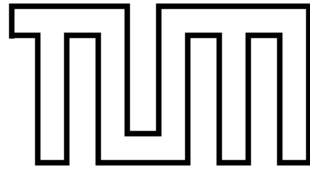


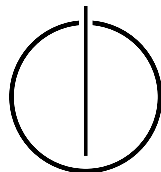

TECHNICAL UNIVERSITY OF MUNICH

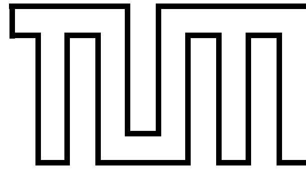
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS


Master’s Thesis in Informatics

# Adaptation of Classical Optimizations to Hybrid Quantum-Classical Programs


Joachim Marin



TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

Master's Thesis in Informatics

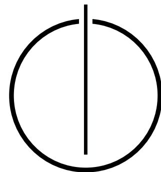
# Adaptation of Classical Optimizations to Hybrid Quantum-Classical Programs

# Anpassung Klassischer Optimierungen an Hybride Quantenklassische Programme



| | |
|---|---|
| Author: | Joachim Marin |
| Supervisor: | Prof. Dr. Helmut Seidl |
| Advisor: | M.Sc. Yannick Stade |
| Date: | 15.4.2023 |





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

Munich, 15.4.2023 Joachim Marin

# Acknowledgements

I would like to thank my advisor Yannick Stade for his continuous guidance and support during my Master's program. His knowledge and enthusiasm about quantum computing and static optimization helped me complete this thesis.

I am also grateful to my supervisor Prof. Dr. Helmut Seidl for giving me the opportunity to write this thesis at his chair.

Finally, I would like to thank Lukas Burgholzer for his work on the Munich Quantum Toolkit and our discussions about quantum circuit mapping.

# Abstract

Optimization of quantum circuits is an important topic, because current quantum computers suffer from big error rates for large quantum circuits. In hybrid quantum-classical computing, the quantum circuit is assembled at run time by a classical program and sent to the quantum computer, which then returns the measured results to the classical program. The final quantum circuit may depend on run time values, so optimization is frequently performed just before the quantum circuit is sent to the quantum computer. In classical computing, static optimization is often preferred, because it is performed at compile time and does not require time during the execution of the program. We therefore present an implementation of a static optimizer for hybrid quantum-classical programs that optimizes quantum circuit generating functions in the program at compile time, so that it produces more efficient quantum circuits at run time. Our implementation is based on the LLVM tool chain and reuses classical optimization techniques to improve the effectiveness of the static optimization.

# Contents

**Acknowledgements** **v**

**Abstract** **vii**

**Contents** **ix**

**1. Introduction** **1**

**I. Theoretical Background** **3**

**2. Quantum Computing** **4**
2.1. Qubits . . . . . . . . . 4
2.2. Quantum Gates . . . . . . . . . 6
2.3. Quantum Circuits . . . . . . . . . 8
2.4. Quantum Computers and Architectures . . . . . . . . . 9
2.5. Hybrid Quantum-Classical Computing . . . . . . . . . 9

**3. Static Optimization of Hybrid Quantum-Classical Programs** **11**
3.1. Joint Optimization of Quantum Instructions . . . . . . . . . 11
3.2. Optimization of the Start of a Quantum Circuit . . . . . . . . . 14
3.3. Optimization of the End of a Quantum Circuit . . . . . . . . . 15
3.4. Optimization of Complete Quantum Circuits . . . . . . . . . 16
3.5. Optimization of Quantum Gate Sequences . . . . . . . . . 16
3.6. Target Specific Optimization . . . . . . . . . 18

**II. Implementation** **20**

**4. Related Work** **23**

**5. Description of Tools** **25**
5.1. LLVM Tool Chain . . . . . . . . . 25
5.2. Munich Quantum Toolkit . . . . . . . . . 26
5.3. Quantum++ . . . . . . . . . 27

**6. QuLib - Quantum Circuit Library** **28**

**7. Linker Interface** **32**

**8. Optimization Pass** **34**
8.1. Quantum Circuit Representations . . . . . 34
8.1.1. LLVM Intermediate Representation . . . . . 35
8.1.2. OpenQASM 2.0 . . . . . 38
8.1.3. QFR . . . . . 39
8.1.4. Custom Quantum Circuit Representation . . . . . 39
8.2. Replacing Quantum Circuits in LLVM . . . . . 42
8.3. Optimization of Quantum Gate Sequences . . . . . 51
8.4. Optimization of Complete Quantum Circuits . . . . . 53
8.5. Run Time Optimization . . . . . 53

**9. CMake Integration** **55**

**10. Implementation Details** **56**

**III. Results** **57**

**11. Optimization Correctness** **59**

**12. Optimization Effectiveness** **60**

**13. Optimization Running Time** **61**

**IV. Conclusion** **63**

**Bibliography** **65**

**V. Appendix** **68**

**List of Figures** **69**

**List of Tables** **70**

**List of Algorithms** **71**

# 1. Introduction

The first computers were programmed by changing the wiring between their components [Ros69]. Since then, programming has advanced a lot and high level programming languages supported by powerful compilers make programming much easier, allowing the creation of more complex and potent programs. In order to ensure the high level language is translated into efficient machine code, a compiler often comes with an advanced optimizer that performs several transformations on the program before it is translated into machine code.

Quantum computing is still in its infancy and quantum algorithms are generally described at a very low level by the quantum circuit consisting of individual quantum gates acting on specific quantum bits [SEL04]. Still, optimization of these quantum circuits is already an important topic in quantum computing, because larger quantum circuits are more likely to lead to errors on current quantum computers [BKM+14]. So optimizing the quantum circuit and reducing the number of gates not only speeds up the computation, but also reduces the probability of incorrect results. In the future, optimization will become even more important, as higher level programming languages for quantum computing are introduced and most of the quantum circuit generation is done by the compiler rather than the programmer.

In hybrid quantum-classical computing, a classical computer drives the overall program flow and uses quantum circuits to perform certain tasks on quantum computers. In the most general case, these quantum circuits can be generated by the classical program at run time, so that arbitrary quantum circuits can be built. However, this creates a big challenge for the static optimization of quantum circuits, because the quantum circuit cannot be determined at compile time. As a result, optimization of quantum circuits is often done during run time, just before the complete quantum circuit is sent to the quantum computer [SHT18, Qis23]. In this work, it will be discussed how optimization can already be done at compile time, so that the hybrid quantum-classical program spends less time on optimizing the quantum circuit at run time, improving its performance.

There are several challenges of static quantum circuit optimization that will be explained in more detail throughout this work. In general, the optimizer needs to find statements that generate quantum circuits in the program and replace them by statements that generate the optimized version of that quantum circuit. Since most quantum circuits are described on a low level, a single statement in the program code is usually equivalent to a single quantum circuit gate and offers little opportunities for optimization, so the optimizer has to find sequences of statements that it can optimize together. This process is not easy, because the optimizer must be able to tell what quantum circuit will be generated by this sequence at run time. These sequences of instructions lack the context of the entire quantum circuit and may even contain arguments whose values are not known at compile time. As a result, the static optimization of quantum circuits must be able to work with very little information about the final quantum circuit at run time.

The main part of this work is an implementation of static quantum circuit optimization based on the LLVM tool chain [LA04] for hybrid quantum-classical programs written in C

or C++. It contains all the required components from compiling the source code to LLVM bitcode files, performing different optimization passes on these LLVM bitcode files and finally linking the final executable. The primary focus of the implementation is a custom LLVM optimization pass for static quantum circuit optimization. This pass translates quantum circuit generating functions into a custom quantum circuit representation, performs optimization on this representation and finally translates it back into function calls, so that the optimized quantum circuit will be generated at run time. Our implementation mostly focuses on the quantum circuit representation and its translation, whereas the optimization performed on this representation is rather basic.

Before the implementation is described, the necessary theoretical background about quantum circuits, hybrid quantum-classical programs and their static optimization will be introduced. Important design decisions, quantum circuit representations, optimization algorithms and implementation details of the implementation will be explained. In the end, the implementation will be evaluated and possible future improvements will be presented.

# Part I.

# Theoretical Background

# 2. Quantum Computing

Quantum computing is a complicated and broad topic, so it cannot be explained in detail in this work. The interested reader can find further information in other works, such as [NC00] and [Hid19]. Nevertheless, the most important concepts and definitions to explain quantum circuit optimization will be introduced in this section.

## 2.1. Qubits

The key difference between classical and quantum computing is the usage of quantum bits, also called qubits, instead of regular bits. A regular bit is either one or zero, so it can only hold two distinct values. Qubits on the other hand can be a combination of the zero and one states. The state of a qubit $|\psi\rangle$ is a linear combination of the zero state $|0\rangle$ and the one state $|1\rangle$ with complex coefficients $\alpha$ and $\beta$ [NC00]:

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

This linear combination is generally referred to as superposition in quantum computing. There are physical limitations when dealing with qubits that do not allow us to directly retrieve the state of a qubit. Instead, a qubit is measured and will return either the zero or one state. According to Born's rule, the probability of a state being measured is equal to the square of the absolute value of its coefficient [Bor26]. As the probabilities need to sum up to one, the following must hold as well:

$$|\alpha|^2 + |\beta|^2 = 1$$

Another way to visualize the state of a qubit is the Bloch sphere shown in Figure 2.1. The state of a qubit is given by a vector from the origin to a point on the sphere that is associated with the state of the qubit. The sphere lives in a three dimensional space with $x$, $y$ and $z$ coordinate axes. The special zero state $|0\rangle$ is on the positive $z$-axis, while the one state $|1\rangle$ is on the opposite side of this axis. The superposition $(|0\rangle + |1\rangle)/\sqrt{2}$ is on the positive $x$-axis, while $(|0\rangle + i\,|1\rangle)/\sqrt{2}$ is on the $y$-axis. Modifications of the qubit state can then be seen as rotations on the Bloch sphere, which allows us to give more intuitive descriptions of individual quantum gates.

If the states $|0\rangle$ and $|1\rangle$ are written as two-dimensional basis vectors, quantum gates can easily be described by matrices. The basis vector belonging to the qubit states are given by [Hid19]:

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

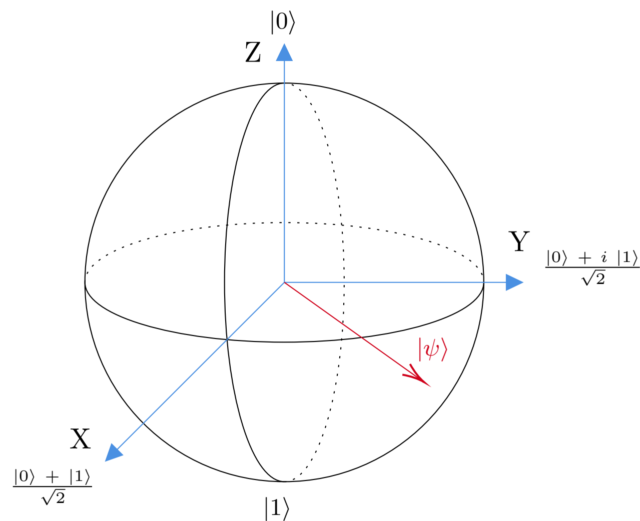



Figure 2.1.: The Bloch sphere represents the state of a qubit as a vector starting at the origin and ending at a specific point on the sphere. This point is associated with the state of the qubit. A change in the state of the qubit can be seen as a rotation of the vector on the Bloch sphere.

Then a quantum gate like the **X** gate can easily be described as a matrix:

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

Applying this gate on a qubit is the same as a matrix multiplication between the gate matrix and the state of the qubit:

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

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

$$\mathbf{X}|\psi\rangle = \mathbf{X}(\alpha|0\rangle + \beta|1\rangle) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} = \beta|0\rangle + \alpha|1\rangle$$

The **X** gate is also known as the **NOT** gate and flips the qubit state, turning $|0\rangle$ into $|1\rangle$ and $|1\rangle$ back into $|0\rangle$.

This idea can easily be extended to several qubits. The state of a two-qubit system can then be described as follows [NC00]:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{2.1}$$

$$= \alpha_{00} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \alpha_{01} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \alpha_{10} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \alpha_{11} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.2}$$

The probability to measure a specific state is again the square of the absolute value of its coefficient, meaning that $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$ must hold. More generally, for $n$ qubits, the combined state is a linear combination of $2^n$ basis vectors and coefficients. This is an interesting property, because it means the combined state of multiple qubits contains more coefficients than just the number of qubits times the number of coefficients per qubit. Every additional qubit doubles the number of coefficients, leading to the immense computational power of quantum computers. However, these coefficients cannot be obtained. When a qubit is measured, its state collapses to the measured result of $|0\rangle$ or $|1\rangle$ [NC00].

## 2.2. Quantum Gates

The X gate was already mentioned, but there are many more quantum gates. Explaining all of them is outside the scope of this thesis, so only the most important ones for our implementation will be introduced. For a detailed overview of the quantum gates the reader is referred to other works such as [Wil11].

The **U** gate is defined in the OpenQASM 2.0 standard [CBSG] and is used as the primary single qubit gate in our implementation. Its matrix is given by:

$$\mathbf{U}(\theta, \phi, \lambda) = \begin{pmatrix} e^{-i(\phi+\lambda)/2} cos(\theta/2) & -e^{-i(\phi-\lambda)/2} sin(\theta/2) \\ e^{i(\phi-\lambda)/2} \sin(\theta/2) & e^{i(\phi+\lambda)/2} \cos(\theta/2) \end{pmatrix} \tag{2.3}$$

This gate is the universal single qubit gate, because all other single qubit gates can be translated into it.

Single qubit gates can be extended to two-qubit gates by adding a control qubit. A controlled gate is only applied to the target qubit, if the control qubit is one. Since these gates operate on two qubits, they require a state vector of size $2^2 = 4$ and hence their matrix must also have that same number of rows and columns. The controlled versions of the already mentioned **X** and **U** gates have the following matrix representations [Qis23]:

$$\mathbf{CX} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.4}$$

$$\mathbf{CU} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta/2) & 0 & -e^{i\lambda} \sin(\theta/2) \\ 0 & 0 & 1 & 0 \\ 0 & e^{i\phi} \sin(\theta/2) & 0 & e^{i(\phi+\lambda)} \cos(\theta/2) \end{pmatrix} \tag{2.5}$$

Note that there are different conventions on the order of the target and control qubit, resulting in slightly different matrices. Here, the convention from OpenQASM 2.0 [CBSG] and Qiskit [Qis23] will be used, as OpenQASM 2.0 code is used by our implementation to output the generated quantum circuit. OpenQASM 2.0 is a well established language for quantum circuits and is supported by many quantum computing tools, allowing our implementation to be used with a wide array of existing tools.

By combining the **U** gate with either the **CX** or the **CU** gate, a universal gate set is already achieved, meaning that any quantum gate can be built with its gates [NC00]. A single **CU** gate can be used to express the **CX** gate, while multiple **CX** and **U** gates are required to express the **CU** gate[CBSG] as shown in Algorithm 1. As the **CX** gate is part of the basis gate set of OpenQASM 2.0 and **CU** is defined as shown in the algorithm, our implementation will also assume a basis gate set of **U** and **CX** gates. The basis gate set will be important in the evaluation of the optimizer, because its effectiveness can be quantified by the reduction in the number of basis gates.

**Algorithm 1:** Expressing **CX** gate and **CU** gate with each other. The gate arguments $c$ and $t$ are the control and target qubits.

```
1 Function CX(c, t):
2     CU(π, 0, π, c, t);

3 Function CU(θ, ϕ, λ, c, t):
4     U(0, 0, (λ + ϕ)/2, c);
5     U(0, 0, (λ − ϕ)/2, t);
6     CX(c, t);
7     U(−θ/2, 0, −(ϕ + λ)/2, t);
8     CX(c, t);
9     U(θ/2, ϕ, 0, t);
```

As the **U** and **CU** gates form a universal gate set, they are sufficient to describe any quantum circuit. However, for the purpose of optimization additional gate definitions are useful, because they introduce certain optimization opportunities. The first group of gates are rotational gates. These gates rotate the qubit about one coordinate axis on the Bloch sphere. They are particularly useful for optimization, because two consecutive rotations with angles $\theta_1$ and $\theta_2$ about the same axis are obviously equivalent to a single rotation with angle $\theta_1 + \theta_2$. These rotational gates are defined as follows [NC00]:

$$\mathbf{R_x}(\theta) = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \tag{2.6}$$

$$\mathbf{R_y}(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \tag{2.7}$$

$$\mathbf{R_z}(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \tag{2.8}$$

Another important gate is the Hadamard or **H** gate. This gate transforms the computational basis states $|0\rangle$ and $|1\rangle$ into the superposition states $(|0\rangle + |1\rangle)/\sqrt{2}$ and $(|0\rangle - |1\rangle)/\sqrt{2}$ and back [Wil11]:

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.9}$$

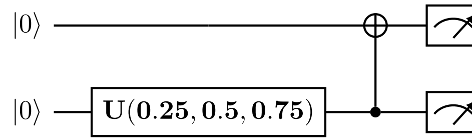



Figure 2.2.: Quantum circuits can be described by intuitive circuit diagrams.

The last gate that will be introduced is the **SWAP** gate, which simply swaps the states of two qubits. Its governing matrix is given by [NC00]:

$$\mathbf{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.10}$$

## 2.3. Quantum Circuits

Using these gates, a quantum circuit can be built. The most intuitive way to describe quantum circuits are circuit diagrams, as shown in Figure 2.2. The quantum circuit diagram is read from left to right, each horizontal line presenting a qubit. The depicted quantum circuit starts with both qubits in state $|0\rangle$. After that, the **U** gate is applied to the second qubit and then a **CX** gate is performed on the first qubit with the second qubit as the control. Most gates are indicated by a rectangle containing the gate name, but the **X** gate usually only uses the $\oplus$ symbol [Hid19]. Control qubits are shown as small dark dots connected to the gate symbol on the target qubit. At the end, both qubits are measured.

OpenQASM 2.0 offers an alternative way to describe quantum circuits that is shown in Listing 2.1. At the start, the OpenQASM version is indicated by `OPENQASM 2.0;`. After that, additional files can be included. For our purposes this will always be the quantum experience standard header `qelib1.inc`, which defines the most common gates apart from the basic **U** and **CX** gates that are always part of OpenQASM 2.0. This is followed by definitions of the qubit and classical bit registers. Quantum gates acting on the qubits can then be added to the quantum circuit. Gate parameters are specified in parentheses, while the affected qubits are always at the end. The control qubit always comes before the target qubit and this convention will also be followed in our implementation. There are generally measurements

Listing 2.1.: OpenQASM 2.0 is a language specification to describe quantum circuits.

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q[2];
4 creg c[2];
5 U(0.25, 0.5, 0.75) q[1];
6 CX q[1], q[0];
7 measure q[0] -> c[0];
8 measure q[1] -> c[1];
```

at the end of the quantum circuit, which are listed with an arrow from a qubit to a classical bit. There also exists a shorthand for measuring all qubits at once by not specifying the bit indexes.

Two quantum circuits will be considered equivalent, if their measured results written to the classical bits have the same probabilities. As explained, a two-qubit state can be written as $|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, and the probability to measure a certain qubit state is equal to the square of the absolute value of its coefficient. So measuring state $|01\rangle$ has probability $|\alpha_{01}|^2$. When the $i$-th qubit is written to the $i$-th classical bit, measuring the qubit state $|01\rangle$ results in the classical state 01. However, if measured qubits are written differently to the classical bits, one might get the classical state 10 from the qubit state $|01\rangle$, because the first qubit is written to the second classical bit and vice versa. As a result, the way qubit measurements are written to classical bits plays an important role when determining the equivalence of quantum circuits. The notion of equivalence of quantum circuits is very important for optimization, because an optimized quantum circuit must always be equivalent to the original one, leading to the same observable outcome.

## 2.4. Quantum Computers and Architectures

Much like classical computers, quantum computers also use different architectures. For the purpose of this paper, two important aspects of the architecture are required. The first one is the basis gate set, which contains the gate types the quantum computer can run. All gates that are not in the basis gate will be called advanced gates and need to be translated into basis gates before the quantum circuit can be run on the quantum computer. The second aspect is the connectivity of the qubits, controlling which qubits can be affected by the same gate. The connectivity can be represented by a directed graph, which is also called coupling map. Making quantum circuits compatible with a coupling map is called quantum circuit mapping. This process is more complicated and is explained in Section 3.6.

Quantum computers do not only differ in which gates they support, but also how well they can perform specific gates. Different quantum computers may have different error rates for the same gate. Gates with higher error rates can sometimes be expressed as an equivalent sequence of different gates to reduce the overall error rate of the circuit.

These different architectures motivate target specific optimization, so that the optimizer makes different choices depending on the target architecture and produces an optimized circuit using the basis gate set of the target architecture. Our implementation mostly focuses on target independent optimization, but it contains a limited version of quantum circuit mapping, which is explained in Section 8.4.

## 2.5. Hybrid Quantum-Classical Computing

Current quantum computers are fundamentally different from their classical counterparts. Not just from a technical or physical perspective, but also a conceptual. Classical computers work on bits that are just zeros or ones and a lot of technology has been built around these bits. They are used to represent many different files, including text, images, videos and many more. Modern computers are also capable of taking input from the user via mouse or keyboard and produce images on the screen or play sounds through speakers.

In contrast, quantum computers are much more limited. Their qubits do not have these many different interpretations as regular bits and quantum computers also have many technical limitations, such as qubits collapsing and losing their state. For these reasons, quantum computers are not used on their own, but in combination with classical computers to perform hybrid quantum-classical computing. There, the classical computer can handle the majority of the program flow. In particular, file access, user input, video and audio output all need to be done by the classical computer. The quantum computer is only employed, once a particular task can be solved by a quantum algorithm. The classical computer then assembles the quantum circuit for this quantum algorithm, sends it to the quantum computer and gets back the measured values as regular bits. In that sense, the quantum computer can be seen as a hardware acceleration unit employed by the classical general-purpose computer.

# 3. Static Optimization of Hybrid Quantum-Classical Programs

Static optimization of hybrid quantum-classical programs is a special form of optimization, because it does change the semantics of the program. The hybrid quantum-classical program produces a quantum circuit at run time with which the programmer can interact. The programmer can retrieve information about the circuit, such as the number of qubits, the number of gates, or even a full representation of the quantum circuit via OpenQASM for instance. If the quantum circuit is optimized at compile time, all this information will differ in the optimized program, resulting in observable differences in the outcome of the program. These differences are not allowed in regular optimization, but for hybrid quantum-classical programs changes in the generated quantum circuit are allowed and desired as long as they result in equivalent measured results on a perfect quantum computer.

Technically, a hybrid quantum-classical program is just a classical program and all its instructions are classical instructions. It is only considered hybrid quantum-classical, because at some point during run time the program communicates with a quantum computer or simulator. The form of communication depends on the target quantum computer, but it usually contains some representation of the quantum circuit that is supposed to be executed. How this quantum circuit is built at run time is also not defined. It could be a quantum circuit language such as OpenQASM, that is included as a string literal or file path in the program or library calls to build a quantum circuit data structure. As a result, it is impossible for an optimizer to perform static optimization of quantum circuits, because it simply does not know what instructions affect the quantum circuit. For that reason, we will introduce specific functions to build quantum circuits and only support static optimization of quantum circuits generated by these functions. Using these functions, all instructions of the hybrid quantum-classical program can be divided into quantum and classical instructions.

## 3.1. Joint Optimization of Quantum Instructions

One important aspect of quantum instructions is that they do not have any effect until the generated quantum circuit is read. Suppose there are five quantum instructions after each other and only the last instruction reads the generated quantum circuit in order to send it to the quantum computer or simulator. The intermediate state of the quantum circuit after the first four instructions does not matter, as long as the state after the last instruction is correct, because only this state is observed. This idea is central to static quantum circuit optimization, because it allows us to produce arbitrary intermediate quantum circuits as long as all observed quantum circuits in the optimized program are equivalent to the original program. The first major challenge is to find sequences of instructions where the quantum circuit is not observed in between. This sequence can then be replaced by a new sequence resulting in an equivalent quantum circuit after the last instruction. These sequences will

**Algorithm 2:** Coin optimization of a simple program with linear control flow, reducing the number of coins from 3 to 2.

```
1 Function Original():
2     GiveCoin(20)
3     GiveCoin(20)
4     GiveCoin(20)
```

```
1 Function Optimized():
2     GiveCoin(10)
3     GiveCoin(50)
```

be called chunks and are the basis for static quantum optimization. In most cases a chunk does not end because the quantum circuit is definitely read by an instruction, but because it cannot be statically verified that the quantum circuit will not be read. This is especially the case just before a branch instruction or at the end of a function, where the next instruction is not known at compile time. Chunks may also contain classical instructions that must be preserved by the optimization.

The instructions contained in the chunk generate the quantum circuit at run time and can be analyzed and optimized at compile time. There are very little restrictions on the instructions in the chunk, as long as the requirement of the quantum circuit never being read is fulfilled. As a result, the instructions may still contain complex control flow, leading to fundamentally different quantum circuits depending on run time values. In order to explain optimization of chunks in a more intuitive and shorter way, the program being optimized will now have a different task. The program will now return coins of a certain total value. We will assume, that coins of values 1, 2, 5, 10, 20, 50 and 100 exist. Two programs are considered equivalent, if they return the same total value. A simple example of such an optimization is shown in Algorithm 2, which reduces the number of coins by one.

**Algorithm 3:** Coin optimization of a program with control flow statements resulting in different outcomes at run time. All outcomes are enumerated and each one is optimized individually. The optimized outcomes are selected by a condition.

```
1  Function Original():
2      GiveCoin(20)
3      GiveCoin(20)
4      GiveCoin(20)
5      if x then
6          GiveCoin(20)
7          GiveCoin(20)
8          GiveCoin(20)
9      if y then
10         GiveCoin(50)
```

```
1  Function Optimized():
2      if x and y then
3          GiveCoin(100)
4          GiveCoin(50)
5          GiveCoin(20)
6      else if x and not y then
7          GiveCoin(100)
8          GiveCoin(20)
9      else if not x and y then
10         GiveCoin(100)
11         GiveCoin(10)
12     else
13         GiveCoin(50)
14         GiveCoin(10)
```

By allowing control flow statements in chunks, the problem becomes more complicated as shown in Algorithm 3. Here, the desired result depends on some run time value. The optimal solution is to enumerate all individual execution paths and optimize the instructions contained within each path. This approach does not work well for several reasons. Every if-condition doubles the number of execution paths, while a loop can even result in infinite execution paths, because the number of iterations may be unknown at compile time. Even if the number of execution paths is limited, the resulting execution paths might be very long. The problem of optimizing a long sequence of quantum instructions as a whole is difficult [SBM06] and time intensive [AMMR13, DM16].

A simpler and more efficient way to optimize quantum circuits is peephole optimization, which only looks at a short section of the quantum gate sequence and optimizes this short section. After the section is optimized, the next section is considered until the end of the sequence is reached. As a result, this method scales linearly with the size of the entire sequence. Obviously, this approach can be used to optimize different execution paths individually. In order to avoid the problems caused by too many or infinite execution paths, chunks can simply be broken up at control flow statements. This may result in significantly more, but smaller chunks. However, as long as the chunks do not get too small, peephole optimization will still be effective. This is because peephole optimization within each chunk will stay the same and only optimization across the new chunk border will be prevented. Therefore, removing control flow statements from chunks and applying peephole optimization results in very fast optimization and still good results for most use cases.

By splitting chunks at control flow statements, three chunks are obtained in the given coin example. These chunks are all optimized individually as shown in Algorithm 4. Obviously, this method does not yield optimal results, but also does not result in exponential optimization time when conditions are encountered, due to the exponential number of execution paths.

**Algorithm 4:** Coin optimization of a program with control flow statements resulting in different outcomes at run time. The original control flow is preserved and the chunk is split up, resulting in three chunks being optimized individually. As the optimization cannot consider all instructions at the same time, no optimal solution is obtained.

```
1  Function Original():
2      GiveCoin(20)
3      GiveCoin(20)
4      GiveCoin(20)
5      if x then
6          GiveCoin(20)
7          GiveCoin(20)
8          GiveCoin(20)
9      if y then
10         GiveCoin(50)
```

```
1  Function Optimized():
2      GiveCoin(50)
3      GiveCoin(10)
4      if x then
5          GiveCoin(50)
6          GiveCoin(10)
7      if y then
8          GiveCoin(50)
```

Whether a chunk is split up at diverging control flow or each execution path is optimized individually, the optimization is performed on a sequence of quantum instructions without

control flow. There exists a lot of literature on the optimization of quantum circuits [SP08, KM13, AMM14, HHT20]. However, in static quantum circuit optimization, the optimized chunk may only be a part of the final circuit at run time and not all gate arguments may be known at compile time. As a result, not all optimization techniques are applicable to static quantum circuit optimization. In the following, several optimization methods for static optimization of quantum circuits will be presented. This is far from an exhaustive list and is tailored towards optimization techniques that are well suited for the optimization of chunks. To enable the best optimization, different cases of the chunk in relation to the final quantum circuit at run time should be distinguished.

## 3.2. Optimization of the Start of a Quantum Circuit

If a chunk is known to not be preceded by another chunk, it can be considered the start of the quantum circuit. Before quantum gates can be added to a quantum circuit, the quantum circuit must be created and stored in a variable that will be passed to the function calls adding quantum gates. This creation usually happens with a function call itself, so the optimizer can easily tell if a chunk is the start of a quantum circuit by checking whether the quantum circuit is created at the start of the chunk.

All qubits start at qubit state $|0\rangle$. If a controlled gate is encountered and the control qubit is known to be in state $|0\rangle$, the controlled gate can be removed. If the optimizer can verify that the qubit is in state $|1\rangle$, the controlled gate can be replaced by its regular version. An example of this optimization process can be seen in Listing 3.1. The first **CX** gate can safely be removed, because the control qubit `q[0]` still has its initial state of $|0\rangle$. The next **CX** gate can be replaced by a regular X gate, because the control qubit `q[0]` is only modified by a single **X** gate before, meaning it is now in state $|1\rangle$. This is a very simple version of state dependent optimization and more advanced optimization methods can be found in [LBZ21] and [JTS+22].

Listing 3.1.: The optimizer can infer the current state of the control qubit `q[0]` at the controlled gates and optimize the controlled gates. If the control qubit is known to be in the state $|0\rangle$, the controlled gate is removed. While the gate is replaced by its non-controlled version if the control qubit is known to be in the state $|1\rangle$.

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q[2];
4 creg c[2];
5 CX q[0], q[1];
6 H q[1];
7 X q[0];
8 CX q[0], q[1];
9 ...
```

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q[2];
4 creg c[2];
5
6 H q[1];
7 X q[0];
8 X q[1];
9 ...
```

## 3.3. Optimization of the End of a Quantum Circuit

If a chunk is known to not be followed by another chunk, it can be considered the end of the quantum circuit. In general, this property is difficult to determine statically, because it is hard to tell whether a given quantum circuit may be used in the future. The optimizer can only confirm this property, if the quantum circuit is removed or made immutable at the end of the chunk. In both cases, the quantum circuit cannot be altered any further at run time, meaning the current chunk will be the last one for this quantum circuit. In many applications, a quantum circuit is generated once, but executed several times at different points in the program. In that case, the circuit is not removed in the same chunk it is defined, so the optimizer will not know whether it is the last chunk. For that reason, a specific function to mark a quantum circuit as finished may be added. This function makes the quantum circuit immutable at run time and allows the optimizer to more easily detect the end of a quantum circuit. Since the quantum circuit will be complete when this function is executed at run time, additional run time optimization may be performed as well.

The end of a quantum circuit allows special optimization techniques. A **SWAP** gate may be eliminated by adjusting all following gates and measurements acting on the swapped qubits. This is shown in Listing 3.2. It is important to note, that this kind of optimization may be detrimental, if the **SWAP** gates are introduced to make the circuit compatible with the target architecture. As a result, this optimization should only be applied, if the optimizer also comes with a good algorithm to map quantum circuits to target architectures. In that case, the quantum circuit can first be stripped from **SWAP** gates and later add **SWAP** gates again if necessary.

Listing 3.2.: A **SWAP** gate at the end of quantum circuit may be eliminated by adjusting all following gates and measurements acting on the swapped qubits.

```
1 ...
2 CX q[0], q[1];
3 SWAP q[0], q[1];
4 H q[1];
5 CX q[1], q[0];
6 X q[0];
7 measure q[0] -> c[0];
8 measure q[1] -> c[1];
```

```
1 ...
2 CX q[0], q[1];
3
4 H q[0];
5 CX q[0], q[1];
6 X q[1];
7 measure q[1] -> c[0];
8 measure q[0] -> c[1];
```

Additionally, the end of the quantum circuit can be searched for gates that do not affect the measured results. This optimization should be done iteratively from the back, because removing a gate can open further optimization possibilities. For instance, a qubit may only affect the measured result because it is used as control qubit by another gate. If the controlled gate is removed, the control qubit no longer affects the measurements and previous gates on the control qubit may become candidates for this optimization. This process is shown in Listing 3.3. Only `q[2]` is measured, so the gates `CX q[0], q[1]` and `X q[1]` can be removed, as both target `q[1]`, which is not read afterwards. By removing `CX q[0], q[1]`, a read of `q[0]` is removed, so that now `H q[0]` can be eliminated as well.

Listing 3.3.: Quantum gates targeting qubits whose state is not measured or read by following gates can be removed. As the removal of a gate can also remove a read of a qubit, this can allow removal of additional gates.

```
1 ...
2 CX q[0], q[2];
3 CX q[1], q[2];
4 X q[2];
5 H q[0];
6 CX q[0], q[1];
7 X q[1];
8 measure q[2] -> c[0];
```

```
1 ...
2 CX q[0], q[2];
3 CX q[1], q[2];
4 X q[2];
5 H q[0];
6
7
8 measure q[2] -> c[0];
```

```
1 ...
2 CX q[0], q[2];
3 CX q[1], q[2];
4 X q[2];
5
6
7
8 measure q[2] -> c[0];
```

## 3.4. Optimization of Complete Quantum Circuits

Combining the two previous concepts, one can obtain complete quantum circuits in a single chunk. However, some arguments of quantum gates may still have unknown values at compile time. On the other hand, if all gate arguments are known at compile time, the quantum circuit is fully resolved at compile time. Such a quantum circuit could even be computed at compile time and the entire run time execution of the quantum circuit replaced by the precomputed result. For quantum circuits, every state has a certain probability to be measured, so the run time could essentially be implemented using a random number generator. Obviously, computing all the probabilities is expensive as it is a form of quantum simulation. As a result, optimization time could be very slow. Additionally, this optimization can only be used in a very specific case, so this optimization is not very effective on average.

In general, there exist a multitude of optimization methods for complete quantum circuits, as many quantum computing frameworks employ run time optimization [SHT18, SPSD20, SDC+21]. This work mostly focuses on the challenges of incomplete quantum circuits caused by the limited knowledge of the quantum circuit at compile time, so optimization of complete quantum circuits will not be covered further.

## 3.5. Optimization of Quantum Gate Sequences

The by far most common occurrence are chunks containing sequences of quantum gates with no information on how the chunk may be combined with other chunks at run time. Without knowledge about possible preceding chunks, the optimizer cannot make any assumptions about the initial state of the qubits. Additionally, there is no information about how the outgoing qubits will be used, so all the outgoing qubit states as well as their indices need to be preserved in the optimized quantum gate sequence. In other words, the optimized quantum gate sequence must be equivalent to the original quantum gate sequence for all qubit states.

As already mentioned, peephole optimization is much faster than optimizing the entire chunk at once. Therefore, this technique is used in our implementation and the individual peephole transformations used by our implementation will be introduced in this section.

### Identity Elimination

The identity gate has no effect and can be safely removed from any quantum circuit. Obviously, adding this gate in the first place seems pointless, but it can also be the result of certain parameter combinations of more advanced gates. The **U** gate becomes equivalent to the identity gate when all three parameters are zero, because then the gate matrix becomes the identity matrix. As a result, if an advanced gate contains a **U** gate and it is possible that all arguments of said **U** gate become zero, the identity gate may be included indirectly in the final circuit. In general, if a specific gate such as the identity gate is used within an optimization, it is important to interpret more generic gates such as the **U** gate as the specific gate if they are equivalent considering the arguments of the more generic gate.

### CX Elimination

Two consecutive **CX** gates with the same control and same target qubit can be removed. Intuitively, if the control qubit is one, the target bit is flipped twice, leading to no change. If the control qubit is zero, no gate is ever applied. As the control qubit may also be in a superposition, it makes sense to look at the matrices to confirm that this optimization always works:

$$\mathbf{CX} \cdot \mathbf{CX} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By obtaining the identity matrix, it is proven that two consecutive **CX** gates do not affect the qubit state and can be removed.

### SWAP Elimination

Swapping two qubits twice in a row returns them to their original position, so these instances of repeated **SWAP** operations on the same qubits can be removed. As the argument order does not matter for the **SWAP** operation, this also works if the two consecutive **SWAP** operations have flipped arguments.

### Rotation Folding

This optimization was already explained in Section 2.2 and affects two consecutive rotations about the same axis. If one of the rotational gates $\mathbf{R_x}(\theta)$, $\mathbf{R_y}(\theta)$ or $\mathbf{R_z}(\theta)$ is encountered twice in a row, the gates can be merged by adding the angles together. Obviously, the gates must also act on the same qubit. This optimization can be extended to the controlled version of the rotational gates, which adds the requirement that the control qubit must also be the same between the two consecutive controlled gates.

### More Optimization Possibilities

The presented optimization methods were chosen for this implementation, because they are easy to implement and showcase how optimization can be performed on chunks. There

are many more possible optimization methods. Peephole optimization can be improved by adding optimization methods that look at larger sections than just two consecutive gates. For smaller sequences, it might make sense to avoid peephole optimization and directly optimize the sequence as a whole, leading to better results at still fast optimization times due to the small problem size.

## 3.6. Target Specific Optimization

As explained in Section 2.4, different target architectures support different gate sets and the coupling map limits which qubits can be affected by the same gate. Additionally, different quantum computers will have specific error rates for every gate. All of this information can be incorporated into target specific optimization. The optimizer must put out a quantum circuit only using gates that are supported by the target architecture and the qubit arguments of the gates must be connected in the coupling map of the architecture.

Translating advanced gates into basis gates is quite straight forward, because equivalencies between the common gates exist and are well known. For instance, OpenQASM 2.0 contains definitions of many common gates based on its basis gates [CBSG]. This can be done for every advanced gate individually, so it can be applied to chunks without having to worry about the relation to other chunks.

Ensuring compatibility with the coupling map is much more difficult, because it cannot simply be done locally. This process is also called quantum circuit mapping and works by distinguishing between logical and physical qubits [WB23]. The logical qubits are the qubits specified in the original quantum circuit, where gates may act on arbitrary qubits. The mapping algorithm then needs to map every logical qubit to a physical one. This mapping may change throughout the circuit, if no single mapping fulfilling all requirements can be found. Additional gates, such as **SWAP** gates are added to the circuit to update the mapping throughout the circuit. Suppose there is a simple star-shaped coupling map containing four qubits with physical qubit $p_0$ in the center and the three other qubits only connected to qubit $p_0$. Listing 3.4 shows how a simple quantum circuit can be mapped to this architecture. Clearly, the quantum circuit contains two similar parts in which a single qubit interacts with three different ones. As a result, this single logical qubit is mapped to the center of the star, resulting in the mapping $q_2 \rightarrow p_0$ for the first part. The other mappings can be chosen freely, because all other physical qubits are equivalent in the star architecture, resulting in $\{q_0 \rightarrow p_2, q_1 \rightarrow p_1, q_2 \rightarrow p_0, q_3 \rightarrow p_3\}$ as a possible initial mapping. In the second part, the logical qubit $q_3$ interacts with the other qubits, so now this qubit must be mapped to the physical qubit $p_0$ in the center of the star architecture. The current qubit in the center, must be mapped to one on the outside, though it again does not matter which one. As a result, the physical qubit $p_0$ in the center is swapped with the physical qubit $p_3$, resulting in the new mapping $\{q_0 \rightarrow p_2, q_1 \rightarrow p_1, q_2 \rightarrow p_3, q_3 \rightarrow p_0\}$. The new mapping must also be applied to the measurements, so that the classical bits contain the results according to the logical qubits as defined by the original quantum circuit.

Finding the optimal mappings is a complicated task and there exist exact [WBZ19] and heuristic [ZPW19] algorithms for it. One important challenge is to add as few additional operations as possible. This makes the mapping of chunks difficult, because different chunks running after each other at run time might have completely different mappings, requiring

Listing 3.4.: Mapping a quantum circuit to a specific quantum circuit requires a mapping from logical qubits to physical ones. This mapping is adjusted throughout the circuit by the addition of **SWAP** gates.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
creg c[4];
CX q[2], q[0];
CX q[2], q[1];
CX q[2], q[3];
CX q[3], q[0];
CX q[3], q[1];
CX q[3], q[2];
measure q -> c;
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
creg c[4];
// 0->2, 1->1, 2->0, 3->3
CX q[0], q[2];
CX q[0], q[1];
CX q[0], q[3];
SWAP q[0], q[3];
// 0->2, 1->1, 2->3, 3->0
CX q[0], q[2];
CX q[0], q[1];
CX q[0], q[3];
measure q[0] -> c[3];
measure q[1] -> c[1];
measure q[2] -> c[2];
measure q[3] -> c[0];
```

many **SWAP** operations to be inserted at run time. This problem cannot fully be solved, but there are several strategies to combat it. Finding larger chunks reduces the number of chunk borders, where different mappings meet each other and **SWAP** operations need to be added at run time. The mapping algorithm could also consider this uncertainty of chunk relations and try to minimize different mappings at chunk borders. The mapping algorithm could find a global mapping that all chunk borders must adhere to and simply add additional **SWAP** operations at the start and end of chunks to conform to this global mapping. The algorithm could find such a global mapping that requires a minimum number of **SWAP** operations at the start and end of chunks on average. This strategy produces chunks with the same mappings, so that no **SWAP** operations need to be added at run time, forming a fully static quantum circuit mapping. However, this approach cannot guarantee optimal mapped quantum circuits and may end up adding many additional **SWAP** operations.

Another approach is a form of hybrid static-dynamic mapping. The mapping algorithm would analyze possible mappings within the chunk, but not finalize them. Instead, the analysis results are written into the program code, so that they can be used at run time. This could speed up the mapping process, because analysis within the chunk does not have to be performed at run time, while the run time still has access to the full quantum circuit allowing an optimal circuit mapping. However, this requires the mapping algorithm to be able to perform meaningful analysis on small portions of the quantum circuit that can be reused when making mapping choices for the entire quantum circuit.

Regardless of the strategy used, adding **SWAP** operations to translate between two mappings is not trivial, because the **SWAP** operations itself must also adhere to the coupling map. As a result, it is far easier to reuse existing mapping algorithms at run time when the entire quantum circuit is known.

# Part II.

# Implementation

In this part, we present an implementation of static optimization for hybrid quantum-classical programs. This implementation adds an extra compilation step that detects quantum circuit chunks in the user program, optimizes the chunks and then writes the optimized chunk back to the user program, so that the optimized quantum circuit will be generated at run time. As there already exist several optimization algorithms for entire quantum circuits, this implementation focuses on performing optimization statically and uses only basic optimization algorithms.

The communication between the classical program and the quantum computer or simulator can be realized with quantum circuit languages such as OpenQASM 2.0 [CBSG]. As a result, hybrid quantum-classical programs can be written in arbitrary classical languages by assembling an OpenQASM 2.0 string at run time and sending it to the quantum computer. However, the optimizer needs to be able to detect quantum circuit generating instructions in the program, so that they can be optimized. Therefore, the quantum circuit must be generated by a well defined instruction set and the optimizer must be designed to understand these instructions. There are several ways to achieve this well defined instruction set. Hybrid quantum-classical languages that already include quantum instructions could be designed. Quantum instructions can also be added to existing classical languages via language extensions or libraries. For this purpose, we define a new C library called QuLib. This allows the programmer to write the hybrid quantum-classical program in a well known language with extensive library and tooling support. This library is described in more detail in Chapter 6.

By using the C language for the hybrid quantum-classical program, existing compiler technology can be reused. The LLVM project[LA04] is a modern compiler framework written in C++ and comes with its own intermediate representation that simplifies writing custom optimization passes. The LLVM Clang compiler serves as an entry point to convert the hybrid quantum-classical program into LLVM bitcode files, which can then be used as input for LLVM optimization passes. As LLVM Clang also supports C++, the hybrid quantum-classical program can be written in both C and C++. These languages are some of the most commonly used languages, so supporting them makes the optimizer accessible for many projects.

Once the LLVM bitcode files are obtained, several optimization passes are applied to them. These optimization passes are triggered by the Linker Interface, which is an executable replacing the original linker. As a result, the quantum circuit optimization happens between compilation and linking of the hybrid quantum-classical program. Once the optimization passes are finished, the Linker Interface sends the optimized LLVM bitcode files together with other linker arguments to the real linker that produces the final executable. The Linker Interface is explained in Chapter 7.

The optimization passes applied to the LLVM bitcode files are a combination of existing LLVM optimization passes and a custom optimization pass for static quantum circuit optimization. This custom optimization pass is presented in Chapter 8. This entire compilation process is visualized in Figure 3.1.

Outside of these three main components, there are some additional adjustments required compared to the regular compilation process. In particular, special compile and link flags must be set to use LLVM bitcode files during the compilation process.

This project provides new CMake functions to simplify the process of triggering this new compilation process. This contains the inclusion of the QuLib headers, linking the QuLib

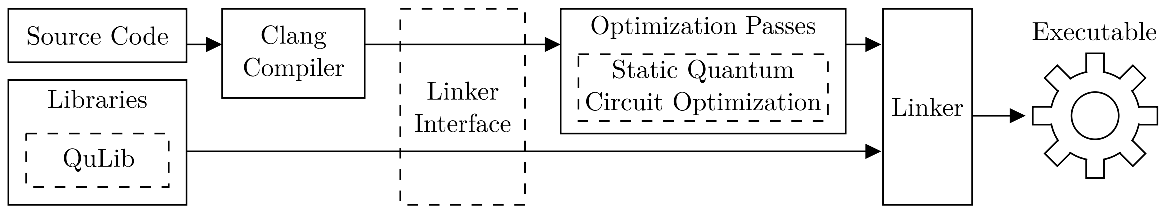



Figure 3.1.: Modified compilation process with new components marked as dashed boxes. The source code is compiled to LLVM bitcode files with the Clang compiler. The Linker Interface triggers optimization passes on the LLVM bitcode files before passing them to the linker, while libraries are directly passed to the linker. The linker then produces the final executable.

library, enforcing the correct flags in the LLVM Clang compiler and using the Linker Interface instead of the regular linker. The CMake integration is explained briefly in Chapter 9.

# 4. Related Work

Optimization of quantum circuits in itself is an important topic, because more instructions are more likely to introduce errors in the computation. As a result, there exists a lot of work related to the optimization of quantum circuits, such as [SP08, KM13, AMM14, HHT20]. There also exist several compilers for quantum programs that often come with optimizations as well.

### ScaffCC

ScaffCC [JPK+14] introduces the Scaffold programming language, which is an extension of C. This project also uses LLVM, but works differently, because it compiles an extension of the C language. ScaffCC does not support arbitrary hybrid quantum-classical programs and requires the classical control flow within quantum modules to be statically resolvable.

### ProjectQ

ProjectQ [SHT18] is a compiler framework for quantum computing. It defines a high level language as an embedded domain specific language in Python. The compiler then optimizes the quantum program over different intermediate representations and compiles it to a given back-end. This back-end may be a quantum computer with a specific architecture, a quantum simulator or even just a circuit drawer. As an embedded language in Python, there is no static compilation and the quantum circuit must be compiled and optimized at run time.

### t|ket⟩

t|ket⟩ [SDC+21] is a compiler that enables targeting different quantum architectures. Naturally, it supports multiple quantum computer and simulator back-ends with different architectures. It can also work with multiple different front-ends, including OpenQASM, Qiskit and ProjectQ. t|ket⟩ is especially designed to perform target specific optimization, minimizing the error caused by the target quantum computer. However, static optimization is only indirectly possible with t|ket⟩. The project itself can optimize quantum circuits given in any of its front-ends, so one would need to obtain such a representation at compile time.

### QIRO

Perhaps the most closely related project is QIRO or Quantum Intermediate Representation for Optimization [IHKH22]. This is an intermediate representation designed for quantum-classical co-optimization, meaning that both quantum and classical instructions are optimized within the same intermediate representation. This is different from our approach, because we only extract quantum instructions, transform them into a quantum circuit representation and optimize this representation independently from the classical instructions. QIRO is

based on the Multi-Level Intermediate Representation (MLIR) [LAB+21], which allows many existing optimization passes from MLIR to be applied to QIRO. This is similar to how our implementation reuses LLVM optimization passes. QIRO is intended to be used with a high level quantum programming language as the front-end and a quantum computer or simulator at the back-end. However, the front-end and back-end are not part of QIRO itself and require future contributions to create a full compilation stack for static quantum circuit optimization.

# 5. Description of Tools

This chapter introduces various tools and libraries related to static optimization and quantum computing that make this implementation possible:

## 5.1. LLVM Tool Chain

The LLVM toolchain[1] [LA04] is used to perform static optimization during compilation. It contains several sub projects relevant for our implementation that are described in this section.

### LLVM Intermediate Representation

The LLVM intermediate representation (LLVM IR) is the abstraction layer on which all optimization passes of LLVM operate. It is employed to detect quantum circuit generating function calls in the program code. The found functions calls are then processed to build chunks. The quantum circuit represented by the chunk is then optimized. Finally, the optimized circuit is written back to the LLVM intermediate representation, replacing the initially detected quantum circuit generating functions. The intermediate representation gives a hierarchical view of the program code, exposes the control flow of the program and allows the optimizer to easily distinguish between many different instruction types, such as function calls, variable assignments or conditions. This allows us to understand how the program will behave at run time. This information is then used to properly optimize the quantum circuit generating functions.

### LLVM Clang

The Clang[2] compiler produces LLVM bitcode files from the C or C++ source code of the hybrid quantum-classical program. Special compiler flags are required to enforce the output of these LLVM bitcode files instead of regular object files. These bitcode files are essentially the LLVM intermediate representation as binary files. LLVM bitcode files are required as input for the other LLVM tools, so Clang serves as an entry point for the entire LLVM tool chain.

### llvm-link - LLVM Bitcode Linker

The `llvm-link`[3] binary combines several LLVM bitcode files into one, simplifying working on the LLVM bitcode. Dealing with a single file is easier for two reasons. Firstly, the

[1] `https://github.com/llvm/llvm-project`
[2] `https://clang.llvm.org`
[3] `https://llvm.org/docs/CommandGuide/llvm-link.html`

optimization process only has to be started once. Secondly and more importantly, the optimization pass has access to the entire LLVM bitcode at once, allowing it to use more information to make optimization choices.

#### opt - LLVM Optimizer

The `opt`[4] binary runs an optimization pass by reading a LLVM bitcode file, turning it into LLVM intermediate representation in memory and then performing the actual optimization pass on the LLVM intermediate representation. After the pass, the LLVM intermediate representation is turned back into a LLVM bitcode file. Many optimization passes are implemented by `opt` already, but it is also possible to load and run new optimization passes with this binary.

#### llvm-dis - LLVM Disassembler

The `llvm-dis`[5] binary converts a LLVM bitcode file to the human-readable LLVM assembly language. This is mostly useful during development. For our purposes this binary is also used to distinguish between regular object files and LLVM bitcode files, as our optimizer only operates on the LLVM bitcode files.

#### LLVM LLD Linker

The final optimized LLVM bitcode file, regular object files, and library files are linked to the final executable using the LLD[6] linker. This linker is required, as it supports LLVM bitcode files unlike the regular ld linker.

## 5.2. Munich Quantum Toolkit

The Munich Quantum Toolkit is a quantum computing framework that comes with many useful sub projects, some of which are used in our implementation and will be explained in this section.

#### QFR - Quantum Functionality Representation

The Quantum Functionality Representation (QFR)[7] is a quantum circuit representation written in C++ and is described in [BRSW21]. It is used by the other tools of the Munich Quantum Toolkit. In order to use these other tools, the quantum circuit must first be defined as a QFR circuit. This representation is explained in more detail in Subsection 8.1.3.

#### QMAP - Quantum Circuit Compilation

QMAP[8] is a quantum circuit mapping tool that makes quantum circuits compatible with the coupling map of a given architecture as described in Section 3.6. It contains both

[4] `https://llvm.org/docs/CommandGuide/opt.html`
[5] `https://llvm.org/docs/CommandGuide/llvm-dis.html`
[6] `https://lld.llvm.org/`
[7] `https://github.com/cda-tum/qfr`
[8] `https://github.com/cda-tum/qmap`

exact [WBZ19] and heuristic [ZPW19] mapping algorithms and is summarized in [WB23]. However, the exact algorithm is much slower and only works for up to eight qubits according to the documentation of QMAP, so this implementation makes use of the heuristic mapping algorithm. The tool is easy to use and only requires as input the QFR circuit and the coupling map and returns the optimized OpenQASM 2.0 code. It is important to note, that there are some limitations on which QFR circuits can be used with QMAP. At the time of writing, the **CX** gate is the only two-qubit gate that is allowed, though there is active development for bringing support for arbitrary two-qubit gates. This is not much of a problem for our implementation though, because our basis gates **U** and **CX** are supported.

### QCEC - Quantum Circuit Equivalence Checking

The Quantum Circuit Equivalence Checking (QCEC)[9] tool is not directly used in our implementation, but is invaluable during development and for evaluating the implementation. This tool takes two QFR circuits as input and checks them for equivalence as described in [BW21]. As optimized quantum circuits need to be equivalent to the original circuits, this tool allows us to easily verify this property.

## 5.3. Quantum++

Quantum++[10] is a modern quantum simulator written in C++ and is explained in [Ghe18]. As a header only library, it is easy to add to C++ programs and does not add additional requirements for the user program. It is used in implementation to run the generated quantum circuits.

[9] `https://github.com/cda-tum/qcec`
[10] `https://github.com/softwareQinc/qpp`

# 6. QuLib - Quantum Circuit Library

The first component of the project is a custom circuit library called QuLib that allows the user to generate quantum circuits at run time. Our optimizer requires a specific set of quantum circuit generating functions, so that it can detect them at compile time and distinguish quantum circuit generating instructions from regular instructions. It would also be possible to use an existing quantum circuit library for that task, though defining a new one gives us full control over it and enables us to specifically design it with static optimization in mind.

In this section, the different functions from this quantum circuit library will be explained, focusing on why certain design decisions have been made to either simplify or improve static optimization. While the overall project is mostly written in C++, the quantum circuit library itself is written in C with additional compatibility for C++ compilation, so that both C and C++ programs can make use of QuLib. As a result, the internal quantum circuit class cannot directly be part of the signature of QuLib functions and instead a wrapper struct is used as the main quantum circuit data type, as shown in Listing 6.1. Additional data types are a 32-bit unsigned integer for the indices of qubits and classical bits, as well as 64-bit double precision floating points for most gate arguments.

Listing 6.1.: The C++ class type is included as a void pointer in a C compatible wrapper struct. This allows the header file to be used by C programs while the implementation in the C++ source file can access the class type by casting the void pointer to the class type.

```
1  typedef struct {
2    void* cppCircuit;
3  } cCircuit;
```

The function signatures shown here are simplified versions of the actual implementation that hide certain implementation details that are not important for the optimization process. Every QuLib function will also be classified as a read, write or special operation. This classification decides how the function affects the chunk detection process. A read operation reads some information about the quantum circuit object, so the optimized quantum circuit must always be equivalent to the original quantum circuit before every read operation in the program code. A write operation modifies the quantum circuit object. The special classification is used, if a function affects the chunk detection in a unique way. This chunk detection process is described in more detail in Section 8.2.

`cCircuit* qulib_alloc(uint32 numberOfQubits)`

This function creates a new quantum circuit with `numberOfQubits` qubits and returns a pointer to it. The type `uint32` is a 32-bit unsigned integer, as the number of qubits obviously cannot be negative. Additionally, 32 bits is more than enough to represent the number of qubits, considering the largest quantum computer in 2022 only has 433 qubits [Gam22], and the largest 32-bit number is over four billion. By not choosing 64-bit integers, there are still qubit indices that are not available to the user of QuLib, but can help us during the optimization process. For example, an optimization algorithm might require all qubit indices to be known, but during the static compilation process some qubit indices are variables rather than known values. Then every unique variable can be mapped to a qubit index that is equal or greater than $2^{32}$, so that they cannot collide with regular qubit indices. While this is currently not used by our optimizer, the 32-bit limitation comes at no cost due to the already mentioned comparatively small number of qubits in current quantum computers, so keeping some qubit indices reserved for the optimizer is a good way to simplify further optimizations in the future.

This function conveys two important pieces of information for the optimization process. If the static optimizer encounters this function call, the number of qubits of the current quantum circuit is known and more importantly it is also clear, that this is the start of a quantum circuit and it contains no gates yet. This allows the optimizer to see the start of a quantum circuit definition. For these reasons, this function is always at the start of a chunk and is considered a special operation.

`void qulib_free(cCircuit* circuit)`

Freeing the struct itself is not enough, because it contains a pointer to the quantum circuit object that needs to be freed as well. Therefore, `qulib_free` is supposed to be used in order to free both the quantum circuit object and the wrapper struct at the same time. This function is a special operation, because it can never be part of a chunk.

`char* qulib_compile(cCircuit* circuit, int optimize, const char* architecture)`

This function compiles the quantum circuit and returns the compiled circuit as OpenQASM 2.0 code. It also performs run time optimization if `optimize` is true. By providing an architecture string, the quantum circuit will be mapped to this architecture at run time using QMAP. The string is passed directly to QMAP, so it must adhere to QMAP's format for architecture strings.

As some of these steps can majorly transform the quantum circuit, the quantum circuit is marked immutable and can no longer be modified by other functions. This is necessary, because the mapping can result in swapped qubit indices, so that newly added gates would also need to be adjusted. Furthermore, this allows the optimizer to more easily detect the end of quantum circuits, which enables additional optimization as described in Section 3.3. As a result, this function is also a special operation.

Our implementation also allows static target specific optimization, even if it is very basic at the moment. This requires the target architecture to be provided at compile time. Since this function could be used to target different architectures at run time, the architecture

parameter is ignored while target specific static optimization is used and replaced by the architecture provided at compile time.

**`char* qulib_get_openqasm(cCircuit* circuit)`**

It is also possible to obtain the OpenQASM 2.0 code of the quantum circuit without compiling it. This is mostly useful for debugging, in order to get the OpenQASM 2.0 code before the circuit is optimized at run time. This function is a read operation, because it only reads the quantum circuit and does not alter it. This is important, because static optimization must ensure the optimized circuit is equivalent to the original circuit at any point it may be read, as explained in Section 3.1.

**`void qulib_simulate(const char* openQASM, uint32 numberOfShots, void* outResults)`**

This function is different from other QuLib functions, because it is not related to static optimization at all. It is simply a utility function to make it easier to simulate quantum circuits without requiring additional dependencies. It does not operate on our quantum circuit type, but on a string instead. This means it is neither a read nor a write operation and can be ignored by the static optimization.

**`void qulib_measure_single(cCircuit* circuit, uint32 qubit, uint32 clbit)`**

This function inserts a measurement operation in the quantum circuit that measures the qubit with index `qubit` and writes the result into the classical bit with index `clbit`. This function modifies the quantum circuit and is therefore a write operation.

**`void qulib_measure_all(cCircuit* circuit)`**

In many applications all qubits are measured, so this function allows the user to formulate these quantum circuits more concisely. This function inserts a measurement for every qubit index and writes the result into the classical bit with the same index. Like the previous function, this is also a write operation.

**`void qulib_u(cCircuit* circuit, double theta, double phi, double lambda, uint32 targetQubit)`**

**`void qulib_cx(cCircuit* circuit, uint32 controlQubit, uint32 targetQubit)`**

**`void qulib_cu(cCircuit* circuit, double theta, double phi, double lambda, uint32 controlQubit, uint32 targetQubit)`**

These functions insert the respective gates. They are currently the only gates available in QuLib in order to keep it small and make the detection of QuLib calls easier. However, additional gates can easily be added to QuLib and the internal quantum circuit data structure already uses additional gates to efficiently perform quantum circuit optimization. All gate functions modify the circuit and are write operations.

`void qulib_message(cCircuit* circuit, const char* message)`

This function exists solely for the optimizer and it can be inserted to move arbitrary information from the compile time to the run time. Currently, this function is used to notify the run time how much a particular quantum circuit has already been optimized at compile time and whether run time optimization is still necessary. Additionally, if the program is compiled for a specific target architecture, this function is used to pass the architecture information to the run time. As this function is inserted by the optimizer itself and is not supposed to be part of the original user program, this function is not relevant for finding chunks. Still, it could be considered a write operation, because it adds information to the quantum circuit object.

# 7. Linker Interface

The custom optimization passes from `opt` operate on LLVM bitcode files. The regular C and C++ compilation processes do not use LLVM bitcode files, so some adjustments are required. The regular compilation process first compiles source files to object files and then links the object files and required library files to an executable. This two step compilation process is often automated by build tools, because invoking the compiler and linker manually requires passing many arguments, especially for large projects consisting of multiple files and many dependencies. The object files generated by the compiler are then merely input files for the linker and the user does not have to interact with them directly.

One can use the `Clang` compiler with special flags and the `LLD` linker to use LLVM bitcode files instead of the regular object files. However, when using build tools to automate this compilation process, the object files or LLVM bitcode files are merely temporary files. For our optimizer on the other hand, the LLVM bitcode files must be optimized before they are linked.

A possible solution is to use another executable, which we will call Linker Interface, in place of the `LLD` linker. The `Clang` compiler will still produce LLVM bitcode files, but instead of invoking the `LLD` linker, it will invoke our Linker Interface with the exact same arguments. This allows us to store the arguments, perform optimization on the LLVM bitcode files and then invoke the `LLD` linker with the stored arguments and the now optimized LLVM bitcode files. The entire process from the original arguments of the Linker Interface to the invocation of the `LLD` linker is visualized in Figure 7.1.

The linker is invoked by the compiler using many different arguments. The most important arguments are the object and library files that are linked together. In the case of the `LLD` linker, there might also be LLVM bitcode files. The Linker Interface needs to find all LLVM bitcode file arguments, so that it can perform quantum circuit optimization on them. This process is fairly straight forward, as a linker argument needs to meet three requirements for it to be considered a valid LLVM bitcode file. The argument must be a valid path to an existing file, because only those can be processed. Additionally, the file extension must be `.o` . Even though LLVM bitcode files generally use the extension `.bc`, LLVM Clang always produces `.o` files whether they are regular object files or LLVM bitcode files. Finally, `llvm-dis` is used to verify that a given file is indeed an LLVM bitcode file. This binary of the LLVM tool chain converts an LLVM bitcode file to the human-readable LLVM assembly language and will return an error message if the input file is not a valid LLVM bitcode file.

Once all LLVM bitcode files in the arguments have been found, the remaining arguments are stored, because they will later be passed unaltered to the real linker. This is required, so that all functionality of the original linker is preserved. The LLVM bitcode files are then processed. At first, they are combined into a single LLVM bitcode file via `llvm-link`. Always having a single LLVM bitcode file simplifies any further steps, because they only have to be run on a single file. Additionally, this file contains all the available context, so that optimization can be as effective as possible.

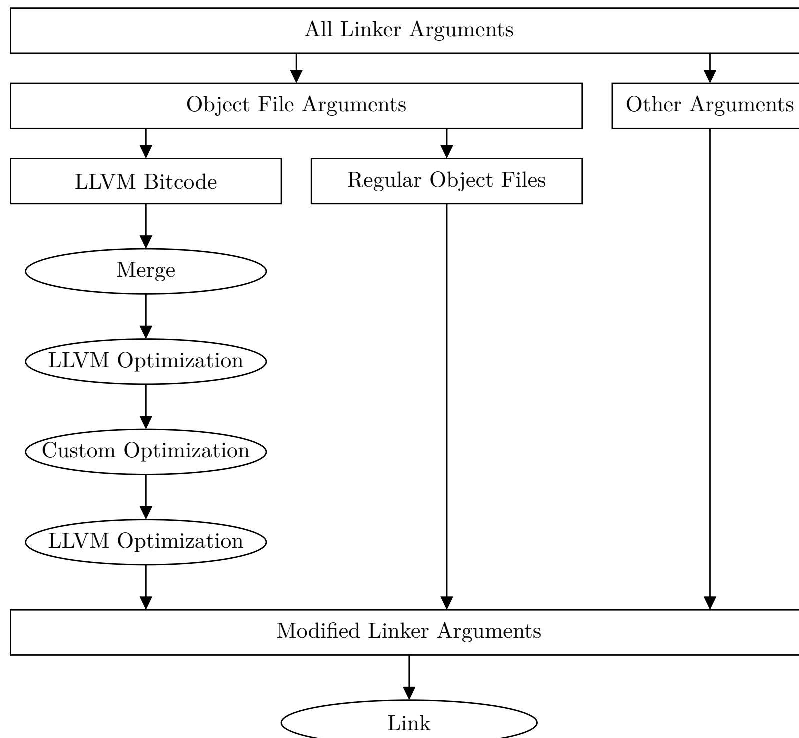



Figure 7.1.: The Linker Interface is initially used instead of the real linker, receiving the same arguments. The arguments are filtered for LLVM bitcode files, which are merged into a single LLVM bitcode file. Different optimization steps are then performed on the merged LLVM bitcode file. Finally, the resulting LLVM bitcode file is passed with all other original arguments to the real linker to produce the final executable.

In the next step, the LLVM bitcode file receives several optimization passes. At first the file is optimized with the regular optimization group O3 of LLVM, which contains many common optimization methods. The optimized code is easier to work with for our custom optimization pass, because unnecessary parts are removed, function calls are inlined and loops unrolled. This removes many control flow statements and produces longer sequences of basic function calls, increasing the size of chunks.

After the regular LLVM optimization passes have simplified the program as much as possible, our custom optimization pass runs as described in Chapter 8 and produces a new LLVM bitcode file once again. The regular LLVM optimization passes are run another time, since our own optimization pass can transform the program quite a bit and open up additional optimization opportunities.

Finally, the `LLD` linker is invoked with the stored arguments that are not LLVM bitcode files together with the path of the newly produced LLVM bitcode file. The difference between the original arguments and the new ones is that all LLVM bitcode files have been replaced by the single bitcode file that went through the custom optimization pass, essentially running custom optimization passes just before linking the final executable.

# 8. Optimization Pass

The custom optimization pass written for the `opt` tool is the main part of the implementation, as it takes the existing program and modifies it to produce more optimized quantum circuits at run time.

## 8.1. Quantum Circuit Representations

Due to the different tools used by this project, multiple different quantum circuit representations are required, because most tools only support a limited set of quantum circuit representations. Every quantum circuit representation comes with certain advantages and disadvantages. They are important, when deciding which quantum circuit representation to use for static optimization.

This section introduces the different quantum circuit representations used by this implementation and discusses their viability for static quantum circuit optimization. Table 8.1 gives a quick overview in that regard, showing whether they fulfill specific important requirements for static quantum circuit optimization. The first requirement is that the quantum circuit representation must be available at run time, so that QuLib can build a circuit in this representation. While this is technically not required for static circuit optimization, it does make the development easier, because a lot of functionality will automatically be available both at run time and compile time. In particular, any optimization passes that operate on the quantum circuit representation can then also be applied as run time optimization.

Obviously, the quantum circuit representation must be present at compile time, otherwise no static optimization is possible. This requirement however is not hard to fulfill, because during compile time arbitrary code can be executed. This is not necessarily the case for the run time, because adding additional libraries to the run time increases the size and dependencies of the user program. Keeping the executable size small and dependencies limited makes both the compilation and installation process faster and easier.

Another very important requirement is that the circuit representation must support symbolic arguments. During the static optimization, not all values will be known and the optimizer must be able to deal with variables with unknown values. These variables will be referred to as symbolic variables, to differentiate them from regular variables. If the user program contains a variable, it may become a constant value, if the optimizer can infer its value. In that case, this value of numerical type is simply used as argument for the quantum circuit representation. This is obviously supported by all quantum circuit representations. On the other hand, if the optimizer cannot infer the value of a variable, it must be converted to a symbolic variable that is then used as input argument for the quantum circuit representation. Ideally, the quantum circuit representation should give as much information on the relations between symbolic variables. Most importantly, if the same symbolic variable appears twice, it must know that they have the same value. Additionally, if a certain symbolic variable is computed as a term of other symbolic variables, this relation

| | Run Time | Compile Time | Symbolic | Ease of Use |
|---|---|---|---|---|
| LLVM IR | X | ✓ | ✓ | X |
| OpenQASM 2.0 | ✓ | ✓ | X | X |
| QFR | ✓ | ✓ | X | ✓ |
| Custom | ✓ | ✓ | ✓ | ✓ |

Table 8.1.: Overview of the capabilities of the individual quantum circuit representations in the context of static quantum circuit optimization.

can also be helpful. How these relations can be used for optimization will be shown in Section 8.3.

Finally, the quantum circuit representation should be easy to use in the context of static optimization. While this is obviously a subjective criterion, the circuit representation should have definitions for common gates and contain type-safe functions to add these gates and retrieve their arguments. Additionally, the surrounding gates must be easily accessible, because peephole optimization operates on short sequences of gates.

### 8.1.1. LLVM Intermediate Representation

The LLVM Intermediate Representation (LLVM IR) is the first quantum circuit representation in our optimization process and calling it a quantum circuit representation does not serve it justice, because it can do much more. As an intermediate representation it can describe arbitrary program code. However, it also has no special support for quantum circuits. As a result, the intermediate representation has to be filtered for instructions related to quantum circuits. This is done purely based on the function, so that QuLib functions are interpreted as quantum circuit functions, while all others are not.

An LLVM IR program consists of modules, which contain global variable and function definitions. The global variable definitions are not interesting for the quantum circuit optimization, because they are not inherently related to quantum circuit definitions. The function definitions on the other hand may contain function calls to QuLib functions that generate a quantum circuit at run time. These function calls need to be processed, so that our optimizer can find chunks, optimize the partial quantum circuit described by the chunk and then change the function calls in the LLVM IR, so that they now produce the optimized quantum circuit at run time.

Each function in the LLVM IR consists of so called basic blocks. A basic block is a list of instructions that execute sequentially. This property is important, because it means QuLib function calls within the same basic block will always be executed in a certain order. All quantum circuit optimization can only happen within a basic block, because there the optimizer can verify, that the quantum circuit is not read in between and can optimize these instructions in a way, that the the new instructions generate an equivalent circuit. This entire process is a bit more involved, because there also exist other instruction types in the basic blocks and some of them may have side effects such that QuLib function calls before and after them cannot be assumed to be in the same chunk. This topic is discussed in more detail in Section 8.2. For now it will be assumed that all QuLib calls from the same basic block also form a chunk.

Listing 8.1.: QuLib function calls can be found in the LLVM IR by iterating over the basic block and checking for call instructions that have specific function names. Given a function call, it is easy to obtain the arguments, their types and whether their value is known at compile time.

```
for (llvm::Instruction &I : BasicBlock) {
  if( llvm::isa<llvm::CallInst>(I)) {
    auto& callInst = llvm::cast<llvm::CallInst>(I);
    auto calledFunc = callInst.getCalledFunction();
    if(calledFunc) {
      std::string functionName = calledFunc->getName().str();
      if(functionName == "qulib_u") {
        std::cout << "Found qulib_u call: " << std::endl;
        for (auto& arg : callInst.args()) {
          if (auto* CI = llvm::dyn_cast<llvm::ConstantInt>(arg)) {
            std::cout << "Constant integer argument: ";
            std::cout << CI->getSExtValue() << std::endl;
          } else if (auto* CF = llvm::dyn_cast<llvm::ConstantFP>(arg)) {
            std::cout << "Constant double argument: ";
            std::cout << CF->getValue().convertToDouble() << std::endl;
          } else {
            if (arg->getType()->isIntegerTy()) {
              std::cout << "Integer variable argument." << std::endl;
            } else if (arg->getType()->isDoubleTy()) {
              std::cout << "Double variable argument." << std::endl;
            } else if(arg->getType()->isPointerTy()) {
              std::cout << "Pointer variable." << std::endl;
            } else {
              std::cout << "Unexpected type." << std::endl;
            }
          }
        }
      }
    }
  }
}
```

A basic block in LLVM can be read by iterating over its instructions. This is best explained using a basic C++ example code as seen in Listing 8.1. There are many different types of instructions in LLVM, but the QuLib function calls are all call instructions (`llvm::CallInst`), which is why the code checks whether the given instruction is a call instruction in line two. Since our optimizer needs to treat every QuLib function differently, the function being called must be determined in line four. As not every function call has an associated function, it is important to ensure that the function is not null. In the case of library functions like the QuLib functions, the function always exists, so they will never be skipped by this algorithm. The exact function can then easily be determined by the function name. The fact that the QuLib library uses a C header now becomes quite helpful, because C function names are preserved, whereas C++ function names are not. C++ uses name mangling to implement some of its features such as function overloading. As a result it cannot be assumed that a C++ function has the same name in the LLVM IR and the source code. With C functions

however, the function name will always be preserved, so that detecting QuLib functions in the LLVM IR becomes much easier. The example code then iterates over the arguments and prints different output depending on the argument type. For QuLib functions, there exist three important types: Integers to refer to qubit and classical bit indices, doubles as real valued arguments for gates and finally a pointer to the circuit being modified. There are special cases for the numerical types, because they might be statically known so that we can actually print their numerical value. By combining all the available information, specifically the function name and the function arguments, a full quantum circuit operation can be read from a call instruction in LLVM IR.

It is also possible to modify the QuLib calls in a basic block and therefore alter the quantum circuit represented by the LLVM IR. The easiest form of modification is simply deleting a call instruction. Adding a new instruction is more complicated and is shown in Listing 8.2. Before a new function can be inserted into the LLVM IR, a lot of setup is required. As already said, the module contains global variables and functions, so line 2 is simply the containing module of the function that is supposed to be modified. The context in line 3 is a helper object in LLVM to support multi-threaded modification of the LLVM IR and can be retrieved from most LLVM objects, including the module. The builder is an object that makes adding new instructions easier and controls where these new

Listing 8.2.: QuLib function calls can be added to the LLVM IR after some initial setup. The function `insertU` adds a single call to the `qulib_u` function with statically known arguments to the LLVM IR.

```
1  // general setup
2  llvm::Module*              module = ... ;
3  llvm::LLVMContext&        context = ... ;
4  llvm::IRBuilder<>*        builder = ... ;
5  llvm::Value*              circuit = ... ;
6  llvm::PointerType* circuitPtrType = llvm::PointerType::get(context, 0);
7  llvm::IntegerType*        intType = llvm::Type::getInt32Ty(context);
8  llvm::Type*            doubleType = llvm::Type::getDoubleTy(context);
9
10 // setup function
11 std::vector<llvm::Type*> uArgTypes(
12   {circuitPtrType, doubleType, doubleType, doubleType, intType});
13 llvm::FunctionType* signature = llvm::FunctionType::get(intType, uArgTypes, false);
14 llvm::FunctionCallee func = module->getOrInsertFunction("qulib_u", signature);
15
16 ...
17
18 void insertU(double theta, double phi, double lambda, int target) {
19   builder->CreateCall(func, llvm::ArrayRef<llvm::Value*>{
20                             circuit,
21                             llvm::ConstantFP::get(doubleType, theta),
22                             llvm::ConstantFP::get(doubleType, phi),
23                             llvm::ConstantFP::get(doubleType, lambda),
24                             llvm::ConstantInt::get(intType, target)
25   });
26 }
```

instruction will be inserted. The circuit is the variable in the LLVM IR that will point to the circuit being modified at run time. Before the function can be added to the LLVM IR, its signature must be defined, so that LLVM can ensure it is being called with the correct number and types of arguments. At first, the relevant types must be defined, then a vector of the argument types can be built. After that, the entire signature is defined using the return type and the vector of argument types. Finally, the function is retrieved from the module. In our case, an existing set of QuLib calls will be replaced by a different set of QuLib calls, so module, context, builder and circuit can easily be obtained based on the existing set of QuLib calls.

From these code examples, it should be clear that performing quantum circuit optimization directly on the LLVM IR is not ideal. When reading a potential QuLib call, other instruction types as well as other functions have to be considered, so that the code becomes bigger and more complicated through additional type checks, casts, and null checks. While writing new instructions seems even worse at first sight, the initial setup is only required once or once per function and inserting the function call can be achieved with very little code. However, optimization generally requires a lot more reading compared to writing, because all QuLib function calls have to be analyzed to search for potential optimization opportunities, but new instructions only have to written if an optimization is applied. Additionally, LLVM offers little type safety because the types and function signatures are run time objects, so the types will only be checked when the optimization pass runs. Incorrect function arguments will then lead to run time errors in the optimizer that will appear as compilation errors to the user. As a result, performing quantum circuit optimization directly on the LLVM IR leads to more verbose code and more potential for bugs. Finally, the LLVM IR is generated by the `Clang` compiler and therefore only available at compile time. While it is technically possible to create the LLVM IR at run time as well, this requires a lot of effort and would add a heavy dependency to the run time and hence the user program. If the LLVM IR is only available at compile time, optimization methods implemented in the LLVM IR will not be possible at run time.

With all that in mind, it is better to use LLVM as the first and last step in the optimization process and use it to read and write the program, while the quantum circuit optimization itself is performed on a different quantum circuit representation.

### 8.1.2. OpenQASM 2.0

OpenQASM 2.0 was already introduced in Section 2.3 as a textual representation for quantum circuits. It is very useful during development, because it is a human readable format and can be used to manually verify that a quantum circuit is generated correctly.

By far the biggest advantage of OpenQASM 2.0 over many other representation is the fact that it is well established in quantum computing and supported by many other tools. For that reason, QuLib returns the generated quantum circuit as OpenQASM 2.0 code, so it can easily be run by many simulators or quantum computers.

As the OpenQASM 2.0 code is just text, it can easily be generated without requiring any additional dependencies. Directly performing quantum circuit optimization on the OpenQASM 2.0 code is not desirable, because it requires a lot of string parsing to retrieve the arguments of individual gates. These arguments are necessary to decide whether certain gates cancel each other or can be replaced by another gate. Finally, OpenQASM 2.0 has no

support for symbolic variables, so quantum circuits with unknown argument values cannot be translated to OpenQASM 2.0.

### 8.1.3. QFR

The Quantum Functionality Representation (QFR) is a quantum circuit representation of the Munich Quantum Toolkit and is used as a base for other tools from the Munich Quantum Toolkit, such as QMAP. QFR is specifically required as input for QMAP in our implementation, so QFR will be discussed with the usage of QMAP in mind.

QFR represents a quantum circuit as a list of quantum operations and some general data about the circuit, such as the number of qubits or classical bits. The quantum operations are usually gates or measurements. It implements a lot of the common gate types and automatically translates generic gates into more specific ones if applicable. The list-like structure of the circuit enables easy iteration over the quantum operations. Additionally, different gate types can be distinguished efficiently using an enumerated type.

All of this makes QFR a good quantum circuit representation for optimization and QMAP even implements some optimization before and after the mapping is performed. However, the **CX** gate is the only two-qubit gate supported by QMAP and more advanced two-qubit gates must be translated into the **CX** gate and single qubit gates before the mapping can be performed. This is not an issue for our implementation, because it uses the **CX** gate as the two-qubit basis gate.

Static optimization on the other hand adds some new challenges that QFR is not designed for. Gate arguments may be variables whose values are not known at compile time. This requires symbolic variables in the quantum circuit representation. This feature is only partially supported by QFR and qubit arguments cannot be symbolic variables. As static optimization requires support for symbolic variables for all gate arguments, this requirement is marked as not supported in Table 8.1. Additionally, symbolic variables for real valued arguments is a new feature of QFR and not supported by QMAP at the time of writing.

### 8.1.4. Custom Quantum Circuit Representation

As all the above quantum circuit representations are not perfect for static optimization, it makes sense to develop a new representation that fulfills all the requirements for static optimization of quantum circuits. While it is not as advanced and sophisticated as the other representations that have been improved over many years, it is specifically designed for static optimization and hence handles it sufficiently.

Like QFR, this representation is also based on a list of quantum operations, which makes supporting QuLib function calls very easy. A QuLib function like `qulib_u` then simply adds a new quantum operation at the end of the list. The valid quantum operation types for our quantum representation are quantum gates, measurements and messages. The supported gates are currently all the gates introduced in Section 2.2, although it is easy to add new gate types.

One important aspect of our gate class is that single qubit gates inherit from the universal **U** gate. This makes writing optimization passes significantly easier, because equivalencies between individual gates can be performed implicitly. Before optimization is started, all gates are translated into the most specific equivalent gate. So if a **U** gate is equivalent to an

$\mathbf{R_x}$ gate due to its arguments, it will be replaced by the $\mathbf{R_x}$ gate. Then, if the optimizer requires specifically a $\mathbf{R_x}$ gate, it will be able to find one. On the other hand, if the optimizer requires a **U** gate, it will also find one, because every $\mathbf{R_x}$ gate is a **U** gate due to inheritance.

Still, the main requirement for our quantum circuit representation is full support for symbolic variables. There must be symbolic variables for both real and integer valued variables, so that all gate arguments can be replaced by symbolic ones. To unify the arguments of gates, a single class is used for both symbolic variables and numeric values, which is called `SymbolOrValue<T>` with a type parameter indicating whether it is a real or integer value. The optimizer might perform arithmetic on the gate arguments itself, so that a `SymbolOrValue<T>` must also be able to hold arithmetic expressions consisting of other `SymbolOrValue<T>` objects. The different kinds of arithmetic expressions used during optimization are highly specific, so that not all arithmetic expressions need to be supported. In fact, arbitrary arithmetic expressions create a problem of equality checking between two objects of type `SymbolOrValue<T>`. Consider the terms $a \cdot b + c \cdot a$ and $a \cdot (b + c)$. These terms have a completely different structure, but are equal. As a result, equality checking is not trivial. If the optimizer fails to recognize equality between complicated expressions, it might miss out on some optimization opportunities. Fortunately, the optimizer only produces very basic expressions, so that simpler equality checking is sufficient. In general, the arithmetic expression can be reduced to a sum of products of a coefficient and a symbolic variable and finally a single numeric value as offset:

$$\text{SymbolOrValue<T>} \coloneqq a_0 + \sum_{i=1}^{N} a_i \cdot s_i \tag{8.1}$$

Here, $a_i$ are numeric values, while $s_i$ is a symbolic variable. Obviously, this expression can also be single numeric value for $N = 0$ or a simple symbolic variable if $N = 1$, $a_0 = 0$ and $a_1 = 1$. By applying a strict ordering on the addends depending on the symbolic variable, the expression becomes deterministic and two expressions can easily be checked for equality. It is important to understand that this expression only affects arithmetic expression produced by the optimizer, not in the user program. The user can use an arbitrary complex expression. The return value of this expression will then be a single value in LLVM IR and can be translated to single symbolic variable. The arithmetic expression only becomes more complicated, if the optimizer itself does arithmetic. Consider the rotation folding optimization as explained in Section 3.5. The original two rotations may already both require a symbolic variable, so by merging them, the optimizer requires an argument that represents the sum of two symbolic variables.

As the other circuit representations are still necessary as input or output for third party tools, our optimizer contains translation functionality to convert other representations into our own and back. This translation is not always possible, because if the quantum circuit contains symbolic variables for the qubit arguments, it simply cannot be translated to QFR, as there exists no equivalent QFR quantum circuit.

In the following, the translation between the different representations will be discussed briefly. The translation involving LLVM IR is more complicated, because the quantum circuit is only a part of the entire LLVM IR program and the circuit might contain references to the surrounding LLVM IR code that must be preserved by the translation. This process is explained in Section 8.2.

### Translation to OpenQASM 2.0

The translation from our quantum circuit representation to other representations is implemented using the visitor pattern. The main advantage of the visitor pattern is that the implementation of the translation can be separated from the implementation of the quantum circuit representation itself. This is necessary for the translation to the LLVM IR, because the quantum circuit representation is required both at compile time and run time, while LLVM is only part of the compile time. This allows the run time to be independent from LLVM and removes a heavy dependency for the user program.

Using the visitor pattern it is very easy to translate every quantum operation into a line in the OpenQASM 2.0 code. It is not possible to produce valid OpenQASM 2.0 code for quantum circuits with symbolic variables. However, a textual representation of the symbolic variables can still be produced, which creates a human readable quantum circuit representation. This is obviously not valid OpenQASM 2.0 code, but can be useful for development and during debugging. If the quantum circuit contains no symbolic variables, the output is valid OpenQASM 2.0 code and can be used as input for many other quantum computing tools.

Translation to OpenQASM 2.0 is only required for the `qulib_compile` and `qulib_get_openqasm` functions at run time. As there are no symbolic variables at run time, the produced output will always be valid OpenQASM 2.0 code.

### Translation from OpenQASM 2.0

The translation from OpenQASM 2.0 is implemented using basic string parsing and is mostly targeted at OpenQASM 2.0 code produced by QFR. This is because the mapping tool QMAP produces OpenQASM 2.0 output based on the mapped QFR circuit. As a result, the translation algorithm only needs to supports a subset of the OpenQASM 2.0 language. The language specification of OpenQASM 2.0 only defines a very limited gate set, namely the single qubit **U** gate and the controlled **CX** gate. Additional gates can be defined similar to functions in regular programming languages. The so called quantum experience standard header is a file containing many of the most common gate definitions and is therefore very useful to avoid having to define these gates yourself. Our translation algorithm does not consider included files or user defined functions, but it contains translation rules for the quantum experience standard header. This allows it translate all OpenQASM 2.0 code produced by QFR and most of the basic OpenQASM 2.0 programs and results in a much simpler and smaller implementation of the translation algorithm. Additionally, it allows us to define custom translation rules for gates from the quantum experience standard header, which can speed up optimization. Rather than translating a $\mathbf{R_x}$ gate contained in the OpenQASM 2.0 code into a **U** gate as defined by the quantum experience standard header, it can be directly added to our quantum circuit representation and used for optimization specifically targeting the $\mathbf{R_x}$ gate, such as rotation folding. If the $\mathbf{R_x}$ gate was translated to a **U** gate, it would first need to be converted back to a $\mathbf{R_x}$ gate, before the optimization can be performed.

Finally, the library ExprTk[1] is used to evaluate mathematical expressions that can be used as real valued gate arguments in OpenQASM 2.0. These expressions can include certain

[1] `https://github.com/ArashPartow/exprtk`

constants like $\pi$ and mathematical functions like trigonometric functions. Note that these expressions are not symbolic and their numeric value can always be determined. As a result, the resulting quantum circuit in our representation will not have symbolic variables.

#### Translation to QFR

The visitor pattern can again be used to easily convert our quantum circuit representation into a QFR circuit. As QFR supports a wide array of gates, the translation is trivial and every gate in our representation can usually be converted to the same gate in QFR. Two-qubit gates are specifically translated to the basis gates **U** and **CX**, as other two-qubit gates are not yet supported by QMAP.

#### Translation from QFR

A translation from QFR back to our quantum circuit representation is not implemented, because it is not required. QFR is only used as the input to the QMAP mapping algorithm, however the output of this mapping process is the OpenQASM 2.0 code, which is why the translation from OpenQASM 2.0 has been specifically designed for the output from QMAP. Since QMAP produces the OpenQASM 2.0 code based on an internal QFR circuit, this is basically the same as translating QFR to our circuit representation indirectly by first translating QFR to OpenQASM 2.0 and then to our quantum circuit representation.

## 8.2. Replacing Quantum Circuits in LLVM

This section will explain how chunks as defined in Section 3.1 can be found and replaced in the LLVM IR. In order to explain the challenges and their solutions, code examples will be used. Note that the examples given in this section are only crafted to explain how the optimizer works and their quantum circuits have no practical use.

In the most basic case, the source code is a sequence of QuLib function calls as shown in Listing 8.3. The optimizer can easily tell, that three quantum gates form a chunk. Additionally, these function calls are surrounded by `qulib_alloc` and `qulib_compile`, so the optimizer can even realize that this quantum circuit is complete, as described in Section 3.4.

Listing 8.3.: A simple sequence of QuLib calls can be easily identified as a chunk. The presence of both `qulib_alloc` and `qulib_compile` calls allows the optimizer to verify that this chunk is complete and independent from other chunks.

```
1 char* f(void* results) {
2         auto circuit = qulib_alloc(1);
3         qulib_u(circuit, 1, 1, 1, 0);
4         qulib_u(circuit, 2, 2, 2, 0);
5         qulib_u(circuit, 3, 3, 3, 0);
6         char* qasm = qulib_compile(circuit, 1, "");
7         qulib_free(circuit);
8         return qasm;
9 }
```

Listing 8.4.: A condition that leads to optional quantum gates splits the chunk in two, because the entire quantum circuit depends on a run time condition.

```
char*  f(bool condition, void* results) {
        cCircuit circuit = qulib_alloc(1);
        qulib_u(circuit, 1, 1, 1, 0);
        qulib_u(circuit, 2, 2, 2, 0);
        if(condition) {
                qulib_u(circuit, 3, 3, 3, 0);
                qulib_u(circuit, 4, 4, 4, 0);
        }
        char* qasm = qulib_compile(circuit, 1, "");
        qulib_free(circuit);
        return qasm;
}
```

In most cases the program will be much more complicated, reducing the static knowledge of the optimizer about the generated quantum circuit. In Listing 8.4, a condition is introduced that can only be evaluated at run time. Only the first two gates are sure to be added to the quantum circuit, while the other two gates may or may not be part of the quantum circuit. As explained in Section 3.1, the optimizer could enumerate all execution paths, which would be equivalent to optimizing Listing 8.5. However, this does not scale well with more conditions and does not work with loops, so the chunk is broken up instead.

As a result, the first two gates form a chunk and the last two gates also form one. Neither of these two chunks contains a complete quantum circuit at compile time. The first chunk does not form a complete circuit, because an additional chunk may follow at run time. The second chunk is obviously not complete either, because it is always preceded by the first chunk.

Listing 8.5.: A condition can be used to decide between two chunks. This leads to more code if QuLib calls are repeated between the two choices. These repeated gates need to be optimized in both chunks, leading to longer optimization times.

```
char* f(bool condition, void* results) {
        cCircuit circuit = qulib_alloc(1);
        if(condition) {
                qulib_u(circuit, 1, 1, 1, 0);
                qulib_u(circuit, 2, 2, 2, 0);
                qulib_u(circuit, 3, 3, 3, 0);
                qulib_u(circuit, 4, 4, 4, 0);
        } else {
                qulib_u(circuit, 1, 1, 1, 0);
                qulib_u(circuit, 2, 2, 2, 0);
        }
        char* qasm = qulib_compile(circuit, 1, "");
        qulib_free(circuit);
        return qasm;
}
```

In general, only QuLib functions in the same basic block can form a chunk. Basic blocks have already been introduced as sequences of LLVM instructions. To be more specific, a basic block is a sequence of instructions which always execute as a unit in the same order. This means, only the first instruction of the basic block can be triggered externally, which will cause all functions in the basic block to run one after another without interruption until the last instruction has completed. Suppose, QuLib functions across two basic blocks were optimized as a chunk. The optimizer could then decide that the last instruction in the first basic block and the first instruction in the second basic block cancel each other and remove them. Now if the first basic block runs, but the second one does not, the generated quantum circuit will be incorrect, because it is missing one gate.

However, basic blocks are not flat lists, but hierarchical data structures. The instructions inside a basic block can itself contain basic blocks. For example, a basic block with three instructions might contain a function call, a loop and another function call afterwards. The loop body is also a basic block, so it may contain many more instructions. The function called by a call instruction also contains a basic block, adding many more instructions. As a result, finding all functions that will be called by a basic block is a complex task.

In general, a chunk is a section of a basic block, in which all QuLib calls happen equally regardless of the execution path chosen at run time. These QuLib function calls can then be used to build the quantum circuit in our quantum circuit representation at compile time. There is a single fixed compile time representation, because all QuLib calls happen equally regardless of the execution path chosen at run time. The quantum circuit is then optimized and translated back to QuLib function calls, replacing the original function calls. The requirement for the QuLib calls to happen equally regardless of execution path can be weakened, if the quantum circuit representation supports this variability. In our case, the quantum circuit representation allows symbolic variables, so the QuLib calls may have gate arguments whose values can only be determined at run time.

The requirement for all QuLib calls to happen equally regardless of execution path can be formulated more easily by defining the QuLib path. If two execution paths do not differ in terms of the QuLib functions called, they are considered the same QuLib path. Additionally, QuLib paths consider the quantum circuit they operate on, so if the quantum circuit argument of two QuLib function calls differs, two QuLib paths are created. The first QuLib path for the two quantum circuits being the same at run time and the second for the two quantum circuits being different at run time. This concept is explained using Listing 8.6 as an example. Clearly, there are four different execution paths for the different combinations of `a` and `b`. The QuLib paths are more complicated. The second condition can be ignored, because there are no QuLib calls contained in it. The other condition contains QuLib calls, leading to two initial QuLib paths. However, the QuLib call between the two conditions uses a different circuit variable. This doubles the number of QuLib paths, because either the basic block produces a single circuit or two shorter ones. With this definition of QuLib paths, a chunk can simply be defined as a section in the program with only a single QuLib path, where the reading QuLib calls are after the writing QuLib calls. As a result, the writing QuLib calls can be optimized together, knowing the generated circuit will only ever be read after the last writing QuLib call.

The chunk detection algorithm can be described implicitly, using QuLib paths. One starts with an empty chunk. New instructions are read in the LLVM IR and added to the chunk if they are on the same QuLib path. If not, a new chunk is started containing the new

Listing 8.6.: QuLib paths follow a similar idea as execution paths, but only consider the effect of instructions on quantum circuit generation. There are two conditions, leading to four execution paths. Only the first condition affects quantum circuit generation, so there are two QuLib paths initially. However, the QuLib call between the conditions uses a different circuit variable. At run time this leads to two different scenarios, where the two quantum circuit variables might point to the same quantum circuit or not. This distinction doubles the number of QuLib paths to four. The four circuit sets resulting from the four QuLib paths are shown as OpenQASM 2.0 gate sequences.

```
void f(cCircuit* circuit1, cCircuit* circuit2) {
        qulib_u(circuit1, 0, 0, 0, 0);
        if(a) {
                qulib_u(circuit1, 1, 1, 1, 0);
        } else {
                qulib_cx(circuit1, 0, 1);
        }
        qulib_u(circuit2, 2, 2, 2, 0);
        if(b) {
                printf("b");
        } else {
                printf("!b");
        }
        qulib_u(circuit1, 3, 3, 3, 0);
}
```

```
// a == true, circuit1 == circuit2
        // single circuit:
                U(0,0,0) q[0];
                U(1,1,1) q[0];
                U(2,2,2) q[0];
                U(3,3,3) q[0];
// a == true, circuit1 != circuit2
        // first circuit:
                U(0,0,0) q[0];
                U(1,1,1) q[0];
                U(3,3,3) q[0];
        // second circuit:
                U(2,2,2) q[0];
// a == false, circuit1 == circuit2
        // single circuit:
                U(0,0,0) q[0];
                CX q[0], q[1];
                U(2,2,2) q[0];
                U(3,3,3) q[0];
// a == false, circuit1 != circuit2
        // first circuit:
                U(0,0,0) q[0];
                CX q[0], q[1];
                U(3,3,3) q[0];
        // second circuit:
                U(2,2,2) q[0];
```

instruction. Since differing quantum circuit arguments always lead to multiple QuLib paths, they always start a new chunk.

This algorithm is not practical however, because determining QuLib paths is difficult and not always possible. Due to external libraries, the LLVM IR does not have access to all functions and cannot determine how many QuLib paths they contain. Function calls to unknown functions therefore stop the chunk. This is an aspect in which the optimizer could be improved in the future, because QuLib functions are currently the only external functions where the optimizer knows how they affect QuLib paths. It is difficult to prove that external functions do not affect the QuLib paths, as their code is not available in the LLVM IR. However, one could add an optional unsafe mode that assumes all external functions do not affect QuLib paths. As long as QuLib functions are not used in external libraries, this could still be sufficiently safe and a good way to increase possible chunk sizes. Additionally, a function attribute could be introduced to mark functions containing QuLib functions. A library using QuLib functions could then use this attribute to tell the optimizer which functions cannot be assumed to not affect QuLib paths.

Instead, our implementation uses a simplified approach that is designed to work well for common use cases. The instructions in the LLVM IR are divided into three types. The first type are instructions with no effect on chunks. These instructions include variable assignments or binary operators. The next type are QuLib calls, which usually have a very specific effect on the chunk and will be discussed later in more detail. The last type are all remaining instructions which are simply assumed to end the chunk. This includes any function calls that are not QuLib function calls and also all control flow statements, including loops or conditions. This means, all conditions will end a chunk, even if neither the `then` nor the `else` block contain QuLib calls.

This could be improved by performing a deep search for QuLib calls in the basic blocks of control flow statements and allow them inside the chunk, if no QuLib calls are found. However, adding instructions that do not contain QuLib calls to a chunk is only useful, if they allow further instructions to be added to the same chunk that do contain QuLib calls. This means, the control flow statement would need to be between two QuLib calls and not contain QuLib calls itself. This is rather rare, because quantum circuit generating functions are often concentrated to the same area in the program and unrelated instructions are rarely between them. Additionally, the basic blocks of control flow statements will generally contain external function calls at some point, so supporting control flow statements between QuLib calls only makes sense, if external functions are allowed to be between QuLib calls.

Our implementation implicitly supports internal function calls between QuLib calls by relying on inlining from `opt` that is triggered by the Linker Interface. Internal functions are visible in the LLVM IR and may also be inlined by regular LLVM optimization passes. Inlining achieves that function calls between QuLib calls are replaced by their function bodies, potentially increasing the chunk size. The function body will often be QuLib calls, because more advanced gates are often defined by the user based on QuLib functions and these functions are naturally more likely to appear at areas in the program where a quantum circuit is generated. These cases also work very well with the inlining of LLVM and will increase the size of the chunk, assuming the function body itself could be considered a chunk. This approach works quite well for the most common use case and does not require exact advanced analysis and potentially unsafe assumptions about whether a quantum circuit might be modified by external functions.

Listing 8.7.: The same variable is contained in both QuLib calls, though with different value. As a result, different symbolic variables must be used in the static circuit representation.

```
1  void f_original(cCircuit* circuit, void* results) {
2          double theta = x();
3          qulib_u(circuit, theta, 0, 0, 0);
4          theta++;
5          qulib_u(circuit, theta, 0, 0, 0);
6  }
7
8  void f_ssa(cCircuit* circuit, void* results) {
9          double theta_1 = x();
10         qulib_u(circuit, theta_1, 0, 0, 0);
11         double theta_2 = theta_1 + 1;
12         qulib_u(circuit, theta_2, 0, 0, 0);
13 }
14
15 void f_opt(cCircuit* circuit, void* results) {
16         double theta_1 = x();
17         // merged qulib_u(circuit, theta_1, 0, 0, 0);
18         double theta_2 = theta_1 + 1;
19         // merged qulib_u(circuit, theta_2, 0, 0, 0);
20         qulib_u(circuit, theta_1 + theta_2, 0, 0, 0);
21 }
```

As already said, certain instructions do not affect the quantum circuit and can be inside chunks without causing problems. This includes variable assignments. However, allowing variable assignments between QuLib calls creates another challenge shown by the function `f_original` in Listing 8.7. While increasing `theta` does not lead to multiple QuLib paths and also does not directly modify the quantum circuit, naively using the two QuLib calls to build a quantum circuit at compile time will not work, because then the `theta` argument would be the same in both gates. Fortunately, LLVM uses a concept called static single assignment, which means that every variable is only ever assigned once. LLVM automatically enforces static single assignment when converting the source code to the LLVM IR, but function `f_ssa` shows how the source code can be changed to conform to static single assignment. Suddenly the problems disappear, because the two quantum gates now use completely separate variables. Coincidentally, the **U** gate with parameters $\mathbf{U}(\theta, 0, 0)$ is equivalent to the $\mathbf{R}_y(\theta)$ gate. Since this is a rotational gate, consecutive applications can be merged by adding up the angles. As both QuLib calls can be optimized together, the code can be optimized to function `f_opt`. It is important to ensure that the merged QuLib call happens after all the variable assignments it depends on. The static single assignment of LLVM again comes in useful. Within a chunk, all QuLib calls can be moved to the end. With static single assignments, executing a function later cannot change its arguments, because every variable is only assigned a single value.

Theoretically, the function could still act differently, because an argument could be a pointer and the referenced value could be altered, indirectly changing the input to the function. However, it is important to remember that only QuLib calls that add quantum

operations need to be optimized and replaced, so only their arguments could be affected. These function only have a single pointer parameter, namely the circuit pointer, which is a pointer to a struct with a void pointer to the quantum circuit object. Theoretically it is possible to manually modify the void pointer or even the quantum circuit object using manual memory management, which would break assumptions by the optimizer. While this is currently no problem, because the chunk would be split at any function call that could tamper with the struct, this is a general problem when larger chunks are desired and external functions are allowed between QuLib calls. In the end, modifying the struct or the quantum circuit without QuLib calls is not intended by QuLib, so undefined behavior is expected.

The final chunk detection algorithm is given by Algorithm 5. All member variables of the chunk are initially false or null depending on the type. The core of the algorithm is the function `ProcessInstruction`, which returns whether an instruction continues the current chunk or if a new chunk must be created. Additionally, it updates the chunk's information about the quantum circuit generated by it. In particular, instructions writing to the quantum circuit are stored. The `qulib_alloc` function adds the number of qubits and tells the chunk, that it is the beginning of a circuit by setting `chunk.allocated` to true. The `qulib_compile` function sets `chunk.compiled` to true, telling the chunk that it is the end of a circuit. The outer function `ProcessBasicBlock` has a somewhat surprising loop body, because `ProcessInstruction` is performed another time, if it fails in the condition. This is necessary, because the return value of `ProcessInstruction` depends on the state of the chunk and creating a new chunk changes its state. More specifically, an instruction might not be able to continue an existing chunk, but be the first instruction in a new chunk.

The decision when a new chunk starts is more complicated and depends on the different instructions. Instructions that definitely do not contain QuLib calls can be ignored by the optimization and always continue a chunk. As already mentioned, this includes variable assignments and binary operators, although further analysis of instructions could be performed to add more instructions to this category. The `qulib_simulate` function also belongs to this group, because it does not directly act on the quantum circuit object and is just a utility function to start the Quantum++ simulator. Instructions that read the circuit set `chunk.read` to true, indicating that the chunk may no longer accept any write operations. This is required, because the entire write operations are optimized together and the intermediate quantum circuit state may be incorrect compared to the original quantum circuit, so that read operations may only appear after all write operations. Instructions that write to the circuit, are therefore not allowed, if `chunk.read` is true. Additionally, write instructions must operate on the same circuit as the chunk. If the chunk has no circuit yet, because it is a new chunk, the write operation is allowed as well and sets the chunk's circuit to the circuit being modified by the instruction. The `qulib_free` function always ends a chunk, because it can lead to erroneous behavior at run time if a circuit is still being used by QuLib functions after it is freed. `qulib_alloc` always starts a new chunk, but the chunk is not directly created here, resulting in more complicated behavior. The first time this function is encountered, the chunk is usually not empty and `ProcessInstruction` returns false to end the chunk. The outer function then creates a new chunk and invokes `ProcessInstruction` again. This time the chunk is empty, so that the `qulib_alloc` call can initialize the chunk with its information and return true to continue on this new chunk. Finally, `qulib_compile` always ends the chunk, because no further write operations are allowed. While read operations

**Algorithm 5:** Algorithm to detect chunks in LLVM IR basic blocks. The ProcessInstruction function returns, whether the chunk can continue after the given instruction and may also add some instructions to the chunk, if the instruction adds a quantum operation to the quantum circuit. The ProcessBasicBlock function iterates over a basic block and finds all the chunks in it. Once a chunk is complete, it is optimized.

```
1  Function ProcessBasicBlock(basicBlock):
2      chunk = new() for instr : basicBlock do
3          if not ProcessInstruction(chunk, instr) then
4              OptimizeChunk(chunk)
5              chunk = new()
6              ProcessInstruction(chunk, instr)
7      OptimizeChunk(chunk)
8  Function ProcessInstruction(chunk, instr):
9      if DoesNotAffectCircuit(instr) then
10         return true
11     if IsReadOperation(instr) then
12         chunk.read = true
13         return true
14     if IsWriteOperation(instr) then
15         if chunk.read == false and (chunk.circuit == null or instr.circuit ==
            chunk.circuit) then
16             AddToChunk(chunk, instr)
17             chunk.circuit = instr.circuit
18             return true
19         else
20             return false
21     if instr.name == "qulib_free" then
22         return false
23     if instr.name == "qulib_alloc" then
24         if chunk.size == 0 then
25             chunk.circuit = instr.circuit
26             chunk.qubits = instr.qubits
27             chunk.allocated = true
28             return true
29         else
30             return false
31     if instr.name == "qulib_compile" then
32         chunk.compiled = true
33         return false
```

would still be possible, they are not optimized anyway, so they do not have to be part of the chunk.

The overall sequence of a chunk is now an optional `qulib_alloc` call, followed by write operations, followed by read operations with an optional `qulib_compile` call at the end. And the `qulib_compile` call is the only reason, why read operations are even handled specifically by this algorithm. It would be possible to simply optimize the sequence of write operations as its own chunk and end the chunk at the first read. The quantum circuit that needs to be optimized would be the same. However, it is missing the information whether it is the end of a quantum circuit. By specifically handling read operations, the chunk detection algorithm is allowed to continue with the chunk and potentially find the `qulib_compile` call at the end.

A chunk consists now of a sequence of write instructions and some additional information about the chunk, namely whether it is known to be the start or end of a quantum circuit and maybe also the number of qubits. The write instructions are the LLVM instructions as they are contained in the basic block, but they are not contiguous in the basic block, because other instructions might be between the write instructions. The write instructions are now translated to our quantum circuit representation using a similar approach as shown in Listing 8.1 to parse them. Gate arguments are translated into objects of type `SymbolOrValue<T>`, which was explained in Subsection 8.1.4. There are no arithmetic expressions yet, because all arguments in LLVM are of type `llvm::Value`, which is just a single symbolic variable. In some cases, the numeric value is even known at compile time and can be passed to our quantum circuit representation. For values that are symbolic variables, a pointer to the `llvm::Value` object is stored inside the `SymbolOrValue<T>` object, so that the LLVM variable can be used when the quantum circuit is translated back into LLVM IR.

After that, the quantum circuit is in our quantum circuit representation and can be optimized, as explained in Section 8.3 and Section 8.4. Once this is done, the quantum circuit must be translated back into LLVM IR. The optimization might transform the quantum circuit so much, that it no longer resembles the original one and it is not clear where the non-QuLib instructions between the original write instructions should now be placed between the potentially completely different optimized write operations. As already mentioned, the write operations can be moved to the end of the chunk, because of the static single assignment of LLVM and the fact, that the quantum circuit only needs to be in a valid state after the last write operation. The translation itself is then rather easy and similar to Listing 8.2, although translating `SymbolOrValue<T>` objects requires some additional steps. Still, the sum shown in Equation 8.1 can be directly calculated in LLVM IR by using the `llvm::Value` objects stored in the symbolic variables inside each `SymbolOrValue<T>` object.

Finally, the optimizer will add some calls to `qulib_message` to transfer some information about the static optimization to the run time. The first message is inserted after every `qulib_alloc` call and contains content from the configuration file. This configuration file is used by the static optimization, but is not directly available at run time, so its content is included in the program code as a message. By inserting it directly after the quantum circuit allocation, every circuit will have access to this information at run time. The second message is inserted before every `qulib_compile` call and tells the run time to which extent the quantum circuit was optimized already, so that repeated optimization can be avoided at run time in some cases. The run time optimization is explained in more detail in Section 8.5.

## 8.3. Optimization of Quantum Gate Sequences

As explained, the chunk is now just a sequence of write instructions and some general information about the chunk. This general information includes whether the chunk is the start or the end of a quantum circuit and also the number of qubits, if it is the start of a quantum circuit. If the chunk is both the start and end of a quantum circuit, it is considered complete as explained in Section 3.4 and special optimization steps can be performed, which are explained in Section 8.4. If the chunk is only the end or only the start of a quantum circuit, there would still be special optimization methods possible, but they have not yet been implemented.

There are still a lot of optimization steps possible for arbitrary sequences of quantum gates as explained in Section 3.5 and their implementation will be explained in this section. The optimization process can be divided into several passes.

The specialization pass converts generic gates into more specific versions, if their arguments allow it. The optimizer iterates over the quantum gates and looks for generic **U** gates. Then for every specialized gate, such as $\mathbf{R_x}$, $\mathbf{R_y}$ and $\mathbf{R_z}$, the optimizer checks whether the **U** gate is equivalent given its arguments. If that is the case, the gate is replaced by the more specialized gate. Due to the inheritance in the quantum operation classes, the specialized gates still inherit from **U** gates, so that anything that relies on **U** will still be able to work with the more specialized version. However, some peephole optimization methods specifically require the more specialized gates, so that this process is necessary.

The peephole optimization pass includes all the individual peephole optimization techniques as explained in Section 3.5. These are very easy to implement using our quantum circuit representation, as usually only the types and arguments of two neighboring quantum operations need to be considered. An example for the rotation folding of the $\mathbf{R_x}$ gate is shown in Algorithm 6. At first, two consecutive quantum operations are obtained from the list and cast to the relevant gate type. If they are not the correct gate type, the cast will return `null` and the next condition will fail. If both operations are $\mathbf{R_x}$ gates, they must also target the same qubit to be merged. The merge process is easy to write down

**Algorithm 6:** Implementing a single peephole optimization is extremely simple in our quantum circuit operation. The gates within the peephole are obtained from the list and checked for their gate type. If the gate types are correct, the gate arguments must also be checked. After that, the optimization can be applied.

```
1 Function RXGateFolding(chunk):
2     for i = 1 to chunk.size - 1 do
3         rCurrent = chunk.operation[i] as RXGate
4         rPrev = chunk.operation[i - 1] as RXGate
5         if rCurrent != null and rPrev != null then
6             if rPrev.target == rCurrent.target then
7                 chunk.operation[i] = RXGate(rPrev.theta + rCurrent.theta,
                   rCurrent.target)
8                 chunk.operation[i - 1].delete()
```

due to our quantum circuit representation and specifically the `SymbolOrValue<T>` type. This type supports addition between two arguments, whether they have numeric values or are symbolic variables. The process of adding, removing or deleting quantum operations in the list depends on the underlying data structure. In our case, a `std::vector` is used, so that the deletion would require adjusting the indices of following quantum operation. To prevent too frequent index shifts, the regular peephole optimization replaces deleted gates by special `null` gates. At the end of the peephole optimization pass, all `null` gates are deleted, resulting in much fewer index shifts.

The basis gate pass converts all quantum operations into the basis gates **U** and **CX**. This pass has two purposes. During optimization, the translation into the basis gates can offer new optimization opportunities for the peephole optimization pass. After the optimization is completed, all gates are converted to basis gates, so the number of basis gates can be counted. This number is then compared to the number of basis gates in the original quantum operation sequence to determine the effectiveness of the entire optimization. The resulting overall optimization process is shown in Algorithm 7 and relies on repeated application of the passes. At first, the peephole pass with prior specialization is repeated until it no longer modifies the chunk. After that, all gates are translated to basis gates and the peephole pass without prior specialization is repeated until the chunk no longer changes. This is necessary, as the translation to basis gates might reveal new optimization possibilities for the peephole pass. The specialization pass is not performed prior to the peephole pass this time, because it would reverse the translation into basis gates. Finally, all gates are again translated into basis gates, because the peephole optimization pass could have produced gates that are not basis gates. This results in the final optimized chunk, containing only basis gates.

**Algorithm 7:** Overall optimization of a chunk. The process is based on iterative application of optimization passes until they no longer change the chunk.

```
1  Function OptimizeChunk(chunk):
2      do
3          SpecializationPass(chunk)
4          PeepholePass(chunk)
5      while chunk was changed
6      BaseGatePass(chunk)
7      do
8          PeepholePass(chunk)
9      while chunk was changed
10     BaseGatePass(chunk)
```

The translation into basis gate sometimes requires arithmetic operations on the gate arguments and is the reason why the `SymbolOrValue<T>` type contains coefficients for symbolic variables. So far, only rotation folding added two gate arguments together, which can be implemented without the coefficients as a simple sum of symbolic variables and a numeric offset. The translation of the **CU** gate into basis gates is shown in Algorithm 1 and requires dividing gate arguments by two, negation, subtraction and addition. All of these operations can be applied to the `SymbolOrValue<T>` type by performing the same operations on all its coefficients and the numerical offset.

## 8.4. Optimization of Complete Quantum Circuits

Complete quantum circuits within a chunk allow additional optimization as explained in Section 3.4. This was not a focus of our implementation and was primarily included to showcase how both complete quantum circuit optimization and target specific quantum circuit optimization can be implemented within this optimization framework.

The only implemented optimization that is specific to complete quantum circuits is the mapping to target architectures with QMAP. As symbolic variables are a rather new addition to QFR, they are not yet supported by QMAP. This means mapping is only possible if all gate arguments are known at compile time. However, QMAP is still in active development, so future support for symbolic variables is likely. The target architecture must obviously be specified at compile time, so that the quantum circuit can be mapped to it. This is done inside a configuration file and the target architecture can be left empty to perform no target specific optimization.

If the target architecture is provided, the chunk is complete and it contains no symbolic variables, our quantum circuit representation can be translated into QFR as explained in Subsection 8.1.4 and mapped with QMAP to the target architecture. The resulting OpenQASM 2.0 code is then translated back into our own quantum circuit representation. To simplify the task for QMAP, the quantum circuit is optimized as explained in Section 8.3 before it is translated into QFR. As the mapping may significantly alter the quantum circuit, the regular optimization will be repeated after the mapping process is completed.

## 8.5. Run Time Optimization

All of the optimization methods also work at run time and can be triggered with `qulib_compile`. The only difference is that the quantum circuit is always complete and contains no symbolic variables, allowing more optimization. Regular optimization and mapping to target architectures can be toggled individually and happens mostly independently from static optimization. There are some interactions between static optimization and run time optimization that are realized by messages inserted by the static optimizer with `qulib_message`.

If a chunk was already mapped at compile time using QMAP, it will not receive any optimization at run time. This is because QMAP already requires a complete quantum circuit without symbolic variables, so that no further optimization will be possible at run time. This is done with the message `compiled=-2` before the `qulib_compile` call, which tells the run time to perform no optimization or mapping at all.

In that case, the architecture argument of the `qulib_compile` call would be ignored, because the configuration file could provide a different architecture than the function argument. However, not all chunks may be able to be mapped at compile time and it is not clear for the programmer which quantum circuits will be mapped at compile time. As a result, the architecture argument of all quantum circuits should be overridden by the configuration file at compile time, so that the architecture to which a quantum circuit is mapped does not depend on how much information the optimizer could gather about the quantum circuit at compile time. So for all circuits that could not be mapped at compile time, the optimizer will insert the message `compiled=-1` before the `qulib_compile` call, telling the run time to use the architecture in the configuration file over the function argument. The configuration

file is indirectly available at run time via a message inserted after every `qulib_alloc` call. This message contains the content of the configuration file that is required at run time.

In case no architecture is provided in the configuration file, target specific optimization must still be performed at run time and the architecture argument will be used instead. In that case, the message `compiled=-2` will never appear, because no mapping is possible at compile time without a target architecture in the configuration file. Then all quantum circuits will contain the message `compiled=-1` and the run time will see that the configuration file has no architecture and use the function argument instead.

Finally, if no static optimization is performed at all, no messages are inserted and the architecture argument will always be used. As a result, QuLib will also work without static optimization and can be used as a regular C library with the regular compilation process.

# 9. CMake Integration

To include our static quantum circuit optimization in a C or C++ program, the compilation process needs to be adjusted. Obviously, the QuLib library needs to be linked, though this is not different from other C libraries. Additionally, the Clang compiler needs to be used with special compile and link flags. The LLVM Clang compiler must be instructed to produce LLVM bitcode files, which is accomplished by adding `-flto` to both compile and link flags. Furthermore, the linker executable used by LLVM Clang must be replaced by our Linker Interface by setting the link flag `-fuse-ld=...` to the path of the Linker Interface.

Additionally, the Linker Interface requires some arguments to configure the optimization process. This includes the target architecture, but also paths to the LLVM tool chain. LLVM Clang passes arguments to the linker by setting the link flag `-Xlinker ...`, but this is just the path to a directory containing configuration files in our case. As compiler commands can become quite lengthy, this approach reduces all our arguments to a single one and all configuration can happen inside specific files. The configuration is based on a project file and a user file. The project file is supposed to be shared between collaborators and may include settings such as the target architecture that should be the same for all collaborators. The user file can be used to override the project file and is not supposed to be shared. There, device specific settings such as the path to the LLVM tool chain can be specified.

Finally, regular optimization is disabled, because it will be triggered manually in the Linker Interface. This is achieved by the compile flags `-O1 -Xclang -disable-llvm-passes`.

All of this can be performed manually, though it leads to quite complicated compiler commands. A build system like CMake may be used to simplify this process. Our implementation is designed as a relocatable CMake package [Kit23], so that it can easily be installed to other devices. This will install the Linker Interface, optimization pass, QuLib library and some CMake files, including a `.cmake` file, at a specified installation path. The programmer only has to include this `.cmake` and has immediate access to the QuLib library and a custom CMake function that enables static quantum circuit optimization by correctly setting all link and compile flags of LLVM Clang. Additionally, the programmer must ensure the LLVM Clang compiler is used by CMake, otherwise no static quantum circuit optimization will be performed. This also makes it easy to test QuLib without static quantum circuit optimization by either using another compiler or commenting out the CMake function that sets the link and compile flags of LLVM Clang. Finally, the static quantum circuit optimization relies on the LLVM tool chain, which must be installed by the programmer and its path provided in the configuration file. All other tools described in Chapter 5 are dependencies of our implementation and only need to be present during the compilation of our optimizer.

# 10. Implementation Details

Our implementation consists of three main components, namely the library QuLib, the Linker Interface as an executable and the LLVM optimization pass, which must be a shared library, so it can be loaded by `opt`. All of these components share a lot of functionality, because both QuLib and the optimization pass must operate on different quantum circuit representations. For this reason, the majority of the quantum circuit logic and many utility function are separated in a shared library that all components rely on. This library will be called SharedLib.

However, the LLVM optimization pass is compiled without exceptions and run time type information (RTTI) [LLV23], so that any headers included by the optimization pass must follow these requirements, particularly headers from SharedLib. This is problematic, because many headers will be included due to transitive includes. The QFR library for instance use exceptions in some of their headers, which can therefore not be included in headers of SharedLib. The solution is to only include QFR in the source files. However, if QFR is not part of the header, SharedLib cannot define functions directly operating on QFR data types and interface functions with custom types must be defined instead. These interface functions allow all components to indirectly access QFR, without directly including QFR headers.

The lack of RTTI in the LLVM optimization pass means class types defined in headers of SharedLib will also not have RTTI and `dynamic_cast<>` will not work. While LLVM implements its own form of RTTI that can be added to types manually, this is also not practical, because SharedLib must stay independent from LLVM, so that QuLib and the user program do not require LLVM. As a result, the quantum operation classes inside SharedLib implement their own dynamic cast functions, so that casting between gate types is possible.

# Part III.

# Results

In the following, the implementation will be evaluated in terms of its effectiveness and running time. As optimization is possible at compile time and run time, there are four different scenarios that will be compared. There can be no optimization at all, only compile time optimization, only run time optimization or optimization at both compile time and run time.

The evaluation is based on an implementation of Shor's algorithm [Sho94], which computes a non-trivial divisor of an integer $N$. There are certain limitations for the input of Shor's algorithm. Obviously, $N$ must be composite, so that a non-trivial divisor exists. Additionally, $N$ must be odd and must not be the power of an odd prime number. Shor's algorithm is asymptotically faster than classical algorithms that perform the same task. However, current quantum computers are far behind classical ones and do not support large integers as input to Shor's algorithm [ASK19]. For these small inputs, the asymptotic running time is not important, so that classical algorithms outperform Shor's algorithm. Shor's algorithm becomes even slower when no quantum computer is available and the quantum computation is simulated. As this implementation of Shor's algorithm is run on the simulator Quantum++, only small inputs can be tested efficiently. With the input limitations outlined above, $N = 15$ is the smallest valid input and our implementation still achieves reasonable computation times with Quantum++. The next valid input $N = 21$ on the other hand is already problematic and can no longer be computed in acceptable time. For this reason, our implementation was only tested for $N = 15$.

If a program is implemented only using basis gates, there will be little room for optimization, if the programmer already generates an optimal sequence of basis gates. However, large quantum circuits consist of thousands of basis gates, so only using basis gates results in a long and complicated program. The solution is to define advanced gates that consist of several basis gates and define the quantum circuit as a combination of basis gates and advanced gates. QuLib already defines the **CU** gate as an advanced gate consisting of six basis gates and the quantum experience standard header of OpenQASM 2.0 contains many more definitions for advanced gates. Some of these gates have been defined using the QuLib functions `qulib_u` and `qulib_cu` in the implementation of Shor's algorithm.

As already said, using advanced gates makes the program smaller and easier to understand. Hence, advanced gates are usually preferred over basis gates. However, using advanced gates may result in non-optimal quantum circuits in some cases. Inserting an advanced gate with specific arguments may enable some optimization on the underlying basis gates. Additionally, the underlying basis gates at the border of two consecutive advanced gates may be optimized together. Manual optimization is not desired in these cases, because it requires replacing the advanced gates by many basis gates. In these cases automatic quantum circuit optimization becomes extremely useful, because it allows writing concise and simple programs with advanced gates, while still benefiting from optimization on the underlying basis gates.

# 11. Optimization Correctness

The correctness of our implementation of Shor's algorithm for $N$ was tested by repeatedly running the program and simulating the generated quantum circuits with Quantum++ by calling `qulib_simulate`. In these tests, the implementation was always able to find a non-trivial divisor of $N$. Obviously, this is no definitive proof that the implementation is correct and will always work. However, a program that returns the correct results is not necessarily required for our use case of optimization. The optimizer does not care whether the program makes sense. All that matters is that the optimizer produces an optimized program with the same behavior as the original program. This implementation of Shor's algorithm was then optimized with different combinations of compile time and run time optimization. The optimized program was tested again and found to produce the same results as the original program.

For $N = 15$, there are four distinct results from the quantum computation that all are supposed to have the same chance, while all other results never occur. In general, there are a total of $2^8 = 256$ possible results, as 8 qubits must be measured. A small change in probabilities will not be noticeable given the low number of performed tests. Additionally, repeated program execution can never prove the original and optimized quantum circuit are equivalent, but can be used to quickly detect if the optimized quantum circuit produces wrong results.

The correctness of the optimization is checked by testing the original and optimized quantum circuit for equivalence with the Quantum Circuit Equivalence Checking (QCEC) tool from the Munich Quantum Toolkit. Similar to QMAP, this tool exists both as a C++ and Python library. While QMAP is directly included in our implementation as a C++ library, QCEC is only required during the evaluation. Therefore, QCEC is used in Python, which makes it easier to use OpenQASM 2.0 code as input and therefore works well with the OpenQASM 2.0 code generated by our implementation of Shor's algorithm. QCEC then confirms that the OpenQASM 2.0 codes generated by the original and optimized programs are indeed equivalent.

# 12. Optimization Effectiveness

The effectiveness of the optimizer is measured by comparing the number of basis gates before and after the optimization. Our implementation of Shor's algorithm contains 13306 basis gates for $N = 15$. However, most of the basis gates are included indirectly by advanced gates and many gates are added inside loops, so the program is not as large as the number of basis gates may indicate.

Enabling static quantum circuit optimization brings the number of basis gates down to 8879. Run time optimization on the other hand achieves an even better result of 6742 basis gates. It is worth noting that this number is reached with and without static quantum circuit optimization. In fact, performing both static optimization and run time optimization leads to the same quantum circuit as applying only run time optimization. This is an important result and shows that the static optimization does not perform a transformation that makes it difficult for the run time optimization to find all optimization opportunities.

However, this may not always be the case and depends on the optimization methods that could be applied at compile time. As more advanced peephole optimization techniques are added, the chance of static optimization weakening run time optimization increases. Suppose there are instructions $a$, $b$, $c$, $d$ with costs $a = 2$, $b = 2$, $c = 3$, $d = 3$ and $ab$ can be optimized to $c$, $ba$ to $d$, while $bb$ can be eliminated. Clearly, a chunk containing $ab$ should be optimized to $c$ to reduce the cost from $2 + 2 = 4$ to 3 and similarly, $ba$ should be optimized to $d$. However, at run time the chunks might happen after another, resulting in the run time sequence $abba$ with cost $2 + 2 + 2 + 2 = 8$. By removing $bb$, the cost can be reduced to $2 + 2 = 4$, compared to the static optimized sequence of $cd$ with cost $3 + 3 = 6$. In this case, it would be possible to add another optimization from $cd$ to $aa$. In reality there will be many more peephole optimizations, so that interactions between all of them are not necessarily covered and an optimization like $cd$ to $aa$ might not be implemented.

This will not happen with the currently implemented peephole optimization methods, because they all operate on equal gates and do not insert gates of another type. This means, only sequences of the same gate can be optimized. Every sequence of equal gates can be treated individually, because optimization between different gate types is not possible. The order of optimization within a sequence does not matter either, because every optimization reduces the number of gates within the sequence by one or two, depending on if the optimization merges two gates or eliminates two canceling gates.

The static optimization already achieves a quite formidable reduction in basis gates for Shor's algorithm and reduces it by around 33%. The run time optimization performs even better and almost halves the number of basis gates. Obviously, these numbers depend largely on the implementation and higher level implementations using more advanced gates will generally offer more room for optimization than sequences of basis gates, because there the programmer can already perform optimization between nearby gates. This indicates that quantum circuit optimization will become even more important in the future, as more and more high level programming will be adopted.

# 13. Optimization Running Time

The previous chapter about reducing the number of basis gates, showed that run time optimization performs better than compile time optimization, even if compile time optimization is already a significant improvement over no optimization at all. This raises the question whether compile time optimization should ever be used. The advantage of compile time optimization over run time optimization is potentially faster run time performance, as optimization after the quantum circuit generation is avoided.

How important compilation speed is, depends on the specific application. In general, trading faster run time performance for longer compilation time is desirable, even if the compilation time increase is much more significant than the performance increase at run time. Fast compilation speed was not a priority for our implementation, so there are certainly improvements possible. Table 13.1 shows the times for different parts of the compilation process. Overall the compilation takes a lot longer with static quantum circuit optimization, requiring more than 14 seconds compared to less than a second without static quantum circuit optimization.

The quantum circuit optimization itself is very fast and only takes around 200ms at compile time. At run time, the quantum circuit optimization showed a lot of variance, though the average optimization time over 20 runs was 97ms with static quantum circuit optimization and 179ms without. These results make sense given the number of basis gates in the quantum circuits before run time optimization. Without static quantum circuit optimization, 13306 gates need to be optimized at run time. By already performing some optimization at compile time, only 8879 gates need to be optimized at run time. This shows that static quantum circuit optimization can improve performance at run time. Another option is to not perform run time optimization and only rely on static quantum circuit optimization. The resulting quantum circuit will not be optimal, but the optimization time is saved.

| Name | Time in ms |
|---|---|
| Processing Arguments | 468 |
| Merge LLVM Bitcode File (`llvm-link`) | 494 |
| Regular LLVM Optimization O3 (`opt`) | 3796 |
| Custom Optimization Pass (`opt`) | 1172 |
| Regular LLVM Optimization O3 (`opt`) | 3800 |
| Linking | 4647 |

Table 13.1.: Time spent for different parts of the optimization process triggered by the Linker Interface. Only around 200ms of the custom optimization pass is spent on our implementation of the optimization pass. It seems most of the time is spent by `opt` itself.

However, static optimization seems not worth it for this implementation. This is mainly because the quantum circuit simulation with Quantum++ takes far longer than any run time optimization. As a result, saving up to 200ms at run time will not be noticeable. On the other hand, the significantly longer compilation time is definitely noticeable.

As quantum computers and simulators become faster, the small run time performance gained by reducing optimization time might become more significant. Additionally, the peephole optimization included in this implementation are very basic and there are much more optimization techniques possible, so that optimization will be slower, but more effective and it again becomes more important to move some of that optimization time to the compile time.

# Part IV.

# Conclusion

An implementation of static quantum circuit optimization based on the LLVM tool chain was presented. It was shown how multiple quantum circuit generating functions form a chunk that can be optimized at compile time. By reusing existing classical optimizations, such as inlining, the static optimizer can find larger chunks more easily, improving the effectiveness of the quantum circuit optimization. As the optimizer must be able to detect quantum circuit generating functions, a specific set of functions must be used for this task and the optimizer must be specifically designed for these functions. Therefore, the QuLib library was introduced. By defining a new library, both the library and optimizer can be designed to collaborate well.

Static quantum circuit optimization requires a specific quantum circuit representation that can encode variability caused by run time dependent values. It makes sense to implement this representation to also work at run time, so that QuLib can reuse this representation and perform run time optimization on it. Peephole optimizations on quantum gates work well for chunk optimization, as they do not require the entire quantum circuit to be known and they can easily be implemented on the custom quantum circuit representation. The usage of LLVM as intermediate representation requires translation algorithms between its intermediate representation and our quantum circuit representation.

In many practical applications, quantum circuits are defined using advanced gates, as it is easier for the programmer and results in less lines of code. The optimizer takes care of the optimization on the basis gate level, so that the resulting quantum circuit still uses a low number of basis gates. Most of the optimization can already be done at compile time, but optimization can still be performed at run time to reduce the number of basis gates even further.

However, there are still many ways in which this implementation can be improved. By performing more analysis on instructions, larger chunks can be found, resulting in more optimization opportunities at compile time. In particular, many function calls currently force a chunk to be split, because the called function could interact with the generated quantum circuit. If the optimizer could verify, that the called function never interacts with the quantum circuit, the chunk would not need to be split.

The different optimization methods performed on the quantum circuit representation are very basic and the effectiveness of the optimizer could be improved by implementing additional optimization methods. This poses a challenge, because most research on quantum circuit optimization assumes the circuit is fully known. In static optimization, the optimizer often has no knowledge about the surrounding gates of a chunk and gate parameters may also be unknown at compile time.

Finally, more target specific optimization can be added. Different quantum computers use different gate sets and have different error rates attached to specific basis gates. The optimizer could use this information to generate optimal gates for the target quantum computer. Performing quantum circuit mapping at compile time is a difficult task, as mapping algorithms generally require the entire circuit to be known and more research is required to find a good strategy for static quantum circuit mapping.

# Bibliography


[AMM14] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014.

[AMMR13] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013.

[ASK19] Mirko Amico, Zain H. Saleem, and Muir Kumph. Experimental study of Shor's factoring algorithm using the IBM Q Experience. *Physical Review A*, 100(1), 2019.

[BKM+14] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, C. Neill, P. O'Malley, P. Roushan, A. Vainsencher, J. Wenner, A. N. Korotkov, A. N. Cleland, and John M. Martinis. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature*, 508(7497):500–503, 2014.

[Bor26] Max Born. Zur Quantenmechanik der Stovorgnge. *Zeitschrift fr Physik*, 37(12):863–867, 1926.

[BRSW21] Lukas Burgholzer, Rudy Raymond, Indranil Sengupta, and Robert Wille. Efficient Construction of Functional Representations for Quantum Algorithms. 2021.

[BW21] Lukas Burgholzer and Robert Wille. Advanced Equivalence Checking for Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(9):1810–1824, 2021.

[CBSG] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language.

[DM16] Olivia Di Matteo and Michele Mosca. Parallelizing quantum circuit synthesis. *Quantum Science and Technology*, 1(1):015003, 2016.

[Gam22] Gambetta, Jay. Quantum-centric supercomputing: The next wave of computing, IBM Research Blog. `https://research.ibm.com/blog/next-wave-quantum-centric-supercomputing`, 2022.

[Ghe18] Vlad Gheorghiu. Quantum++: A modern C++ quantum computing library. *PloS one*, 13(12):e0208073, 2018.

[HHT20] Thomas Häner, Torsten Hoefler, and Matthias Troyer. Assertion-based optimization of Quantum programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–20, 2020.

[Hid19] Jack D. Hidary. *Quantum Computing: An Applied Approach.* Springer International Publishing, Cham, 2019.

[IHKH22] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing*, 3(3):1–32, 2022.

[JPK+14] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC. In Pedro Trancoso, Diana Franklin, and Sally A. McKee, editors, *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, New York, NY, USA, 2014. ACM.

[JTS+22] Wonho Jang, Koji Terashi, Masahiko Saito, Christian W. Bauer, Benjamin Nachman, Yutaro Iiyama, Ryunosuke Okubo, and Ryu Sawada. Initial-State Dependent Optimization of Controlled Gate Operations with Quantum Computer. *Quantum*, 6:798, 2022.

[Kit23] Kitware, Inc. and Contributors. CMake Documentation 3.26.3: cmake-packages(7): Creating Relocatable Packages. `https://cmake.org/cmake/help/v3.26/manual/cmake-packages.7.html#creating-relocatable-packages`, 2023.

[KM13] Vadym Kliuchnikov and Dmitri Maslov. Optimization of Clifford circuits. *Physical Review A*, 88(5), 2013.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pages 75–86. IEEE, 2004.

[LAB+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[LBZ21] Ji Liu, Luciano Bello, and Huiyang Zhou. Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 301–314. IEEE, 2021.

[LLV23] LLVM Project. LLVM Coding Standards. `https://llvm.org/docs/CodingStandards.html#do-not-use-rtti-or-exceptions`, 2023.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* Cambridge University Press, Cambridge, 1. publ edition, 2000.

[Qis23] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[Ros69] Saul Rosen. Electronic Computers: A Historical Survey. *ACM Computing Surveys*, 1(1):7–36, 1969.

[SBM06] V. V. Shende, S. S. Bullock, and I. L. Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, 2006.

[SDC+21] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, 2021.

[SEL04] PETER SELINGER. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Comput. Soc. Press, 1994.

[SHT18] Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.

[SP08] Michal Sedlák and Martin Plesch. Towards optimization of quantum circuits. *Open Physics*, 6(1):128–134, 2008.

[SPSD20] R. S. Smith, E. C. Peterson, M. G. Skilbeck, and E. J. Davis. An open-source, industrial-strength optimizing compiler for quantum programs. *Quantum Science and Technology*, 5(4):044001, 2020.

[WB23] Robert Wille and Lukas Burgholzer. MQT QMAP: Efficient Quantum Circuit Mapping. 2023.

[WBZ19] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. 2019.

[Wil11] Colin P. Williams. *Explorations in quantum computing.* Texts in computer science. Springer, London and Heidelberg, 2. ed. edition, 2011.

[ZPW19] Alwin Zulehner, Alexandru Paler, and Robert Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2019.

# Part V.

# Appendix

# List of Figures

2.1. Bloch Sphere . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2. Quantum Circuit Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1. Modified Compilation Process . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

7.1. Linker Interface Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

# List of Tables

8.1. Quantum Circuit Representations . . . . . . . . . . . . . . . . . . . . . . . . . 35

13.1. Compilation Times . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

# List of Algorithms


1. Relation between **CX** and **CU** gate. . . . . . . . . . . . . . . . . . . . . . . . . 7

2. Coin Optimization - Simple . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3. Coin Optimization - Execution Path Enumeration . . . . . . . . . . . . . . . . 12
4. Coin Optimization - Control Flow Splitting . . . . . . . . . . . . . . . . . . . . 13

5. Chunk Detection - Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6. Optimization - Rotation Folding **RX** Gate . . . . . . . . . . . . . . . . . . . . 51
7. Optimization - Chunk Optimization . . . . . . . . . . . . . . . . . . . . . . . . 52

# Listings


2.1. OpenQASM 2.0 Code . . . . . . . . . . 8

3.1. Controlled Gate Optimization . . . . . . . . . . 14
3.2. **SWAP** Elimination . . . . . . . . . . 15
3.3. Unread Gate Elimination . . . . . . . . . . 16
3.4. Quantum Circuit Mapping . . . . . . . . . . 19

6.1. QuLib - Wrapper Struct . . . . . . . . . . 28

8.1. LLVM IR - Reading Function . . . . . . . . . . 36
8.2. LLVM IR - Writing Function . . . . . . . . . . 37
8.3. Chunk Detection - Simple . . . . . . . . . . 42
8.4. Chunk Detection - Run Time Condition 1 . . . . . . . . . . 43
8.5. Chunk Detection - Run Time Condition 2 . . . . . . . . . . 43
8.6. Chunk Detection - QuLib Paths Code . . . . . . . . . . 45
8.7. Chunk Detection - Variable Assignments . . . . . . . . . . 47