Optimization of Hybrid Quantum-Classical Algorithms*

Lian Remme[1], Alexander Weinert[1], and Andre Waschk[1]

German Aerospace Center (DLR), Cologne, Germany
{lian.remme,alexander.weinert,andre.waschk}@dlr.de

Abstract. Quantum computers do not run in isolation; rather, they are embedded in quantum-classical hybrid architectures. In these setups, a quantum processing unit communicates with a classical device in near-real time. To enable efficient hybrid computations, it is mandatory to optimize quantum-classical hybrid code. To the best of our knowledge, no previous work on the optimization of hybrid code nor on metrics for which to optimize such code exists.
In this work, we take a step towards optimization of hybrid programs by introducing seven optimization routines and three metrics to evaluate the effectiveness of the optimization. We implement these routines for the hybrid quantum language Quil and show that our optimizations improve programs according to our metrics. This lays the foundation for new kinds of hybrid optimizers that enable real-time collaboration between quantum and classical devices.

Keywords: optimization, quantum-classical, hybrid quantum software, Quil, optimization operations, evaluation, quantum-classical metrics, compilers

1 Introduction

In recent years, quantum computing has advanced from a theoretical possibility researched in foundational physics to the development of working quantum computers on which quantum algorithms can be executed. Mainly, quantum computers serve as stand-alone executors of *pure (quantum) programs*. For this, one first encodes the program as a quantum circuit [2–4]. A CPU then sends this circuit to a quantum processing unit (QPU), which executes it and sends the results of the execution back to the CPU [5]. This mode of execution only requires non-real-time communication capabilities between the CPU and the QPU.

Some algorithms, in contrast, require *real-time communication* [6–12] between the CPU and the QPU, i.e. communication within the coherence time of the qubits. In this execution model, the circuit to be executed is not fixed in advance. Instead, the CPU obtains intermediate computation results during the execution of the circuit and may influence subsequent execution steps. This allows to, e.g., reduce the quantum resources required by an algorithm [10–13] or

* This work is based on the master thesis of the first author [1].

to check whether a desired quantum state has been created [7–9, 14–16]. We call quantum programs that require real-time communication between the CPU and the QPU *real-time hybrid (quantum) programs*.

Modern quantum computers are significantly resource-constrained and only offer limited and costly access for researchers. Hence, it is worthwhile to optimize programs prior to execution. The optimization of pure quantum programs is an active field of research [17–20]. In contrast, to the best of our knowledge, no work exists on the optimization of hybrid quantum programs.

In this work, we take a first step towards optimization of hybrid programs by investigating the following research questions:

RQ 1 Which optimizations are applicable to hybrid quantum programs?
RQ 2 How can the effectiveness of the optimization routines be measured?

To answer these questions, we introduce seven optimization routines and three metrics to evaluate the effectiveness of the optimization. We implement these routines for the hybrid quantum language Quil [21] and show that our optimizations improve programs according to our metrics. This lays the foundation for new kinds of hybrid optimizers that enable real-time collaboration between quantum and classical devices.

This work is structured as follows: We will first give an overview of related work about quantum optimization (cf. Section 2). Then we will give background information about compiler construction, real-time quantum architecture and the programming language Quil (cf. Section 3). We will explain how we analysed Quil programs in Section 4. We will present metrics to evaluate quantum-classical algorithms (cf. Section 5) and propose optimization strategies (cf. Section 6). In the end, we will evaluate our optimization strategies in Section 7 before we discuss future work (cf. Section 8).

2 Related Work

Optimizing quantum circuits is an active field of research [17–20]. The optimization of quantum circuits involves NP-hard problems, like T-count optimization [22], a fault-tolerant implementation of topological error-correction [23], or parameter optimization with specific requirements on the circuit [24]. Research about quantum circuit optimizations usually does not consider the interaction between classical and quantum components of a larger computing system [17–20]. This also holds for Quilc [25], the compiler provided for Quil code. This compiler offers optimization routines for quantum circuits only. To the best of our knowledge, no work exists on the optimization of hybrid quantum programs, and no hardware manufacturer implements any such optimizations.

There are different metrics that quantum circuits have been optimized against. One of these metrics is the number of gates in a circuit. An automated optimization has been proposed by Nam et al. [26] that reduced the gate count. Likewise, Bae et al. [27] reduced the gate count by creating a quantum mechanical version of Karnaugh maps [28]. Puram, Karuppasamy and Thomas [29] lowered

the number of gates and the circuit depth by exploiting algebraic expressions representing the quantum circuit.

Some works focus on reducing one type of gate. For example, Gheorghiu et al. [30] proposed a method to reduce the CNOT gate count. This helps to simplify the mapping of logical to physical qubits in QPUs with limited connectivity.

Other research uses reinforcement learning in order to optimize quantum circuits. Fösel et al. [18] presented a reinforcement learning approach which showed to be able to reduce the depth and gate count on random 12-qubit circuits. Nägele and Marquardt [31] applied reinforcement learning on ZX-diagrams [32] that represent quantum circuits. Quietschlich, Burgholzer and Wille [33] used methods from classical compiler optimization to create a reinforcement learning approach. Ruiz, Laakkonen and Bausch [19] introduced AlphaTensor-Quantum, a method to optimize the T-count of a circuit. AlphaTensor-Quantum uses deep reinforcement learning. Li et al. [20] developed Quarl, a quantum circuit optimizer working with reinforcement learning. It decomposes the action space into two parts.

Another way to do quantum circuit optimization is pattern matching. This technique is about finding all matches of a small circuit (pattern) in a large circuit, while considering the commutation relations of quantum gates. A framework that exploits pattern matching is QCIR, developed by Chen et al. [34]. QCIR internally uses a pattern description format, introduced in the same work. Iten et al. [35] presented a classical algorithm which provably finds all maximal matches for a pattern matching algorithm on a quantum circuit.

Another approach is to use genetic programming to enforce quantum circuit optimization. Gemeinhardt et al. [36] proposed QeQuPI, a framework that automatically debugs and improves quantum circuits using genetic programming. Similarly, Wei et al. [37] developed an automatic circuit optimization method that is based on genetic programming.

Multiple optimizers have been developed in other research. Pointing et al. [38] created Quanto, a quantum circuit optimizer that considers circuit identities and automatically generates new ones. Similarly, the quantum circuit optimizer Quartz by Xu et al. [39] automatically generates and verifies circuit transformations for arbitrary gate sets. Paykin et al. [40] presented PCOAST, which does optimizations by exploiting commutation relations of Pauli strings. POCAST considers during its optimization procedures whether the quantum state of the qubits needs to be preserved at the end of the calculation, or if we are only interested in measurement outcomes. Hietala et al. [41] developed VOQC, a verfied optimizer for quantum circuits. It uses the Coq proof assistant [42] for verification. Coq functions are applied to transform the quantum circuit and are proofed to be correct.

There does exist research that considers the inclusion of quantum computers in classical computing systems [43–46]. These works, however, only consider pure quantum programs or the construction of a complete software stack for the use of quantum computers. Our work, in contrast, considers one particular level of such a software stack as will be required for future hybrid computing applications.

3 Background

In this section, we provide an overview about classical compiler construction in Section 3.1, describe the architecture for real-time quantum computation in Section 3.2 and introduce the Quil language in Section 3.3.

3.1 Compiler Construction

Modern compilers typically go far beyond simply translating a program from a source language into a target language. Instead, they optimize for the target language as well as the target environment. Typically, they perform multiple machine-independent as well as machine-dependent passes. Designing novel optimization passes and choosing the optimal order of passes for a given program remains an active field of research [47–51].

Each compiler pass either gathers information about the program, or it changes its structure without changing its semantics. We call the former and latter passes *analyses* and *transformations*, respectively. Typically, transformations use the information extracted by analyses.

Table 1: Typical analyses and transformations [52].

Analyses	Transformations
Available expressions	Constant folding
Constant propagation	Copy propagation
Live-variable analysis	Dead code elimination
Reaching definitions	

There exist a plethora of complex analyses and transformations. In this work, we adapt a number of classical optimization passes to the setting of hybrid programs. We present these operations in Table 1 and provide a brief overview below. For a more detailed discussion, we refer the reader to the work of Aho, Lam, Sethi, and Ullman [52].

Available expressions is about checking whether the result of evaluating an expression is available at a program point p. An expression is available if it is evaluated on every possible path between the start of the program and p. Additionally, there must not be new assignments to a or b between the last evaluation and p.

Constant-propagation means to check whether a variable holds a unique constant value at a program point p.

Live-variable analysis is used to determine which variables are still "in use" (alive) at a point p of the program. If there is any usage of a variable's value between p and the program halt, it is alive. Otherwise, it is dead. Values of dead variables have no influence on the remaining program, and they do not need to be considered any further.

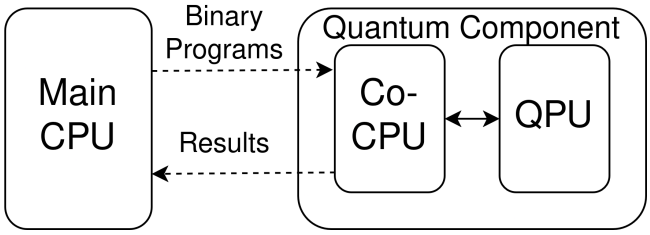

Fig. 1: The refined HQCC model (from [11, Fig. 2]). Dotted lines indicate slow communication (longer than the coherence time of the qubits), the continuous line indicates fast communication (shorter than qubit coherence time).

Reaching definitions is used to determine which definitions of a variable can reach a program point p.

Constant folding is evaluating constant expressions and replacing the expressions by their values.

Copy propagation checks when a copy statement (e.g. `a = b`) is given, whether a can be removed and b be used instead of a.

Dead code elimination removes "dead" (or useless) code from the program. This affects unreachable code as well as code that computes values which are never used.

3.2 Real-Time Quantum Architecture

Real-time communication refers to the exchange of information between the CPU and the QPU during the qubit coherence time. This communication is crucial for tasks such as applying gates to qubits based on prior measurements or adjusting parameters of quantum gates in response to previous measurement outcomes. It is mandatory for some algorithms and can be used to reduce quantum resources of a calculation or to check whether a desired quantum state has been created. Examples for real-time algorithms are quantum teleportation [6], active reset [7], magic state distillation [14–16], repeat-until-success [8,9], iterative phase estimation [10,11], and Shor's algorithm for $2n + 1$ qubits [12,13].

To execute real-time quantum computation, a refined heterogeneous quantum-classical computation (HQCC) architecture is needed [11] (see Figure 1). We will assume this architecture in this work. A co-CPU and the QPU form a quantum component, which gets a quantum program from the main CPU and returns the results to said main CPU. This component can apply real-time feedback on qubits and execute classical instructions. The classical calculations are limited by the coherence time of the qubits, but not by the abilities of the co-CPU.

The HQCC architecture provides an interesting hardware case, as we work with two calculation components: The co-CPU and the QPU. Both devices work in parallel, and only influence each other on defined points (e.g. measurements).

In the following, we will refer to the co-CPU as CPU, unless stated otherwise.

3.3 The Quil Programming Language

Quil [21] is an assembly-style low-level language developed by Rigetti to specify quantum-classical computations on an abstract machine architecture. In this subsection, we give an overview of the instructions that Quil provides, to the extent that it is relevant in this work. For a more extensive overview, we refer the reader to the paper introducing Quil [21].

The Quil ecosystem consists of the language specification as well as some additional tools: PyQuil [53], a Python library to generate and execute Quil code. Quilc [25], a compiler that compiles Quil circuits to circuits a defined hardware can execute. And the Quil-Lang quantum virtual machine (QVM) [54], that can simulate the execution of Quil code.

In Quil, integer indices are used to refer to qubits. Quil supports four classical variable types: `BIT`, `OCTET`, `INTEGER`, and `REAL`. Classical variables need to be declared before usage, e.g. `DECLARE a BIT`. Qubits do not need to be declared upfront.

Quil offers different categories of instructions, namely

- **quantum gates**: Instructions that name a gate and at least one qubit that is applied to the gate, e.g. `H 0`, or `CNOT 0 1`.
- **parameterized quantum gates**: Quantum gate instructions with gates that receive a classical parameter, e.g. `RZ(angle) 0`, with the classical parameter `angle`. Quantum gates that receive a fixed value (e.g. `RZ(1.57)`) do not count as parameterized in the context of this work.
- **classical instructions**: Calculations applied on classical variables. The classical instruction set is turing-complete. It is comparable to the instruction set of the Assembly language [55]: For example, `MOVE` writes the value of the second variable into the first, `ADD` adds the value of the second variable onto the value of the first, and `NEG` negates the given variable [56, 6.5].
- **measurements**: Instructions that measure the value of a qubit. The value can, but does not have to be, saved in a classical parameter. E.g.: `MEASURE 0 ro` or `MEASURE 1`.
- **control structures**: One can define labels (`LABEL @label_name`) and jumps. Jumps can be conditional (e.g. `JUMP-WHEN @label_name cond`) or unconditional (e.g. `JUMP @label_name`).

Quantum gates that are not parameterized can be exclusively executed by a QPU and are therefore *quantum instructions*. Classical instructions can be exclusively executed by a CPU and are therefore *classical instructions*. All other instructions, including control structures, need to be executed by the QPU and the CPU at the same time, and are therefore *hybrid instructions*.

We explore the optimization Quil code with respect to heterogeneous architecture.

4 Analysis of Quil Programs

In order to optimize Quil programs, we first have to analyze their structure. We will optimizie the analyzed Quil programs in Section 7.

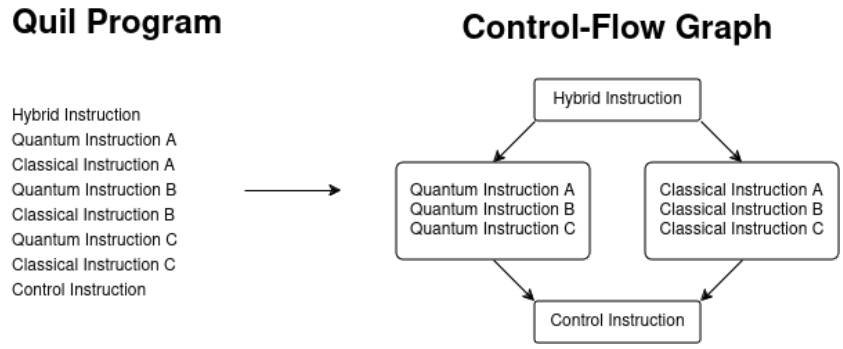

Fig. 2: Creating basic blocks in the CFG from alternating quantum and classical instructions.

To the best of our knowledge, no benchmark database for the real-time quantum-classical use case exists. As most works focus on quantum circuits, there are relatively few algorithms with real-time feedback properties. We chose to work with four algorithms that are mentioned frequently in literature. Creating an extended benchmark database is out of scope for our work and is a topic for future work.

The four real-time quantum classical algorithms used in this paper are:

- Quantum teleportation [6]
- Magic state distillation [14–16]
- Repeat-until-success [8, 9]
- Iterative phase estimation (IPE) [10, 11]

The code for the algorithms is given in [57], alongside the results of the analysations described in this section.

In this section, we will describe how to do control flow analysis and data-dependence analysis on Quil.

4.1 Control Flow Analysis

A way of analysing a program for different compilation steps is to do a control flow analysis [58]. This can be done using a control flow graph (CFG). A CFG represents all possible execution paths of a program. The directed edges of the graphs represent jumps. The nodes describe a basic block, which are "sequences of statements that are always executed one-after-the-other, with no branching" [52, chapter 2.8.1]. This means that a jump on the current instruction as well as a label (jump target) at the next instruction end a basis block [52, chapter 8.4.1].

We created a CFG [58] for all programs we evaluate. To show which device executes which instructions, we divide the instructions depending on their type.

Each node/basic block of the CFG consists of either only quantum instructions, only classical instructions, or only hybrid/control structures. Quantum and classical instructions that are executed without a hybrid/control instruction in between are written into two parallel basic blocks. An example of this is shown in Figure 2. By creating a CFG in this way, one can easily see which parts of the code are executed by which device(s).

4.2 Data-Dependence Analysis

Most optimization techniques depend on data-flow analysis [52, chapter 9.2]. One way to do data-flow analysis is to create a data-dependence graph (DDG)[1] [59, chapter 5.3.2] of the program. A DDG is a directed graph. Nodes represent basic blocks (or, in our case, single instructions) of a program. An instruction A is a successor of instruction B in the graph, if A has to be executed after B in order for the program to be correct, e.g. if A and B access the same variable. Edges only appear between instructions/nodes if no other instruction C has to be executed between A and B.

We create a DDG using information from the Quil program and the CFG. Every node of the DDG holds a single Quil instruction. The edges indicate the order in which the instructions have to be executed. This is done by checking variable dependency. Unconditional jumps are resolved before creating the DDG and not listed as nodes.

Conditional jumps in the program pose a problem to the DDG. If a conditional jump targets an already executed line, it introduces circular dependencies. This could only be resolved exactly if we knew the number of iterations, which would be analogous to solving the Halting problem, and therefore not generally possible.

We resolve this issue by creating a DDG only up to the next conditional jump. By that, we can receive multiple DDGs for a single program, all depicting a part of the program. One of the DDGs is the start DDG, which is the DDG describing the entry of the program. Additionally, we have one or multiple halt DDGs, which include the program instructions last executed before the program terminates. All DDGs beside the start DDG start at the jump target of a conditional jump. If an instruction is the target of multiple conditional jumps, we create multiple DDGs. An example for a program that results in multiple DDGs is given in Listing 1, and the corresponding DDGs in Figure 3.

For the IPE, we resolved conditional jumps before DDG creation. In the executable IPE, we use the code given in Listing 2 to add 1 to `param_no_pi[0]` if `lastMeasurement[0]` is 1.

Conditional jumps have to be used, as Quil's semantic does not allow adding a bit-value to a real value. In the IPE that is used for the DDG and during the optimization, the above code is changed to the one in Listing 3: The bit value is directly added to the real value.

[1] Also called program-dependence graph [52, chapter 11.8.2].

```
DECLARE m BIT
H 0
MEASURE 0 m
JUMP-WHEN @label m
Y 0
LABEL @label
Z 0
MEASURE 0 m
```

Listing 1: An example for Quil code that results into multiple DDGs. The corresponding DDGs can be found in Figure 3.

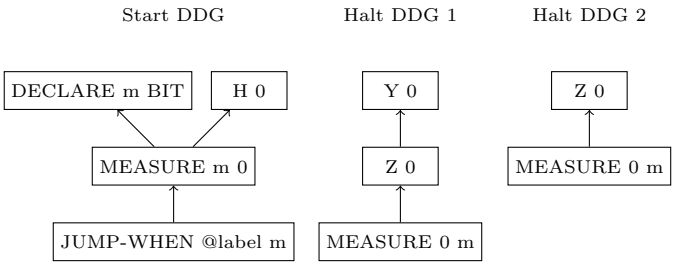

Fig. 3: An example for multiple DDGs originating from one Quil code. The code can be found in Listing 1.

```
DECLARE param_no_pi REAL[1]
DECLARE lastMeasurement BIT[1]

JUMP-UNLESS @noadd1 lastMeasurement[0]
ADD param_no_pi[0] 1
LABEL @noadd1
```

Listing 2: Executable version of adding 1 to `param_no_pi[0]` if `lastMeasurement[0]` is true. Quil's semantic does not allow to add a `BIT` variable to a `REAL` variable. Therefore, we need conditional jumps.

```
1 DECLARE param_no_pi REAL[1]
2 DECLARE lastMeasurement BIT[1]
3
4 ADD param_no_pi[0] lastMeasurement[0]
```

Listing 3: Version we use for analysation of adding 1 to `param_no_pi[0]` if `lastMeasurement[0]` is true. The `BIT` value is directly added to `param_no_pi[0]` to avoid conditional jumps.

While Quil's semantic forbids this construction, it has the same logical effect: Adding 1 to `param_no_pi[0]` if `lastMeasurement[0]` is 1. This is done to prevent conditional jumps in the IPE algorithm, which makes the optimization and its evaluation simpler. This has only been done for IPE, not for the other algorithms. The CFGs and DDGs for all algorithms can be found in our GitHub repository alongside our code [57].

5 Metrics for Quil Programs

How to best evaluate quantum-classical algorithms is an open question. While there are some metrics for the evaluation of quantum circuits available (cf. Section 2), extending them to the quantum-classical case is not straightforward. Two widely used metrics, the numbers of gates and the circuit depth, do not take into account classical calculation nor interaction between classical and quantum calculations.

In this work, we assume a relatively simple execution model, which has an execution time of 1 per instruction and no communication latencies. This model does not accurately depict real-world quantum computers, but suffices for our proof of concept. A more accurate description is dependent on the hardware and out of scope for this work. We plan to access this question in future work.

We propose three metrics to statically evaluate quantum-classical Quil programs: Wall time, quantum instruction number (QIN), and quantum calculation time (QCT). The metrics can be used to optimize Quil programs against or to evaluate how well optimization methods worked. Our GitHub repository [57] provides the functionality to evaluate Quil programs with respect to our metrics. We evaluate our optimization methods against our metrics. The results are given in Section 6.

5.1 Wall Time

The wall time of a program is the total time from the start to the end of a program. To calculate the wall time of a hybrid quantum-classical algorithm, we need to consider that the CPU and the QPU work in parallel.

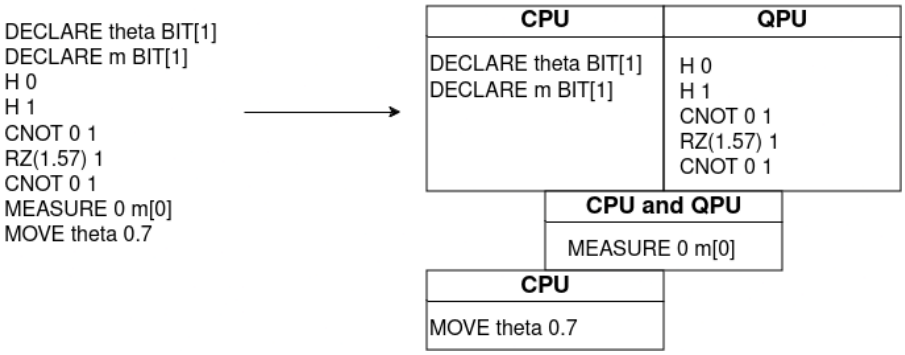

Fig. 4: An example of a Quil execution that has a wall time of 7.

We asses how long the program would take to execute while considering that all instructions have an execution time of 1. Whenever possible, the CPU and the QPU execute instructions in parallel. Two parallel executed instructions receive an execution time of 1 together. At a hybrid or control instruction, one device waits for the other. For example, the program in Figure 4 has a wall time of 7.

The wall time is calculated for all DDGs of a program and summed up for comparison of syntactically different representations of the same program. This means that if two identical DDGs exist due to instructions being targets of multiple jumps (cf. Section 2), that DDG is counted twice.

5.2 Quantum Instruction Number

Quantum instructions can introduce error into a calculation, as quantum gates of noisy intermediate-scale quantum (NISQ) devices do not have perfect fidelity. Therefore, minimizing the QIN is sensible.

The number of quantum instructions per DDG are counted and summed up for comparison to other codes. Hybrid instructions are included in the counting, as the QPU and the quantum state are considered during hybrid calculations.

5.3 Quantum Calculation Time

As quantum states in a QPU have a limited coherence time, it is sensible to minimize the time the QPU has to keep the qubits coherent, i.e. the time during which the QPU calculates. This value is the QCT. As we assume that each instruction needs a time value of 1, the time is given in units of the instruction number.

For the calculation of the QCT, we assume the best case: The QPU starts with the first quantum instruction on the latest possible time such that the CPU does not have to wait for the QPU at the first hybrid instruction. Further, the QPU calculates the quantum instructions after the last hybrid instruction directly after the hybrid instruction. Additionally, the QPU has to keep the

quantum state stable during the wall time between the first and last hybrid instruction.

It should be noted that a reasonable program has no quantum instructions after the last hybrid instruction. This would only create quantum information which is destroyed again, as it would have to be measured for further usage. Measurement, again, would be a hybrid instruction.

This results in a QCT τ_Q of

$$\tau_Q = \delta t_{\text{between}} + n_{\text{q\_before}} + n_{\text{q\_after}} \tag{1}$$

with the wall time between first and last hybrid instruction (including these hybrid instructions) $\delta t_{\text{between}}$, the number of quantum instructions before the first hybrid instruction $n_{\text{q\_before}}$, and the number of hybrid instruction after the last hybrid instruction $n_{\text{q\_after}}$. An example can be seen in Figure 5.

$n_{\text{q\_before}} = 2$	DECLARE r REAL	Classical
	DECLARE m BIT	Classical
	H 0	Quantum
	Y 0	Quantum
$\delta t_{\text{between}} = 6$	MEASURE 0 m	Hybrid (first)
	H 0	Quantum
	Z 0	Quantum
	MOVE r 2	Classical
	RZ(r) 0	Hybrid
	H 0	Classical
	MEASURE 0 m	Hybrid (last)
$n_{\text{q\_after}} = 1$	Z 0	Quantum

Fig. 5: An example of determining $\delta t_{between}$, n_{q_before} and n_{q_after} from Quil code. The QCT is the sum of these three values.

A difficulty of this metric arises when the program is divided into more than one DDG. The first hybrid instruction in the start DDG is the globally first hybrid instruction. The last hybrid instruction in the halt DDG is the globally last hybrid instruction. If there are multiple possible last DDGs, we take the one that causes the longest QCT (worst-case). The QCT is calculated by the number of quantum instructions before the first hybrid instruction and after the last hybrid instruction. Additionally, the wall time of all other DDGs and between the first and the last hybrid instruction (included) are added.

If the last DDG has no quantum instruction, it does not add to the QCT. The wall time of all other DDGs is simply added, as every DDG necessarily ends at a hybrid conditional jump.

Note that, in an ideally optimized program, a DDG with only classical instructions should not exist, as this could be calculated completely by a main CPU and does not have to be sent to a CPU-QPU complex.

Table 2: Instruction Number (instr. no.), wall time, QIN, and QCT of the original Quil files of the different algorithms. The instruction number and wall time are calculated per DDG.

Example program	Instr. no.	Wall time	QIN	QCT
Quantum teleportation	8, 2, 2	6, 2, 1	9	10
Magic state distillation	67, 63, 6	62, 62, 6	66	130
Repeat-until-success	12, 11, 10, 7	9, 10, 9, 6	34	35
IPE	55	45	25	33

Table 2 shows the values of the different metrics for our initial implementations of the algorithms. We will look at strategies to improve (i.e., reduce) the values in the next section.

6 Optimization Strategies

In this section, we aim to create a set of optimization methods for heterogeneous quantum-classical architectures. The optimizations are done statically by a main CPU, before the program is send to the CPU-QPU component. Our GitHub repository [57] provides the functionality to apply these optimization methods to a Quil code.

In the beginning, we will derive some optimization strategies from already well established classical (machine-independent) strategies. Afterwards, we will design strategies specifically targeting the quantum-classical nature of real-time code.

6.1 Strategies Adapted from the Classical Case

We will examine if and how the classical machine-independent optimizations mentioned in Section 3.1 can be applied to the quantum-classical heterogeneous case. Some optimization methods are not sensible for the quantum case, but could in principle be implemented for the classical part of the algorithms. We did not implement these optimizations, to keep our focus on quantum-classical hybrid code.

It is insensible to try to determine **available expressions** in Quil. Instructions that calculate an expression $a \bullet b$ and assign the result to another variable do not exist. Instead, quantum operators act directly on qubits. Additionally, the results of classical calculations are stored in one of the involved variables. For example,

```
ADD a b
```
(2)

is interpreted as

$$a \coloneqq a + b. \tag{3}$$

```
1 DECLARE a INTEGER
2 DECLARE b INTEGER
3 MOVE a 3
4 ADD a 10
5 MOVE b 7
6 MOVE a 10
```

Listing 4: An example of Quil code with live and dead variables. Assuming `a` is a readout variable, it counts as dead in line 4, as the addition result is not read anywhere before a is written to in line 6. `b` is dead in line 5, as it is not read afterward.

We apply **constant propagation** to the classical and the quantum part of the code.

For the classical part of the code, the algorithm works as follows: A classical value is recognized as constant by a `MOVE` instruction if the instruction moves a constant value onto it. E.g

```
MOVE a 10
```
(4)

results into `a` having the constant value 10.

A variable remains constant until it is being written to again.

```
ADD b a
```
(5)

writes and reads `b`, but only reads `a`, thus `a` remains constant. This means if `b` was constant before the instruction the algorithms recognizes `b` to be constant at the reading part of the instruction, but `b` looses its constant state afterward. This would even hold if `a` was replaced by a constant value. Constant folding will replace the instruction by a `MOVE` instruction if `a` and `b` were constant at read-time.

For the quantum part of the code, we restrict ourselves to the Pauli-basis and single qubit gates. This was done to avoid an exponential scaling of the constant propagation algorithm while trying to keep track of arbitrary quantum states.

A qubit is constant with the value `Pauli X Zero` at the start of the program and after a `RESET`. The constant propagation checks at which instructions a constant qubit "arrives" with a specific value, either because it is initially used or has been reset before, or because constant folding has shown that it remained in a constant Pauli basis after a gate application.

Live-variable analysis can be used for the classical and quantum parts of heterogeneous programs.

A classical variable counts as dead if it is not read until the variable's value is overwritten or the program terminates. For the application of this algorithm, we need to know which classical variables hold *readout values*, i.e. which values

need to be reported back to the main CPU at the end of the quantum-classical program. These variables are by default alive at the end of the program, which is handled as if there was an additional read of the variable after the halt of the program. An example is given in Listing 4.

A qubit counts as dead if its information content does not influence any classical information from the current instruction until the end of the program.

The qubit can safely be called dead at a program point p if it is

- not used by any multi-qubit gate, **and**
- not measured with the result saved in a classical variable

until

- the end of the program, **or**
- the next reset.

The variable may be entangled with other variables. Due to the no-communication theorem [60], operations on a qubit A cannot influence the measurement result on other qubits without using a measurement result of A. This even holds if A and the other qubits are entangled. Therefore, we do not have to check for entanglement if a qubit will not be used by any multi-qubit gates in succeeding instructions and if the result of a measurement on it will not be considered in the program.

We can only apply live-variable analysis at the halt DDG of the program, as we cannot say whether a variable will be used in future DDGs. This limits the usage of the live-variable analysis. Live-variable analysis could be applied to non-halt DDGs in a restricted manner: All values count as alive at the end of the DDG (handled as if a read-instruction is about to follow). If a variable is dead because its value is overwritten within one DDG, it can still safely be called dead. However, our current implementation [57] does not support this.

Reaching definitions can be implemented for the classical Quil variables by checking `DECLARE` and `MOVE` statements. Quil programs do not offer define/declare statements for qubits, neither direct assignment of a value to a qubit. Therefore, applying reaching definitions to the quantum part of the program is not sensible. As we focused on quantum-classical routines, we did not implement reaching definitions for the classical part of the programs either.

Constant folding can be applied on classical and quantum parts of our hybrid code. In classical instructions, all constant variables that are read are replaced by their constant values. This means, e.g.,

```
ADD a b
```
(6)

is replaced by

```
ADD a 10
```
(7)

if `b` has a constant value of 10. The same holds for parameterized quantum gates. If the classical parameter is constant, it is replaced by the constant value.

If a value that is written to is constant and additionally, all read values in the instructions are constant, the instruction is replaced by a `MOVE` instruction. As an example: If `a` is 5, `b` is 7 and the instruction is

```
ADD a b,
```
(8)

it is replaced by

```
MOVE a 12
```
(9)

as `a` would be 12 after the addition.

For the quantum part of the program, constant folding calculates the result of the application of a gate on a qubit. This is only calculated if the qubit is constant beforehand (i.e. in a Pauli basis, as defined by constant propagation) and a single-qubit Clifford gate acts on it. We calculate the next Pauli state and still assume the qubit as constant (i.e. in the respective Pauli state).

Constant folding for qubits can be done efficiently for Clifford gates due to the Gottesman-Knill theorem [61]. To avoid dealing with entangled states during constant propagation, we generally assume qubits to be non-constant after a multi-qubit gate.

Due to this, we only trace the constant value of a qubit until the first non single-qubit or non-Clifford gate acts on it. This means we are tracing the six Pauli basis states as values of the qubits.

Measurements are in general considered to be random and write non-constant values in the classical values. The only exception is if we know that a qubit is either in the $|0\rangle$ or $|1\rangle$ state. In that case, the classical value receives the constant value 0 or 1.

Copy propagation can be used in the classical part of heterogeneous code. It is not sensible for quantum code due to the no-cloning theorem [62, 63].

Dead code elimination can be applied to classical and quantum instructions. We use the results of the live-variable analysis about which variables are dead. Classical code is dead if all variables written to in the instruction are dead. Quantum code is dead if all qubits an instruction acts on are dead. This also holds for parametrized quantum gates. All dead instructions are deleted.

These were optimization methods taken from the compilation routine of purely classical code. We end up with *constant propagation*, *live-variable analysis*, *constant folding*, and *dead code elimination* for the quantum-classical case.

6.2 Strategies Designed for Quantum-Classical Calculations

Besides the classical methods, we can apply algorithms that are specific for the heterogeneous quantum-classical architecture. We introduce three algorithms targeting quantum-classical architecture in this work: An analysis operation that finds hybrid-dependencies, and the two transformation operations instruction reordering and latest possible quantum execution.

Finding hybrid-dependencies checks which instructions have to be executed before a hybrid node becomes executable. This analysis first finds all hybrid

```
1 DECLARE m INTEGER[1]
2 H 0
3 MEASURE 0 m
4 RZ(m) 0
```

Listing 5: An example of Quil code with hybrid instructions that depend on previous lines.

instructions, and afterwards determines which instructions have to be executed before the respective hybrid instruction by using the DDG. A hybrid instruction can have other hybrid instructions as dependencies. A hybrid instruction "blocks" the instruction dependencies for all other hybrid instructions.

For example, Listing 5 contains the hybrid instructions `MEASURE` and `RZ(m)`. All other lines have to be executed before `RZ(m)`. But the only direct dependency we save for it is `MEASURE`, as `MEASURE` is a hybrid instruction and depends on `H` and the `DECLARE` instruction as well. For `MEASURE`, the dependencies `H` and `DECLARE` are saved.

Instruction reordering can be done as long as instructions dependent on each other are still in the same order. The DDG provides information about the dependent instructions.

The goal of this routine is to reorder the instructions to maximize the number of parallel executions of the CPU and QPU. In other words, the time in which only one device is calculating and the other one is idling should be minimized.

The two devices only influence each other through hybrid nodes. To prevent long waiting times of one device for the other, we use the knowledge we gained from the DDG and the finding hybrid-dependency analysis.

Listing 6 shows the pseudocode of the instruction reordering algorithm. If the CPU or the QPU have to wait for the other device before a hybrid instruction, the algorithm aims to avoid that the waiting device does not execute instructions. It is checked whether there are instructions for the waiting device that can already be executed, i.e. for which all dependencies have been executed. If this is the case, the waiting device can execute the instructions while waiting instead of idling.

Latest possible quantum execution is one method to keep the total execution time of the quantum device small.

The goal of this algorithm is to execute as many classical instructions as possible before the QPU starts to work. For this, the execution of instructions is reordered in the following way:

- All classical instructions that are not dependent on a quantum instruction are executed at the start.
- The first quantum instruction is executed such that the QPU and the CPU reach the first hybrid node simultaneously. This is again done with the assumption that each instruction has an execution time of 1.

```
Input:
instruction_ddg: The instructions of the programm with the dependencies of the instructions.

Output:
execution_queue: The (probably new) order in which the instructions should be executed. Must keep the topological order of instruction_ddg.

execution_queue ← empty list
relevant_instructions_list ← hybrid instructions and last instruction of instruction_ddg
next relevant_instruction to be executed:
        dependencies ← instructions that still need to be executed before relevant_instruction in instruction_ddg
        quantum_number ← amount of quantum instructions in dependencies
        classical_number ← amount of classical instructions in dependencies
        Add dependencies to execution_queue
        while new instructions in instruction_ddg are excutable wrt current execution queue and quantum_number != classical_number:
                if quantum_number > classical_number:
                        Add executable classical instructions to execution queue
                else:
                        Add executable quantum instructions to execution queue
                Update quantum_number and classical_number according to the number of added instructions
        add relevant_instruction to execution_queue
        repeat
```

Listing 6: Instruction reordering algorithm.

After setting up this set of optimization operations, we will check in the next section how these operations could optimize example algorithms.

7 Evaluation

In this section, we will examine how the optimization operations introduced in Section 6 affect the quantum algorithms mentioned in Section 4. Some of the algorithms from Section 4 are not sensible for the quantum case, but can in principle be applied to the classical part of a quantum-classical algorithm. To keep our focus on the hybrid code, we only implemented the optimization routines that can be applied on quantum and classical parts of the code. This results in the implementation of *constant propagation*, *live-variable analysis*, *constant folding*, *dead code eliminiation*, *finding hybrid-dependencies*, *instruction reordering* and *latest possible quantum reordering*. We implemented these optimization strategies, the codebase can be accessed at our repository [57].

Determining the best order to apply optimization operations on a given code (phase-ordering) is an open question in classical compiler architecture [50, 51]. Finding a sensible order is out of scope for this work, and we will simply draw operations to apply to the codes at random. We do this 500 times for each algorithm and apply 50 optimization operations every time. Although the sequence of operations was random, specific analysis and transformation routines had to follow one another in a prescribed order, as certain analysis operations provide the necessary background information for subsequent transformation operations. The optimization routine consisted of a permutation of the following operations succeeding each other:

- Constant propagation → constant folding.
- Live-variable analysis → dead code elimination.
- Finding hybrid-dependencies → instruction reordering.
- Finding hybrid-dependencies → latest possible quantum reordering.

Which means we did 25 random draws per optimization routine.

Table 3: Best (i.e. minimal) wall time, instruction number, and QCT that could be reached by random application of 50 optimizations. No given number indicates that no improvement could be done compared to the original program. The QIN has never been improved. The original values can be found in Table 2.

Example program	Wall time	Instr. no.	QCT
Quantum teleportation	–	–	–
Magic state distillation	53, 53, 6	–	112
Repeat-until-success	–	–	–
IPE	35	51	30

After all optimization operations had been applied, the metrics of Section 5 were calculated for the optimized Quil program. The best resulting values for the metrics to evaluate Quil programs against can be seen in Table 3.

No improvements were observed in comparison to the original programs for quantum teleportation and repeat-until-success protocols. The IPE's wall time, instruction number, and QCT could be reduced by 22% (wall time), 7.3% (instruction number), and 9.1% (QCT).

The magic state distillation's wall time (summed over all DDGs) could be reduced by 14%, its QCT dropped by 14% as well. It yields the same values for all metrics in every optimization iteration.

Table 4: The results of optimizing the iterative phase estimation with respect to the metrics we evaluate the code against. 500 runs have been done. The frequencies of the different combinations of wall time, instruction number, QIN and QCT are given. The bold results are the best ones for this metric (only given if the metric does not have the same result for all sets).

Wall time	Instr. no.	QIN	QCT	Result frequency
35	**51**	25	**30**	49.2%
36	**51**	25	**30**	43.0%
36	**51**	25	31	6.6%
37	52	25	**30**	0.6%
37	53	25	31	0.2%
39	**51**	25	33	0.2%
39	55	25	32	0.2%

We received varying results for the application of optimization routines on the IPE algorithm, as outlined in Table 4. The most frequent results are a wall time of 35, an instruction number of 51, and a QCT of 30. This is not only the most frequent result set, but also the result set with most optimal values for all metrics. The second most frequent result has a value of 36 for the wall time.

8 Discussion and Future Work

In this work, we demonstrated that it is possible to optimize quantum-classical code based on the metrics we introduced. While the programs we applied this approach to showed modest improvements, the results highlight the potential for further enhancement in future applications. As our results indicate, no improvements were observed for quantum teleportation or repeat-until-success. This is likely due to the relatively low number of instructions within these applications, which may limit the potential for optimization.

Nevertheless, we expect the optimization of heterogeneous quantum-classical programs to become more relevant in the future. The development of more ab-

stract quantum programming languages will lead to more compilation steps between the written program and the hardware, which will also lead to a bigger need of optimization routines. Therefore, we emphasize that the optimization of quantum-classical code is something that should be investigated in more detail.

For evaluation purposes, we had to generate multiple DDGs of a Quil program, eliminating circular dependencies and loops to apply our optimization metrics (cf. Section 7). In contrast, loops are a significant portion of classical programs, making loop optimization a key factor in improving overall program performance [52].

It would be valuable to explore how loop optimizations and branch prediction techniques can be applied to quantum-classical programs. One straightforward approach could be to eliminate a conditional jump if static analysis confirms that the result is certain.

For our evaluations we assumed a rather simple model of a quantum computer. Every instruction had an execution time of 1 and communication latencies between QPU and CPU were omitted. In reality, CPUs execute an instruction much faster than QPUs, and the execution time for one instruction in a QPU varies greatly with the qubit technology used. Additionally, the execution time of one-qubit and multiple-qubit gates often differs [64–67]. An additional consideration is the fact that quantum gates can be executed in parallel if they operate on distinct qubits. Succeeding work could examine in more detail how this affects quantum-classical optimization.

In Section 2 we mentioned that there is research on circuit optimization focused solemnly on the quantum circuit. Combining these quantum circuit optimizations with the optimizations we derived from classical procedures is another non-trivial problem we strive to investigate in the future.

9 Conclusion

The aim of this work was to explore optimization strategies for heterogeneous calculations, where a quantum processing unit interacts with a classical device in near-real time. While much research has focused on optimizing quantum circuits, little attention has been given to the classical component in real-time quantum-classical calculations. Additionally, there has been a lack of established metrics for evaluating the performance of quantum-classical hybrid programs.

In this context, our work makes a significant contribution by introducing three performance metrics—wall time, quantum instruction number, and quantum calculation time—to assess the efficiency of hybrid programs. Furthermore, we proposed seven optimization operations for heterogeneous Quil programs, including constant propagation, live-variable analysis, constant folding, dead code elimination, finding hybrid dependencies, instruction reordering, and latest possible quantum execution.

Through practical evaluation, we demonstrated that these operations can indeed optimize programs based on our proposed metrics. This work paves the way for the development of new hybrid optimization techniques, enhancing the

potential for real-time collaboration between quantum and classical devices. By providing both the foundational metrics and optimization operations, we believe this research opens up exciting avenues for future advancements in hybrid quantum computing.

Acknowledgment

Generative AI (GitHub Copilot) was used while working on the code corresponding to this work [57]. It was used as an integrated tool in the integrated development environments (IDEs). It has been used to assist during code-writing, helped writing documentation strings, deciding on variable/function names, wrote first drafts of some methods/functions, for formulation suggestions, and for Markdown formatting.

References

1. L. Remme, "Optimization strategies for quantum computers in distributed systems," Master's thesis, Heinrich Heine University Düsseldorf, 2 2025.
2. IBMQuantum. (2024) circuit — IBM Quantum Computing. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/circuit
3. Xanadu. (2025) Quantum circuits – pennylane 0.40.0 documentation. [Online]. Available: https://docs.pennylane.ai/en/stable/introduction/circuits.html
4. G. Q. AI. (2025) Circuits — Cirq — Google Quantum AI. [Online]. Available: https://quantumai.google/cirq/build/circuits
5. E. Knill, "Conventions for Quantum Pseudocode," Los Alamos National Lab. (LANL), Los Alamos, NM (United States), Tech. Rep., 06 1996. [Online]. Available: https://www.osti.gov/biblio/366453
6. C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels," *Physical Review Letters*, vol. 70, pp. 1895–1899, 1993.
7. T. Lubinski, C. Granade, A. Anderson, A. Geller, M. Roetteler, A. Petrenko, and B. Heim, "Advancing hybrid quantum–classical computation with real-time execution," *Frontiers in Physics*, vol. 10, 2022.
8. A. Paetznick and K. M. Svore, "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries," *Quantum Information & Computation*, vol. 14, no. 15–16, pp. 1277–1301, Nov. 2014.
9. P. Fu, K. Kishida, N. J. Ross, and P. Selinger, "Proto-Quipper with Dynamic Lifting," *Proceedings of the ACM on Programming Languages*, vol. 7, no. 11, pp. 309–334, Jan. 2023.
10. M. Dobšíček, G. Johansson, V. Shumeiko, and G. Wendin, "Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark," *Physical Review A*, vol. 76, 9 2007.
11. X. Fu, J. Yu, X. Su, H. Jiang, H. Wu, F. Cheng, X. Deng, J. Zhang, L. Jin, Y. Yang *et al.*, "Quingo: A Programming Framework for Heterogeneous Quantum-Classical Computing with NISQ Features," *ACM Transactions on Quantum Computing*, vol. 2, 2021.

12. M. Rossi, L. Asproni, D. Caputo, S. Rossi, A. Cusinato, R. Marini, A. Agosti, and M. Magagnini, "Using Shor's algorithm on near term Quantum computers: a reduced version," *Quantum Machine Intelligence*, vol. 4, no. 18, Jul. 2022.
13. S. Beauregard, "Circuit for Shor's algorithm using 2n+ 3 qubits," *Quantum Information & Computation*, vol. 3, no. 2, p. 175–185, Mar. 2003.
14. E. Knill, "Fault-Tolerant Postselected Quantum Computation: Schemes," *arXiv preprint quant-ph/0402171*, 2004.
15. S. Bravyi and A. Kitaev, "Universal quantum computation with ideal Clifford gates and noisy ancillas," *Physical Review A*, vol. 71, 2005.
16. A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, 9 2012.
17. K. Karuppasamy, V. Puram, S. Johnson, and J. P. Thomas, "Quantum Circuit Optimization: Current trends and future direction," *arXiv e-prints*, p. arXiv:2408.08941, Aug. 2024.
18. T. Fösel, M. Yuezhen Niu, F. Marquardt, and L. Li, "Quantum circuit optimization with deep reinforcement learning," *arXiv e-prints*, p. arXiv:2103.07585, Mar. 2021.
19. F. J. Ruiz, T. Laakkonen, J. Bausch, M. Balog, M. Barekatain, F. J. Heras, A. Novikov, N. Fitzpatrick, B. Romera-Paredes, J. van de Wetering *et al.*, "Quantum Circuit Optimization with AlphaTensor," *arXiv e-prints*, pp. arXiv–2402, 2024.
20. Z. Li, J. Peng, Y. Mei, S. Lin, Y. Wu, O. Padon, and Z. Jia, "Quarl: A Larning-Based Quantum Circuit Optimizer," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649831
21. R. S. Smith, M. J. Curtis, and W. J. Zeng, "A Practical Quantum Instruction Set Architecture," *arXiv e-prints*, p. arXiv:1608.03355, Aug. 2016.
22. J. van de Wetering and M. Amy, "Optimising T-count is np-hard," *arXiv e-prints*, Sep. 2023.
23. D. Herr, F. Nori, and S. J. Devitt, "Optimization of lattice surgery is NP-hard," *npj Quantum Information*, vol. 3, no. 35, 2017.
24. J. van de Wetering, R. Yeung, T. Laakkonen, and A. Kissinger, "Optimal compilation of parametrised quantum circuits," *arXiv e-prints*, Jan. 2024.
25. 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*, vol. 5, no. 4, 7 2020.
26. Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *npj Quantum Information*, vol. 4, no. 1, p. 12, 2018.
27. J.-H. Bae, P. M. Alsing, D. Ahn, and W. A. Miller, "Quantum circuit optimization using quantum karnaugh map," *Scientific reports*, vol. 10, no. 15651, 2020.
28. M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, no. 5, pp. 593–599, 1953.
29. V. Puram, K. Karuppasamy, and J. P. Thomas, "Optimizing Quantum Circuits Using Algebraic Expressions," in *Computational Science – ICCS 2024*. Springer Nature Switzerland, 2024, pp. 268–276.
30. V. Gheorghiu, J. Huang, S. M. Li, M. Mosca, and P. Mukhopadhyay, "Reducing the CNOT Count for Clifford+T Circuits on NISQ Architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 1873–1884, 2022.

31. M. Nägele and F. Marquardt, "Optimizing zx-diagrams with deep reinforcement learning," *Machine Learning: Science and Technology*, vol. 5, no. 3, 2024.
32. B. Coecke and R. Duncan, "Interacting Quantum Observables," in *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, 2008, pp. 298–310.
33. N. Quetschlich, L. Burgholzer, and R. Wille, "Compiler Optimization for Quantum Computing Using Reinforcement Learning," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.
34. M. Chen, Y. Zhang, Y. Li, Z. Wang, J. Li, and X. Li, "Qcir: Pattern Matching Based Universal Quantum Circuit Rewriting Framework," in *ICCAD '22: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–8.
35. R. Iten, R. Moyard, T. Metger, D. Sutter, and S. Woerner, "Exact and Practical Pattern Matching for Quantum Circuit Optimization," *ACM Transactions on Quantum Computing*, vol. 3, no. 1, pp. 1–41, 2022.
36. F. Gemeinhardt, S. Klikovits, and M. Wimmer, "Gequpi: Quantum Program Improvement with Multi-Objective Genetic Programming," *Journal of Systems and Software*, vol. 219, 2025.
37. L. Wei, Z. Ma, Y. Cheng, and Q. Liu, "Genetic Algorithm Based Quantum Circuits Optimization for Quantum Computing Simulation," in *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE, 2021, pp. 1–8.
38. J. Pointing, O. Padon, Z. Jia, H. Ma, A. Hirth, J. Palsberg, and A. Aiken, "Quanto: Optimizing quantum circuits with automatic generation of circuit identities," *Quantum Science and Technology*, vol. 9, no. 4, 2024.
39. M. Xu, Z. Li, O. Padon, S. Lin, J. Pointing, A. Hirth, H. Ma, J. Palsberg, A. Aiken, U. A. Acar, and J. Zhihao, "Quartz: superoptimization of Quantum circuits," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 625–640.
40. J. Paykin, A. T. Schmitz, M. Ibrahim, X.-C. Wu, and A. Y. Matsuura, "Pcoast: A Pauli-Based Quantum Circuit Optimization Framework," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1. IEEE, 2023, pp. 715–726.
41. K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks, "A Verified Optimizer for Quantum Circuits," *Proceedings of the ACM on Programming Languages*, vol. 5, Jan. 2021.
42. The Coq Development Team. (2024, Sep.) The coq proof assistant. [Online]. Available: https://doi.org/10.5281/zenodo.11551307
43. R. Wille, L. Schmid, Y. Stade, J. Echavarria, M. Schulz, L. Schulz, and L. Burgholzer, "Qdmi - quantum device management interface: Hardware-software interface for the munich quantum software stack," in *QCE*, vol. 02, 2024, pp. 573–574.
44. C. G. Almudever, R. Wille, F. Sebastiano, N. Haider, and E. Alarcon, "From designing quantum processors to large-scale quantum computing systems," in *DATE*, 2024, pp. 1–10.
45. E. Kaya, J. Echavarria, M. N. Farooqi, A. Swierkowska, P. Hopf, B. Mete, L. Burgholzer, R. Wille, L. Schulz, and M. Schulz, "A software platform to support disaggregated quantum accelerators," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1646–1653.

46. A. Basermann, M. Epping, B. Fauseweh, M. Felderer, E. Lobe, M. Röhrig-Zöllner, G. Schmiedinghoff, P. K. Schuhmacher, Y. Setyawati, and A. Weinert, "Quantum software ecosystem design," in *Software Engineering 2025 Companion Proceedings, Fachtagung des GI-Fachbereichs Softwaretechnik, Karlsruhe, Germany, February 24-28, 2025*, K. Feichtinger, L. Sonnleithner, and H. Hajiabadi, Eds. Gesellschaft für Informatik e.V., 2025, p. 22. [Online]. Available: https://doi.org/10.18420/se2025-ws-22
47. R. Thunig, M. Johannfunke, T. Wang, and H. Schirmeier, "One flag to rule them all? on the quest for compiler optimizations to improve fault tolerance," in *2024 19th European Dependable Computing Conference (EDCC)*. IEEE, 2024, pp. 33–40.
48. J. Liu, J. Ren, J. Xie, J. Fang, and T. Wang, "Enhancing compiler optimization with reinforcement learning and monte carlo tree search," in *Proceedings of the 36th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE2024, vol. 2024. KSI Research Inc., Oct. 2024, pp. 146–151.
49. H. Ahmed, M. Fahim Ul Haque, H. Raza Khan, G. Nadeem, K. Arshad, K. Assaleh, and P. Cesar Santos, "Selecting the best compiler optimization by adopting natural language processing," *IEEE Access*, vol. 12, pp. 121 700–121 711, 2024.
50. A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A Survey on Compiler Autotuning using Machine Learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 96, pp. 1–42, 9 2018.
51. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 3 2003, pp. 204–215.
52. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed. Pearson Education, 2007.
53. Rigetti Computing. (2019, Jan.) pyQuil Documentation Release 2.3.0. [Online]. Available: https://readthedocs.org/projects/pyquil/downloads/pdf/v2.3.0/
54. quil lang. (2024) qvm: A High-Performance Quantum Virtual Machine. [Online]. Available: https://github.com/quil-lang/qvm
55. Sun Microsystems. (1995) x86 Assembly Language Reference Manual. [Online]. Available: https://docs.oracle.com/cd/E19641-01/802-1948/802-1948.pdf
56. Robert S. Smith; Rigetti & Co. Inc.; and contributors. (2021) Quil Specification. Version 2021.1. [Online]. Available: https://quil-lang.github.io/
57. lirem101. Github - LiRem101/parser-analyser: Analysation and optimization of Quil programs. [Online]. Available: https://github.com/LiRem101/parser-analyser
58. F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, pp. 1—19, Jul. 1970. [Online]. Available: https://doi.org/10.1145/390013.808479
59. K. D. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, 2006.
60. A. Peres and D. R. Terno, "Quantum Information and Relativity Theory," *Reviews of Modern Physics*, vol. 76, pp. 93–123, 1 2004.
61. D. Gottesman, "Theory of fault-tolerant quantum computation," *Physical Review A*, vol. 57, pp. 127–137, 1 1998.
62. W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, pp. 802–803, 1982.
63. D. Dieks, "Communication by EPR devices," *Physics Letters A*, vol. 92, no. 6, pp. 271–272, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0375960182900846
64. IBMQuantum. (2024) ibm_fez. [Online]. Available: https://quantum.ibm.com/services/resources?system=ibm_fez

65. ——. (2024) ibm_marrakesh. [Online]. Available: https://quantum.ibm.com/services/resources?system=ibm_marrakesh
66. ——. (2024) ibm_torino. [Online]. Available: https://quantum.ibm.com/services/resources?system=ibm_torino
67. I. IonQ. (2024) Ionq Aria: Practical Performance. [Online]. Available: https://ionq.com/resources/ionq-aria-practical-performance