize the number of $Swap - gates$ added to the circuit we look for the shortest path, in this case it is the first option. The reason we implement this optimization is because of the correlation between error and circuit depth[1]. Increasing the circuit depth leads to a decrease in the correctness of the output and increase in the noise[1].

Finally, we proceed to swap the nodes in $[1 \rightarrow 0 \rightarrow 3 \rightarrow 2]$ pair by pair till we connect both qubits. It is important to note that we would obtain a correct solution if we opt for the path: $[2 \rightarrow 3 \rightarrow 0 \rightarrow 1]$.

The first swapping is: $Swap(q_1, q_0)$. After inserting this gate we change any occurrence of q_1 with q_0 and q_0 with q_1. So now we operate on q_0 as if it were q_1 because it contains the old state of q_1. We resume the swapping with the second and last gate: $Swap(q_0, q_3)$ and we follow it with the indices update throughout the rest of the circuit like we did before. At the end of the two swaps, the initial state of q_1 is now in q_3. As a result we can apply the gate G on q_3 and q_2.

$$G(q_1, q_2) \Leftrightarrow Swap(q_1, q_0),\ Swap(q_0, q_3),\ G(q_3, q_2)\ then\ update\ qubits\ indices$$

General Solution

The configuration file of the hardware contains the connectivity information of all pairs of qubits. Using that information we start by creating a connectivity dictionary, that represents the graph of the qubits:

$$Graph = \{\textbf{Key} : qubit \mid \textbf{Value} : list\ of\ directly\ connected\ qubits\}$$

We then apply the function *addSwaps* (algorithm 4) to the quantum circuit in order to add the different swap gates and update the logical qubits in the circuit afterwards. The algorithm goes through the different gates of the quantum circuit until it finds a gate applied on qubits q_i and q_j that are not connected. Once such a gate is found, we call the function *getShortestPath* which returns the shortest path from q_i and q_j. This path is found running a standard breadth first search algorithm on the $Graph$[18].

Once we have the list of qubits connecting q_i and q_j, we start creating new $Swap - gates$. However, we can encounter a problem while creating swaps with the new indices found in the path. The user might choose to create a quantum circuit using a number of qubits lower than the number supported by the hardware. On the other hand the *getShortestPath* function considers all qubits supported by the hardware to find the path not just the ones supported by the circuit. Therefore, if the path from q_i to q_j contains an index n and the highest index supported by the circuit is m with $m < n$, applying a $swap - gate$ with q_n is problematic. To make it feasible, we need to make the quantum register of the circuit larger to support the use of the qubit n. The minimum new size of the quantum register has to be $n + 1$. We check this condition before creating the swap gates. We make sure that the quantum register is larger than the maximum index in the path.

After inserting each swap, we run through the rest of the current *Layer* of gates then the rest of the *Layers* to switch the indices of qubits that have been swapped. There is another very important step, not mentioned in algorithm 4, that needs to be done after finishing adding the swaps and changing the circuit. We go through all the instructions from the beginning and update the quantum register they are using to the last version of the *circuit.quantumRegister*. At the end we have new quantum circuit with no connectivity conflicts.

An important note for the user is to apply the connectivity solution on a circuit that respects the maximum number of qubits supported by the hardware. In case the circuit contains gates operating on qubits with indices higher than the limit of the hardware, it is impossible to find a path connecting the qubits as they don't exist in the connectivity graph of the hardware. The algorithm will not cause any errors but will simply ignore those gates and not solve the related issue.

5.1.2 Limitations

In this approach to connectivity, we only focused on solving the problem without taking the circuit depth into consideration. The algorithm 4 does not aim to add a minimal number of swaps.

Another point to be criticized concerns the algorithm that provides the shortest path from one qubit to another. The algorithm does not try to limit the qubits in the path to those supported by the circuit, which can lead to an increased circuit width.

5.2 Solve Crosstalk

As discussed in section 4.3, operating on certain qubits at the same time can create noise between them and cause errors in the results. In our implementation we are bound

Algorithm 4 Add Swaps

Input: Graph, Layers of Gates
Output: Layers of gates after adding the swap gates and updating the qubit indices
for $Layer \in Layers$ **do**
    $j = 0$
    while $j < length(Layer)$ **do**
        $gate \leftarrow Layer[j]$
        if ($gate$ is a 2-qubit gate) and ($gate.qubits$ are not connected) **then**
            path $\leftarrow getShortestPath(Graph, gate.q_0, gate.q_1)$
            if path exists **then**
                size $\leftarrow$ circuit.quantumRegister.size
                highestIndex $\leftarrow$ Max(path)
                if $highestIndex \geq size$ **then**
                    Change the size of the quantum register of the circuit to $size \leftarrow highestIndex + 1$
                end if
                for $k \in [0..length(path) - 3]$ **do**
                    $swap \leftarrow createSwapGate(gate, circuit, path[k], path[k+1])$
                    Insert the new $swap$ gate in current $Layer$ in position j
                    Update the indices of qubits of all the gates in $Layer$ starting from position $j + 1$
                    Update the indices of qubits in all Layers after the current $Layer$
                    $j \leftarrow j + 1$
                end for
            end if
        end if
        $j \leftarrow j + 1$
    end while
end for

to solve crosstalk only by changing the QASM code. An intuitive solution for this issue can be to cancel the parallelism and serialize the operations causing the crosstalk [11]. There are of course other approaches to the problem. We will mention some of them in section 6.2. Meanwhile, our implementation is based on the intuitive approach mentioned.

We have briefly discussed in section 2.3 the approach that Qiskit uses to solve this issue [20]. The logic found in the Qiskit, "separating the problematic gates with barriers", will also be found in our implementation. Although, the implementation of the solution and the resulting circuit will be different from the Qiskit approach. We implemented two different ways to solve the crosstalk. We will explain the first solution in this section and the second one in section 5.3.

The algorithm 5 represents the first approach. It contains the different steps leading to the final crosstalk-free circuit. The general idea is: If two parallel gates cause crosstalk, they should not be in the same layer. At the end, each layer is crosstalk-free. We then separate the different layers with global barriers making sure there is no interference between them in runtime. In the following subsection we will go through the different steps of the algorithm and explain the logic behind them.

Algorithm 5 Crosstalk Solution

1: **Input**: Circuit, Layers, Configuration Data
2: **Output**: New Circuit, New Layers
3: $MaximizeParallelism(Layers)$ -algorithm 2-
4: NewLayers $\leftarrow$ *CrosstalkRemoval* -algorithm 6-
5: $MaximizeParallelismNoCrosstalk(NewLayers)$
6: Barriers $\leftarrow$ $generateBarriers(circuit, NewLayers)$ -algorithm 7-
7: Create a *NewCircuit* using the *NewLayers* and the *Barriers*, creating a separation between the layers.

5.2.1 Solution

Step 1: Maximize Parallelism

Our initial input are layers of gates. We start by calling the function $MaximizeParallelism(Layers)$ (line 3), which we used in section 3.3.3. This will make sure that we have compact layers and will lead to a high number of parallel gates and reduced circuit depth. This function call is particularly important because at the end we will separate the layers with barriers. That separation will lead to forcing a serialization of the gates in the different layers. Therefore, it is better that we make sure we have the maximum

number of gates possible in each layer before we start the separation. In summary, we increase the parallelism inside the layers to compensate for their later serialization.

Step 2: Cancel Crosstalk

Now that we have our compact layers of gates, we call the function *CrosstalkRemoval* (line 4). We iterate through the *Layers*, and we go through each gate of $Layer_i \in Layers$. For each $gate_j \in Layer_i$, we check if it has any crosstalk conflict with any $gate_k \in Layer_i$ with $k > j$. We check the existence of crosstalk using the lists of potential crosstalk partners for the qubits:

- $gate_j.qubits \in gate_k.potentialCrosstalkPartners$
- $gate_k.qubits \in gate_j.potentialCrosstalkPartners$.

These lists are found in the configuration data of the hardware as explained in section 3.2.

If we find a crosstalk issue between $gate_j$ and $gate_k$, then we move $gate_k$ from $Layer_i$ to $Layer_{i+1}$. However, $Layer_{i+1}$ might have other gates operating on $gate_k.qubits$. In this case, we insert an empty layer [] in *Layers* at index $i+1$. To this new Layer we add $gate_k$. At the end we remove $gate_k$ from $Layer_i$. Once we finish moving gates around, we remove all resulting empty layers. After running *CrosstalkRemoval*, we obtain new layers, each containing parallel gates that can be executed crosstalk-free.

To better understand the algorithm we observe how it performs on a simple quantum circuit. Initially we have:

$$Layers = [Layer_0 : [Y(4), Z(2), X(0), X(3), X(1)];\ Layer_1 : [Y(1)]]$$

The left side of fig. 5.3 represents a visualization of this initial circuit. The barriers separating the layers in the example are for visualization purpose only and do not represent real barriers in the code (not to be confused with the barriers we will add to the code in Step 4).

The crosstalk information we get from the configuration file of the hardware is the following:

$$\{(qubit_4 : [0,1,2,3]), (qubit_2 : [1]), (qubit_3 : [2])\}$$

The first gate we start with, in $Layer_0$, is $Y(4)$. This gate has crosstalk with all the gates in $Layer_0$ according to the information above. As a result, we start moving the following gates to $Layer_1 : Z(2)$, $X(0)$ and $X(3)$. The order in which we move the gates depends on their order inside $Layer_0$. The result of this action is presented on the right side of fig. 5.3.

Now we check $Y(4)$ and $X(1)$, the last gate in $Layer_0$. We need to move $X(1)$ to $Layer_1$. However, it is impossible due to the existence of the gate $Y(1)$, which occupies $qubit_1$. Therefore, we create a new layer at index 1 and move $X(1)$ to this new layer. We can notice in fig. 5.4 that now we have 3 *Layers*. The algorithm continues iterating through $Layer_1$ and $Layer_2$ moving the gates around to solve crosstalk conflicts. This was a sample solution to better visualize the way the algorithm works.

Figure 5.3: Moving the gates: X(0); Z(2) and X(3) to next layer due to crosstalk

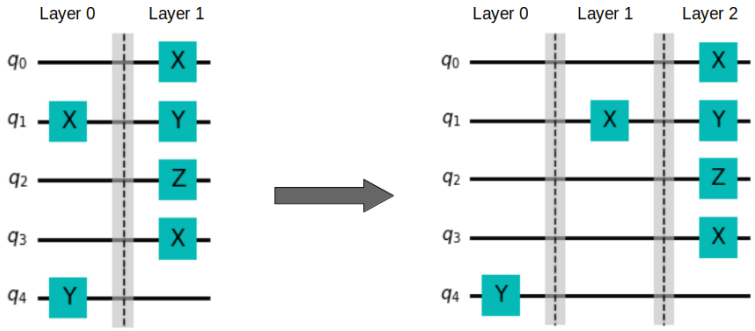

Figure 5.4: Insert X(1) in a new layer at index 1 due to crosstalk

Step 3: Maximize Parallelism -Avoiding Crosstalk-

In Step 2 (section 5.2.1), solving the crosstalk conflicts was our only priority. However, a possible side effect of the solution is having multiple layers that can be merged together and run in parallel. For this purpose we wrote an algorithm very similar to $MaximizeParallelism(Layers)$ (algorithm 2) called $MaximizeParallelismNoCrosstalk$. The new algorithm does the same function as the old version except that it does a crosstalk check before moving a gate. For a *gate* to be moved from $Layer_{i+1}$ to $Layer_i$,

two conditions need to be met instead of one.
Condition 1: In $Layer_i$, the qubits on which the $gate$ operates need to be free
Condition 2: The $gate$ does not result in crosstalk with the gates of $Layer_i$
Similar to $MaximizeParallelism(Layers)$, we will repeat this process until there is no gate that can moved to the left.

An example of a situation where this is needed is in fig. 5.5. All three gates: $X(1), X(0)$ *and* $X(3)$ can be run in parallel without causing crosstalk according to the hardware information in section 5.2.1. Therefore, $Layer_2$ and $Layer_1$ are merged together after running $MaximizeParallelismNoCrosstalk$. On the other hand, the gate $Z(2)$ was not pushed to the previous layer despite the available space. This is due to the crosstalk between $(qubit_1, qubit_2)$ and also $(qubit_3, qubit_2)$. This example perfectly explains the difference between the original algorithm (algorithm 2) and the new one.

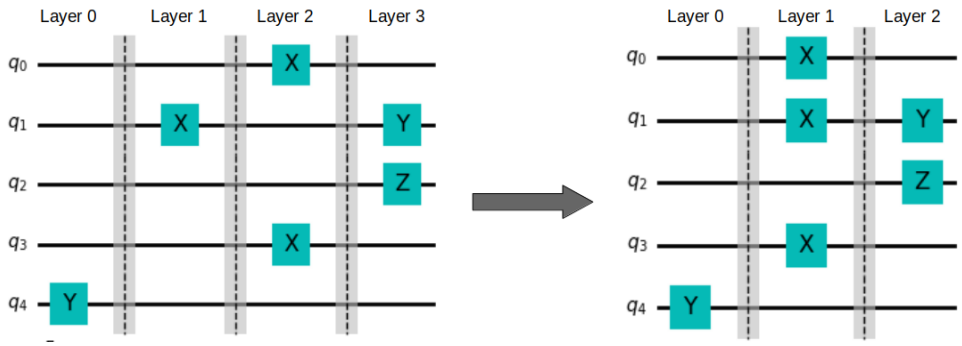

Figure 5.5: Last Iteration of Crosstalk Removal

Step 4: Generate New Circuit With Barriers

This final step plays a major role in the solution. Out of step 1, 2 and 3, we obtain a data structure representing the gates of our quantum circuit. However, if we translate the data structure directly to a quantum circuit we will end up having our initial circuit with unsolved crosstalk issues. This is because the layers are a theoretical way of separating the different quantum gates to better manipulate them. If we want to preserve the new crosstalk free layers we obtained, we need to create a tangible separation between them in the code.

A simple and efficient way to do that is to apply global barriers at the end of each layer. Global barriers are barriers that are applied to all the qubits of the circuit. It is a way of forcing the hardware to execute all the gates in one layer first, then move to the next layer. Because we separate the gates causing crosstalk problems into different

layers, we are certain that those gates can never be run in parallel due to the global barriers[11].

The function $generateBarriers(circuit, NewLayers)$ (algorithm 7) generates those barriers between layers. It also avoids creating double barriers in case the circuit initially has global barriers. This condition is important especially because in the app $Solve\,Crosstalk$ can be called an unlimited number of times. Therefore, the barriers are to be added only when needed.

Finally, we obtain a new circuit out of the layers from step 3 and the barriers we just generated. We return this circuit in form of QASM code displayed in the text editor in the app.

5.2.2 Limitations

Although the presented algorithm solves the initial problem, it is not the best approach. This solution does not consider the time of the different gates. It allows only one gate per qubit in each layer. Therefore, when one gate finishes operating first, its qubits will remain free until the next layer is reached. This is caused by the global barriers we apply to the circuit. So there is a chance that we are increasing the total runtime of the circuit, which can lead to decoherence [7, chapter 5].

Another point is that we don't consider gate commutation which can lead to a reduced number of layers. There are different optimization passes [8] that can be run on the circuit before applying the global barriers and forcing a certain scheduling on the hardware.

5.2.3 Correctness

Two main things that are needed to prove the correctness of this solution:

1. There are no crosstalk issues in the final circuit
2. The semantics of the initial circuit is preserved

Starting with crosstalk, we have explained in a detailed manner throughout Step 1, 2 and 3, in section 5.2, why this solution actually works. We also explained that adding the global barriers at the end plays the most important role in isolating the layers in the actual code. This prevents simultaneous execution of problematic gates.

Moving on to the semantics, our only concern in this algorithm is conserving the relative order of the gates when rearranging them, as no other changes are being made to the circuit.

We move our gates inside the layers in two directions. We can move gates from right to left, and we call these, the parallelism-maximization algorithms. Those algorithms,

however, only push all gates to the left in the order they are initially in. Basically, if the qubit in the layer on the left is occupied, we quit the "move" and go to the next gate. Also, we only move gates between two successive layers. This means there is no situation where a gate from $Layer_{i+2}$ is moved to $Layer_i$ while there is a gate in $Layer_{i+1}$ that must come before.

The second direction we move gates in, is from left to right. This is during the crosstalk removal (algorithm 6). This does not cause a problem, because we are separating gates that can be run in parallel, so there is no order involved. Also, we are pushing gates to a directly subsequent layer to the right if there is space or insert it right before the next layer. In both situations the order is conserved.

5.3 Solve Crosstalk - Time Dependent

Part of the hardware specifications is the time it takes the hardware to execute the different gates. This fact can be utilized to optimize the circuit runtime[9]. We can try to maximize the parallelism of the gates based on their single operating times. A simple example would be: A gate G_1 with $Time = 100t$ can be parallel to two serialized G_2 gates with $Time = 30t$ with t representing a time unit.

The main method (algorithm 11) of our 2^{nd} approach, is actually a mixture between crosstalk solving and scheduling. This solution generates a circuit where each layer contains high parallelism and efficient use of the different runtimes of the gates.

In this section, we will introduce a new concept of *Layers*. Contrary to what we explained in chapter 2, a Layer in this approach can contain serialized gates as well as parallel gates. After applying this solution to the initial layers, we still end up with new layers that are crosstalk-free, however, not all gates inside one layer are parallel.

In fig. 5.6 we can observe an example of what a layer can look like after applying the second approach.

In this example, gate SX takes more time than gate R_z. We can fit multiple R_z gates during the runtime of one SX gate. This is noticeable even from the proportionality of the gates to each other as explained in section 3.3.2. We can see that $R_z(2)$ and $R_z(3)$ would be run after each other due to the barrier between them. Those barriers are a mean of preventing crosstalk between the gates.

5.3.1 Solution

We will now explain in details the different algorithms that make up the solution and the special cases that we had to pay attention to. The different steps are displayed in algorithm 8. We will divide them in 4 steps and discuss them separately.

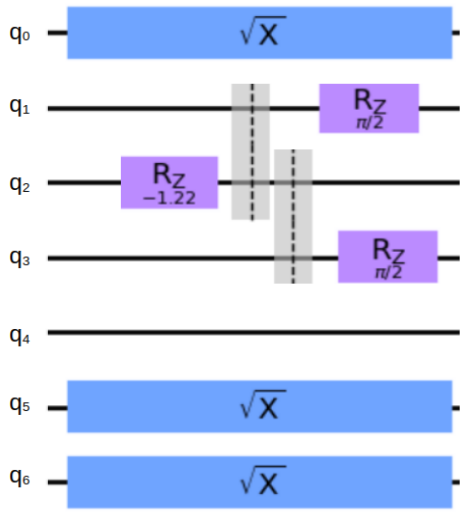

Figure 5.6: Layer after time-optimized crosstalk solution

Step 1: Run the First Approach

We start by applying the 3 algorithms that were used in section 5.2 to our initial layers:

- $MaximizeParallelism(Layers)$ -algorithm 2-
- $CrosstalkRemoval(Layers)$ -algorithm 6-
- $MaximizeParallelismNoCrosstalk(Layers)$

The function-calls above were discussed thoroughly in section 5.2. These will give us crosstalk-free compact layers. Those will be used in the next steps.

Step 2: Necessary Initialization

Before we start moving the gates around, we have a couple of dictionaries to initialize. These will be used later in the main algorithm.

The first dictionary we will need is the Time Dictionary, and it contains the times of all gates in the quantum circuit:

$$TimeDict = \{\textbf{Key} : GateName \mapsto \textbf{Value} : Time\}$$

We also look for the minimum time of all gates and assign it to gates in the circuit that are not supported by the quantum hardware. This is an error prevention in case

the user does not solve the first hardware compatibility issues, see section 4.1, before solving the crosstalk.

Next we need to set the width of each layer in a *LayersWidth* list.

$$LayersWidth[i] = Width\,of\ Layer_i = Max(Times\,of\ the\ gates\ in\ Layer_i)$$

The layer's width corresponds to the gate in the layer that takes the most time to execute. This width is the maximum execution time of all gates inside the layer. In other words, the totality of gates acting on each qubit in $Layer_i$ need to finish running in less or equal time than the width of $Layer_i$.

To better visualize this constraint we observe the two layers in fig. 5.7. In the Time Dictionary, the time of gate Y is 500 *time units* and the time of X is 100 *time units*. As a consequence the width of $Layer_0$ is set to 500. Therefore, in $Layer_0$ we can only fit five $X-gates$ consecutively and the 6^{th} $X-gate$ stays in $Layer_1$.

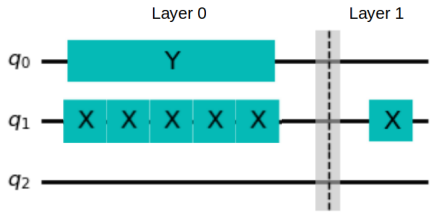

Figure 5.7: Layer width as a time constraint

The last initialization is the time available per qubit in each layer (algorithm 9). Previously we described the layer's width as the upper bound for execution time in the layer. However, this upper bound is not always reached due to the different timings of the gates. The free time for each qubit inside a layer where no gate is operating on it, is stored in *AvailableTime*. The *AvailableTime* is a list of dictionaries, each representing a layer. Inside each dictionary we have an association of (qubit, time).

$$AvailableTime[i][qubit_j] \leftarrow LayersWidth[i] - \Sigma\ time\ of\ gate\ in\ Layer_i\ acting\ on\ qubit_j$$

This is the time which will be utilized in the next steps to serialize gates that can be run consecutively within the time constraint. After Initializing the *AvailableTime* dictionary for the sample circuit in fig. 5.8, we get:

$$AvailableTime = [\{(qubit_0, 0), (qubit_1, 400), (qubit_2, 500)\}]$$

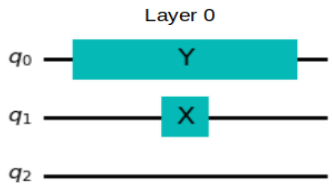

Figure 5.8: Example of different time availability on $qubit_0$, $qubit_1$ and $qubit_2$

Step 3: Removing the Crosstalk

In this step we call the function "Crosstalk Removal-Time Dependent-" implemented in algorithm 11. This algorithm is composed of multiple case-checks. We will explain, when each case can occur and how it will be treated.

Free Gaps Dictionary:
We create a dictionary called *FreeGapsPerQubit*. This is a similar dictionary to *AvailableTime*. We initialize the values corresponding to the different qubits in the different layers to 0: $FreeGapsPerQubit[i][qubit_j] \leftarrow 0$ with i an index of a layer and j index of a qubit.

Qubit Availability Dictionary:

Now we iterate through the different layers, and for each $Layer_i$ we check whether we can move gates from $Layer_j$ with $j > i$. For each $Layer_i$ we create a *QubitAvailibility* dictionary: $QubitAvailibility = \{$**Key:***qubit* $\mapsto$ **Value:***True or False*$\}$.

The logic behind the *QubitAvailibility* dictionary is to preserve the order of the gates. So if a gate $G(qubit_k)$ can not be moved into $Layer_i$, we set $qubit_k$ in *QubitAvailibility* to *False*. Now any gate that acts on $qubit_k$ and comes after $G(qubit_k)$ in topological order can not be moved to $Layer_i$ even if there is enough space or other conditions are met. This is because setting the availability of $qubit_k$ to *False* indicates that there is a gate that came before and is blocking this qubit.

In conclusion, the value corresponding to each qubit represents a permission (or not) to move an outside gate operating on that qubit, to the current layer. All the values in *QubitAvailibility* are originally set to *True* when a layer is first entered. Then, every time an outside gate can not be moved into the current layer, we set the qubits corresponding to that gate to *False* in *QubitAvailibility*.

In fig. 5.9, we can see that on $qubit_2$, there is space next to gate Z in $Layer_0$. Theoretically gate X can fit in there. However, the gate SX came before it in $Layer_1$ and has

already blocked $qubit_2$. Therefore, gate X can not be moved from $Layer_2$ to $Layer_0$. The gate-blocking prevents gates from jumping before other gates that normally need to be executed before them. This way we preserve the semantics of the quantum circuit.

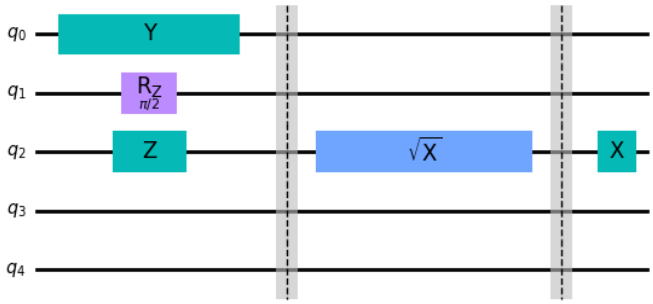

Figure 5.9: Example of gates blocking each other via *QubitAvailibility*

Case 1:
We start by fixing a $Layer_i$ to which we want to move all possible gates from all layers that come after it. Suppose $gate_0(q_h)$ is a gate that we want to add to $Layer_i$ from $Layer_j$ with $j > i$.

First, we check the availability condition. As we discussed previously, $QubitAvailibility[q_h]$ need to be *True* to be able to move the gate. In case we have a multi-qubit gate acting on more than a qubit, all those qubits need to be set to *True* in order to enable the move. If any of the qubits on which the gate operates is set to *False* in *QubitAvailibility*, the move is cancelled. We then proceed to set all *gate.qubits* in *QubitAvailibility* to *False* and continue with the next gate.

Case 2:
Next we check if there is enough space in $Layer_i$ to add $gate_0(q_h)$. For that, we check if $FreeGapsPerQubit[i][q_h] \geq gate_0.time$ or $AvailableTime[i][q_h] \geq gate_0.time$. If none of these two conditions is fulfilled, $gate_0$ can not be added to $Layer_i$, we set q_h in *QubitAvailibility* to *False* and move on to the next gate.

Case 3:
We suppose that there is enough space and the qubits are available. We go check if the free gaps have enough space to fit $gate_0(q_h)$. We left it to this point to explain what is the meaning behind those gaps. In order to understand what a gap is, we consider the example in fig. 5.10. The $barrier(q_1, q_2, q_3)$ is acting as a crosstalk solution because here we have crosstalk (q_2,q_3) and (q_2,q_1). How and when to add a barrier will be discussed

in the next steps of this algorithm. We bring this up here because adding the barrier can result in a wasted space between the barrier and the gates before it and we want to exploit that space to increase parallelism.

As we see in the example, the gate Z takes longer to execute than the gate X. Eventually gate $X(3)$ will finish execution, and we have to wait for gate $Z(1)$ to finish due to the barrier. As a consequence, $qubit_3$ was free for a period of time equal to $Z.time - X.time$. This time slot is what we refer to as a gap. These gaps are a result of adding barriers. We can keep track of them and add gates in those time slots.

As an example, we consider a gate operating on $qubit_3$. This gate comes after the gate $X(2)$ in terms of topological order, it checks the previously mentioned conditions and can fit in $FreeGapsPerQubit[0][qubit_3]$. This gate will be inserted in the position right after $X(3)$. We can see in fig. 5.11 how we could fit another $X(3)$ gate before the barrier.

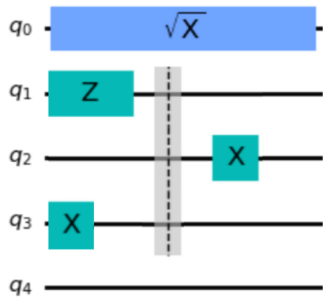

Figure 5.10: Gap space between barrier and $X(3)$

It is very important to know how we find the index where we insert the gate. For this we need to go back to our initial example where we are trying to move $gate_0(q_h)$ in $Layer_i$. It is not enough to find free gaps for all of $gate_0.qubits$, in this case q_h, where the gate can fit. We need to find a gate in the initial $Layer_i$ that operates on exactly the same qubits as $gate_0$. So if we had $gate_1(q_h, q_k)$, we do not insert $gate_0(q_h)$ next to it. The reason for this is that, when inserting a gate in a gap we do not check for crosstalk. This is due to the fact that the kind of crosstalk we are dealing with in this approach depends on the qubits not the nature of the gates.

So if we have a $gate_2$ in $Layer_i$ that operates on exactly the same qubits as $gate_0$ then we can be certain that any kind of crosstalk for those qubits has been already treated. By adding $gate_0$ directly consecutive to $gate_2$, we make sure that any barrier that is applied to $gate_2$ will be also applied to $gate_0$. This is also shown in fig. 5.11. By adding the new $X(3)$ on the left side of the barrier, we have no crosstalk with $X(2)$.

If the $gate_0$ can be moved in the gap after $gate_2$ in $Layer_i$, we need to update the gap time: $FreeGapsPerQubit[i][q_h] \leftarrow FreeGapsPerQubit[i][q_h] - gate_0.time$. We then move

on to the next gate.

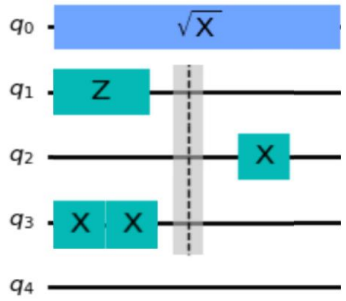

Figure 5.11: Inserting a new gate $X(3)$ in the gap before the barrier

Case 4:
In case we were unable to insert a $gate_0(q_h)$ in a free gap in $Layer_i$, we close all gaps corresponding to $gate_0.qubits$: $FreeGapsPerQubit[i][q_h] \leftarrow 0$. We do this because if $gate_0$ can not fit in the gap, no subsequent gate operating on the same qubits is allowed be moved in there.

Now the only option left is to append $gate_0$ at the end of $Layer_i$, if possible. Therefore, we start with checking if there is any crosstalk between $gate_0$ and any other gate in $Layer_i$. For this, we call a function $BarrierToCreate$.

Inside the function $BarrierToCreate$, we check for crosstalk. We start by creating a list called $BarrierIndices \leftarrow []$. Every time we find a problematic gate, $gate_p$, in $Layer_i$ we check if there is already a barrier preventing $gate_p$ and $gate_0$ from running in parallel. If this barrier already exists, we consider the crosstalk issue solved and do not require an additional barrier.

However, in case we need to add a barrier before adding $gate_0$, we add $gate_0.qubits$ and $gate_p.qubits$ to $BarrierIndices$. We do this for all the gates in $Layer_i$. At the end we return $BarrierIndices$. The returned list represents the barrier that needs to be implemented before moving $gate_0$ to $Layer_i$

Back to the main algorithm (algorithm 11), we check if $BarrierIndices$ is empty. This means we can just move $gate_0(q_h)$ to $Layer_i$ without adding any barriers. After that, we update the time available on q_h, and we move on to the next gate: $AvailableTime[i][qubit_h] \leftarrow AvailableTime[i][qubit_h] - gate_0.time$.

Case 5:
Another case is when $BarrierIndices$ is not empty. Meaning, we have to add a barrier to $Layer_i$, if we want to move $gate_0$ there. However, after adding a barrier there is a

possibility that we will not have enough space for $gate_0$. This is caused by the updates to *AvailableTime* and *FreeGapsPerQubit* that happen as a consequence of adding a barrier to a layer.

This is where algorithm 10 "Update Time And Gaps Before Adding Crosstalk Barrier" is useful. We run this algorithm before moving the barrier and the gate. This algorithm checks if there will be enough space for $gate_0$ after adding the barrier. If there is not enough space, it does not change anything and returns *False*. If there is space available, it updates *AvailableTime* and *FreeGapsPerQubit* and returns *True*.

In case algorithm 10 returns *True*, we proceed with creating a barrier with the indices in *BarrierIndices*. Next, we move the new barrier then $gate_0$ to $Layer_i$. Otherwise, we leave $gate_0$ in its original layer, and we do not add any barriers to $Layer_i$. In case we don't move the gate, it is important to go and set q_h in *QubitAvailibility* to *False*. We finally move on to the next gate.

To better understand how algorithm 10 works, we will discuss the circuit presented in fig. 5.12. In the hardware, we run the code on, we have:

$$TimeDict = \{(SX, 600)|(Y, 500)|(X, 100)|(Z, 200)\}$$

On the left we have the initial $Layer_0$. $Layer_0$ contains the following gates: $SX(0)$, $Y(2)$ and $X(4)$. $LayersWidth[0] \leftarrow SX.time$ given that the gate SX takes the most time. We have a $Z(3)$ gate that we want to move to $Layer_0$. However, we have crosstalk issues in (q_2, q_3) and (q_3, q_4). In this case, we need to add a $barrier(q_2, q_3, q_4)$.

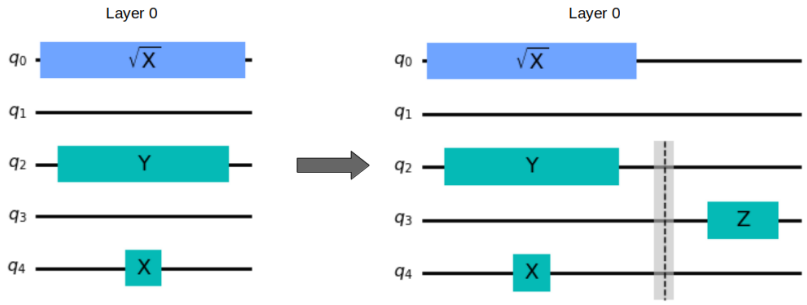

Figure 5.12: Example of a problematic crosstalk-cancelling barrier

At first sight, we notice that $qubit_3$ is empty. We have $AvailableTime[0][qubit_3] = 600 > Z.time$. This however is not the real time available for the gate $Z(3)$. We can see in fig. 5.12 on the right side, how when adding a barrier we actually reduce the time available for $Z(3)$. This is due to the fact that the barrier forces $Z(3)$ to wait for $Y(2)$ and $X(4)$ to finish. This serialization is necessary to avoid crosstalk. For this reason we need to calculate a new $AvailableTime[0][qubit_3]$.

We start by finding the minimum time available of all qubits $\in$ *BarrierIndices*. In our case, it is $LayersWidth[0] - Y.time = 100$. This represents the new available time of all qubits $\in$ *BarrierIndices*.

Now we compare the new available time to the time of the gate we want to add. If the gate takes more time than the time available, it will lead to surpassing the time limit set through the Layer's width. In this case we do not move the gate, nor add the barrier, and we do not update the *AvailableTime* with the new values. This is what happens in this example: $Z.time = 200 > 100$. We can see on the right of fig. 5.12 that if we move $Z(3)$, it exceeds the layer's width.

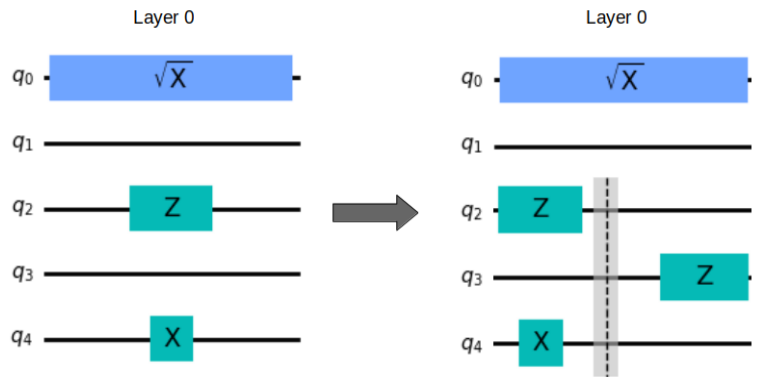

Figure 5.13: Example of a successful gate mitigation with the proper crosstalk-cancelling barrier

Now, we consider an example where we can move the gate and the corresponding barrier into the layer. The fig. 5.13 is very similar to the example in fig. 5.12 to the exception to the gate $Y(2)$ which was replaced with a shorter $Z(2)$ gate.

In this situation, the minimum time available of all qubits $\in$ *BarrierIndices* is $LayersWidth[0] - Z.time = 400$. There is enough time available for $Z(3)$ after adding the barrier. We proceed with updating the gaps then the time available for all qubits in *barrier.qubits*.

We start setting the gaps first because we need the old *AvailableTime* values. Furthermore, we set the gaps corresponding to the qubits of the added gate to 0, in this case it is $Z(3)$.
This is because the gate is located after the barrier. This means it blocks every gate coming after it, which can fit in the gap before the barrier. So $FreeGapsPerQubit[0][q_3] = 0$.

Moreover, we treat the qubits that are contained in *BarrierIndices*, but the moved gate do not operate on them, in this case we have q_2 and q_4:
$FreeGapsPerQubit[0][qubit] = AvailableTime[0][qubit] - minimum\ available\ time$.
In this example:

$FreeGapsPerQubit[0][q_2] = 400 - 400 = 0$ and $FreeGapsPerQubit[0][q_4] = 500 - 400 = 100$.

Now we assign for all qubits $\in$ the barrier, $AvailableTime[0][qubit] = 400$. Additionally, we subtract the time of the added gate from its available time. As a result, $AvailableTime[0][q_3] = 400 - 200 = 200$.

Finally, the time available and the free gaps have been updated and the gate and barrier were successfully moved. Now we can proceed with the next gate.

Clean Up

After reaching the end of every $Layer_j$, from which we have been moving gates to a prior $Layer_i$, we need to clean up $Layer_j$. We call an update function which updates *AvailableTime* and *LayersWidth* according to the new state of $Layer_j$. In case a layer has become empty then we remove the layer and all information corresponding to it in *AvailableTime* and *LayersWidth*.

Step 4: Generate New Circuit

After running algorithm 11 "Crosstalk Removal-Time Dependent-", we obtain crosstalk-free layers with high runtime optimization. Now, to finish off this algorithm, we generate the barriers separating the layers to preserve their structure. Then, we generate a new circuit out of the new layers and the barriers.

5.3.2 Limitations

In this approach we actually present a way of scheduling the different gates and how they can be run on the hardware. This time-aware approach aims to optimize the runtime of the circuit. However, it does not necessary give the optimal solution. To find the optimal solution, we need to consider all feasible permutations of the gates while conserving there topological order. Then, from all the possibilities, we choose the one that has the least runtime.

This algorithm can be further improved to find the best combination for the gates to run in the least period of time possible. A specific condition in the algorithm that can cause a non-optimal solution is the time availability constraint. If the available time on a qubit in a layer is slightly less than the execution time of a gate, we do not move the gate. However, sometimes, that difference is so small that it is better to move the gate to the left than to leave it in the layer to the right and increase the global runtime of the circuit. In this case, the algorithm will not give the most optimized solution.

5.3.3 Correctness

Very similar to the first approach (section 5.2), we need to prove that this new solution solves all crosstalk issues and also preserves the semantics by conserving the order of gates. The first point has been made clear while explaining the different algorithms previously. We explained how we separate the problematic gates via small barriers inside the same layer. Additionally, if the gate has not been moved because there is not enough space but has crosstalk problems with gates in the next layer, this issue will be solved through the global barriers that we apply at the end.

Now coming to the order of the gates we have made it clear that every time a gate cannot be moved to the layer on the left, we block the qubits in $QubitAvailibility$. This is what prevents gates from jumping in front of other gates that should be executed before. So we check the $QubitAvailibility$ before any move to make sure we do not mess the topological order of the gates. This preserves the execution order of the gates on each qubit.

Algorithm 6 Crosstalk Removal

Input: Layers, Configuration Data
Output: Layers after solving crosstalk issues
$k \leftarrow 0$
while $k < (length(Layers) - 1)$ **do**
    for $i \in [0..length(Layer_k) - 1]$ **do**
        if $i < length(Layer_k) - 1$ **then**
            $j \leftarrow i + 1$
            while $j < length(Layer_k)$ **do**
                crosstalkCheck $\leftarrow crosstalkExists(Layer_k[i], Layer_k[j], ConfigurationData)$
                if *crosstalkCheck* **then**
                    TransferCheck $\leftarrow$ $Layer_{k+1}$ has no gates operating on $Layer_k[j].qubits$
                    $\rightarrow$ the gate $Layer_k[j]$ can be transferred to $Layer_{k+1}$
                    if $k < (length(layers) - 1)$ and $TransferCheck$ **then**
                        Move $Layer_k[j]$ to $Layer_{k+1}$
                        Remove $Layer_k[j]$ from $Layer_k$
                    else
                        Insert an empty layer [] in position $k + 1$ in *Layers*
                        Move $Layer_k[j]$ to $Layer_{k+1}$
                        Remove $Layer_k[j]$ from $Layer_k$
                    end if
                else
                    $j \leftarrow j + 1$
                end if
            end while
        end if
    end for
    if $Layer_k$ is empty **then**
        Remove $Layer_k$ from *Layers*
    end if
    $k \leftarrow k + 1$
end while

Algorithm 7 Generate Barriers

Input: Layers, Circuit
Output: Barriers
existingBarriers = $deleteDoubleBarriers(Layers, circuit)$
$barriers \leftarrow []$
Create a *barrier* gate that goes over all qubits in the circuit
$i \leftarrow 0$
for $Layer \in Layers$ **do**
    if $i = 0$ **then**
        $i \leftarrow i + 1$
        continue
    end if
    if $Layer[0]$ is not a barrier over all the qubits of the circuit **then**
        if Not $(i - 1 \in existingBarriers)$ **then**
            Barriers.add(*barrier*)
        end if
    end if
    $i = i + 1$
end for

Algorithm 8 Crosstalk Solution -Time Dependent-

Input: Circuit, Layers, Configuration Data
Output: New Circuit, New Layers
$MaximizeParallelism(Layers)$ -algorithm 2-
NewLayers $\leftarrow$ *CrosstalkRemoval* -algorithm 6-
$MaximizeParallelismNoCrosstalk(NewLayers)$
Create a Time Dictionary:

$$TimeDict = \{\textbf{Key} : GateName \mapsto \textbf{Value} : Time\}$$

Initialize the width of Layers: *Layer Width*=*Max*(Times of the gates in *Layer*)
Initialize available time per qubit with the function: *GetAvailableTime* -algorithm 9-
New Layers $\leftarrow$ $CrosstalkRemoval - TimeDependant-$ -algorithm 11-
Barriers $\leftarrow$ $generateBarriers(circuit, NewLayers)$ -algorithm 7-
Create a *NewCircuit* using the *NewLayers* and the *Barriers*, creating a separation between the layers.

Algorithm 9 Get Available Time

```
1: TimeAvailable ← []
2: for Layer_i ∈ Layers do
3:     timePerQubit ←
4:     for gate ∈ Layer_i do
5:         for qubit ∈ gate.qubits do
6:             timePerQubit[qubit] ← (widthofLayer_i) − (timeofgate)
7:         end for
8:     end for
9:     TimeAvailable.add(timePerQubit)
10: end for
```

Algorithm 10 Update Time And Gaps Before Adding Crosstalk Barrier

```
1: Input: LayerWidth, LayerIndex, TimeAvailable, FreeGaps, Barrier, Gate
2: Get the minimum time available of all qubits ∈ Barrier
3: if minimum time available > gate.time then
4:     for qubit ∈ Barrier.qubits do
5:         if qubit ∈ gate.qubits then
6:             TimeAvailable[LayerIndex][qubit]=minimum time - gate.time
7:             FreeGaps[LayerIndex][qubit]=0
8:         else
9:             FreeGaps[LayerIndex][qubit]=TimeAvailable[LayerIndex][qubit]-
   minimum time
10:            TimeAvailable[LayerIndex][qubit]=minimum time
11:        end if
12:    end for
13:    return True
14: else
15:    return False
16: end if
```

Algorithm 11 Crosstalk Removal-Time Dependent-

Initialize gaps to 0 for all qubits in all layers
$k \leftarrow 0$
for $Layer_k \in Layers$ **do**
    Initialize qubit availability: $qubitAvailability = \{\textbf{Key} : Qubit | \textbf{Value} : True\}$
    $j = k + 1$
    while $j < length(Layers)$ and there are available qubits in $qubitAvailability$ **do**
        for $gate_i \in Layer_j$ **do**
            if Not ($qubitAvailability$ of all $gate_i.qubits$ is $True$) **then**
                Set $qubitAvailability$ of all $gate_i.qubits$ to $False$
                continue
            end if
            if There is no space left in each gate.qubit in $Layer_k$ **then**
                Set $qubitAvailability$ of $gate_i.qubits$ to $False$
                continue
            end if
            if There is a gap with enough space for the $gate_i$ in $Layer_k$ **then**
                $GetInsertionIndex$, insert the $gate_i$ in the free gap and update the gaps
                continue
            end if
            close the gaps for $gate_i.qubits$ in $Layer_k$
            if There is no crosstalk between $gate_i$ and all gates in $Layer_k$ **then**
                Move $gate_i$ from $Layer_{k+1}$ to $Layer_k$ and update time in $Layer_k$
                continue
            end if
            if No need to add a barrier to prevent crosstalk **then**
                Move $gate_i$ from $Layer_{k+1}$ to $Layer_k$ and update time in $Layer_k$
                continue
            end if
            Update available time and gaps before adding crosstalk barrier-algorithm 10
            if There is no space left to add $gate_i$ **then**
                Set $qubitAvailability$ of $gate_i.qubits$ to $False$
            else
                Add a sub-barrier to $Layer_k$
                Move $gate_i$ from $Layer_{k+1}$ to $Layer_k$
            end if
        end for
        Update the information of $Layer_j$ and remove it, if it is empty
        $j = j + 1$
    end while
end for

6 Conclusion and Future Work

6.1 Conclusion

The compiling process of quantum code is as crucial as writing the code. Therefore, it can be beneficial for programmers and researchers to control the compilation process of their code and have a closer look into the specifications of the hardware they use.

This gives them more control over the changes happening to the code and a better understanding of the output.

To achieve this goal, we implemented an app joining the software and the hardware parts of quantum computing. We are able to represent any quantum hardware with a general data structure without having access to the real quantum computer. We enabled the programmer to test code compatibility with any desired hardware and solve these problems manually or automatically. In this work, we implemented innovative visualization algorithms by scaling the gate size to its execution time. We also created a solution to the crosstalk problem combined with a scheduling technique.

At the end, we provide an independent tool which can be used to detect some compilation problems and solve them. It is a useful tool to fill the gap between quantum programming and quantum platforms.

However, our application has its limitations and many aspects of it can be improved. We have already discussed in previous chapters the limitations of the main algorithms in our work, and in the next section we will show, what improvements can be made in the future.

6.2 Future Work

Our implementation can be further improved to generate a more optimized circuit with minimal error. The first algorithm we can add is logical to physical qubits remapping for minimal crosstalk and connectivity issues[6].

Next, we can run optimization passes on the gates such as gate merging, gate cancellation, gate decomposition, pattern matching and others[21].

Moreover, We can reimplement the connectivity solution using qubit routing[13] to minimize the number of swap gates added.

Additionally, we can improve the time-dependent visualization of the quantum circuit for a better representation of the time aware crosstalk solution. We also can modify the app to support gates operating on more than two gates.

Last but not least, we can improve the supported hardware characteristics. We can support an error model to enable the user to track the error throughout the circuit. Also, we can make the gates more universal by only using the U-form of the quantum operators.

In conclusion, with the rapid development of quantum technology and especially quantum compilers, there are many features and algorithms to be added in order to refine and enhance our tool and make it future-proof.

List of Figures

List of Algorithms

Bibliography

[1] D. Azses, M. Dupont, B. Evert, M. J. Reagor, and E. G. D. Torre. "Navigating the noise-depth tradeoff in adiabatic quantum circuits." In: *Physical Review B* 107.12 (Mar. 2023). DOI: 10.1103/physrevb.107.125127.

[2] S. Balakrishnan. "Various Constructions of Qudit SWAP Gate." In: *Physics Research International* 2014 (Aug. 2014).

[3] R. Computing. *The Quil Compiler*. 2021. URL: https://pyquil-docs.rigetti.com/en/stable/compiler.html.

[4] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. *Open Quantum Assembly Language*. 2017. arXiv: 1707.03429 [quant-ph].

[5] C. Developers. *Cirq*. https://github.com/quantumlib/Cirq. 2019.

[6] Y. Ding, P. Gokhale, S. F. Lin, R. Rines, T. Propson, and F. T. Chong. "Systematic Crosstalk Mitigation for Superconducting Qubits via Frequency-Aware Compilation." In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2020. DOI: 10.1109/micro50266.2020.00028.

[7] J. D. Hidary. *Quantum Computing: an Aplied Approach*. Springer International Publishing AG, 2019-09-20.

[8] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks. "A verified optimizer for Quantum circuits." In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021), pp. 1–29. DOI: 10.1145/3434318.

[9] T. Itoko and T. Imamichi. *Scheduling of Operations in Quantum Compiler*. 2020. arXiv: 2011.04936 [quant-ph].

[10] M. Maronese, L. Moro, L. Rocutto, and E. Prati. *Quantum Compiling*. 2021. arXiv: 2112.00187 [quant-ph].

[11] P. Murali, D. C. Mckay, M. Martonosi, and A. Javadi-Abhari. "Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Mar. 2020. DOI: 10.1145/3373376.3378477.

[12] J. Pointing, O. Padon, Z. Jia, H. Ma, A. Hirth, J. Palsberg, and A. Aiken. *Quanto: Optimizing Quantum Circuits with Automatic Generation of Circuit Identities*. 2021. arXiv: 2111.11387 [quant-ph].

[13] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins. *Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers*. 2020. arXiv: 2007.15957 [quant-ph].

[14] I. Quantum. *Operations glossary*. 2023. URL: https://quantum-computing.ibm.com/composer/docs/iqx/operations_glossary.

[15] I. Research. *Qiskit*. https://github.com/Qiskit. 2023.

[16] I. Research. *Qiskit DAG Circuit*. https://github.com/Qiskit/qiskit-terra/blob/main/qiskit/dagcircuit/dagcircuit.py. 2023.

[17] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan. "t|ket⟩: a retargetable compiler for NISQ devices." In: *Quantum Science and Technology* 6.1 (Nov. 2020), p. 014003. DOI: 10.1088/2058-9565/ab8e92.

[18] *Stack overflow: Shortest path Graph BFS python*. 2022 Accesssed: 2022-05.

[19] Q. D. Team. *Qiskit*. 2023. URL: https://qiskit.org/documentation/index.html.

[20] Q. D. Team. *Qiskit: Crosstalk Adaptive Schedule*. 2023/04/13. URL: https://qiskit.org/documentation/stubs/qiskit.transpiler.passes.CrosstalkAdaptiveSchedule.html.

[21] Q. D. Team. *Transpiler*. 2023. URL: https://qiskit.org/documentation/apidoc/transpiler.html.

[22] K. Wintersperger, F. Dommert, T. Ehmer, A. Hoursanov, J. Klepsch, W. Mauerer, G. Reuber, T. Strohm, M. Yin, and S. Luber. *Neutral Atom Quantum Computing Hardware: Performance and End-User Perspective*. 2023. arXiv: 2304.14360 [quant-ph].

[23] J. R. Wootton. *Benchmarking of quantum processors with random circuits*. 2018. arXiv: 1806.02736 [quant-ph].

[24] L. Xie, J. Zhai, and W. Zheng. "Mitigating Crosstalk in Quantum Computers through Commutativity-Based Instruction Reordering." In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 445–450. DOI: 10.1109/DAC18074.2021.9586145.

[25] P. Zhao, K. Linghu, Z. Li, P. Xu, R. Wang, G. Xue, Y. Jin, and H. Yu. *Quantum crosstalk analysis for simultaneous gate operations on superconducting qubits*. 2022. arXiv: 2110.12570 [quant-ph].