Automating the Comparison of Quantum Compilers for Quantum Circuits

Marie Salm, Johanna Barzen, Frank Leymann,
Benjamin Weder, Karoline Wild

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{salm, barzen, leymann, weder, wild}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Salm2021_CompilerComparison,
  author    = {Salm, Marie and Barzen, Johanna and Leymann, Frank and
               Weder, Benjamin and Wild, Karoline},
  title     = {{Automating the Comparison of Quantum Compilers
               for Quantum Circuits}},
  booktitle = {Proceedings of the 15th Symposium and Summer School on
               Service-Oriented Computing (SummerSOC 2021)},
  pages     = {64--80},
  publisher = {Springer International Publishing},
  month     = sep,
  year      = 2021,
  doi       = {10.1007/978-3-030-87568-8_4}
}
```

Automating the Comparison of Quantum Compilers for Quantum Circuits

Marie Salm[0000−0002−2180−250X], Johanna Barzen[0000−0001−8397−7973], Frank Leymann[0000−0002−9123−259X], Benjamin Weder[0000−0002−6761−6243], and Karoline Wild[0000−0001−7803−6386]

University of Stuttgart, Institute of Architecture of Application Systems, Universitätsstraße 38, Stuttgart, Germany
{salm,barzen,leymann,weder,wild}@iaas.uni-stuttgart.de

Abstract. For very specific problems, quantum advantage has recently been demonstrated. However, current NISQ computers are error-prone and only support small numbers of qubits. This limits the executable circuit size of an implemented quantum algorithm. Due to this limitation, it is important that compiled quantum circuits for a specific quantum computer are as resource-efficient as possible. A variety of different quantum compilers exists supporting different programming languages, gate sets, and vendors of quantum computers. However, comparing the results of several quantum compilers requires (i) deep technical knowledge and (ii) large manual effort for translating a given circuit into different languages. To tackle these challenges, we present a framework to automate the translation, compilation, and comparison of a given quantum circuit with multiple quantum compilers to support the selection of the most suitable compiled quantum circuit. For demonstrating the practical feasibility of the framework, we present a prototypical implementation.

Keywords: Quantum Computing · NISQ · Decision Support · Compiler · NISQ Analyzer.

1 Introduction

Quantum computing is a highly discussed and emerging technology in research and industry [17]. Quantum advantage has already been demonstrated for specific problems [3,36]. Furthermore, a variety of quantum algorithms exists promising further beneficial breakthroughs in different areas such as computer and natural sciences [17]. Nevertheless, the *Noisy Intermediate-Scale Quantum (NISQ)* era is not yet overcome [19]. Existing gate-based quantum computers still only offer small numbers of qubits and high error rates. This strongly limits the circuit sizes of implemented quantum algorithms executable on existing quantum computers [11]: The number of required qubits, i.e. the *width*, has to be less than or equal to the number of qubits offered by the quantum computer. In addition, the number of sequential executable gates, i.e. the *depth*, can only be executed in a certain time frame, often in the range of micro seconds, otherwise too many errors would accumulate and interfere the results.

The width and depth of a quantum circuit is significantly influenced by (i) the qubit topology of the quantum computer, i.e. the connectivity between the qubits, (ii) the implemented gate set of the quantum computer, and (iii) the mapping and optimization algorithm of the used quantum compiler [4,25]. The quantum compiler maps the qubits and gates of the quantum circuit to the qubits and implemented gates of the real quantum computer [13].

Today, a great variety of quantum compilers exists [14]. They all differ in their implemented mapping and optimization algorithms. Especially in the NISQ era, the selection of the most suitable, i.e., the best optimizing quantum compiler for a given quantum circuit and quantum computer is tremendously important to make optimal use of the currently limited quantum resources. However, quantum compilers are accessed via *software development kits (SDKs)* providing libraries for implementing and executing quantum circuits on quantum computers or simulators [13,24]. These SDKs differ in their (i) supported programming languages, (ii) supported gate sets, and (iii) supported vendors of quantum computers for execution. Vendor-independent SDKs, such as pytket [30], exist facilitating the import of quantum circuits in various programming languages and a variety of gate sets with the respective hardware access. Nevertheless, several quantum compilers are accessed via vendor-specific SDKs, such as Qiskit [1] from IBM and Forest SDK [23] from Rigetti, only supporting vendor-specific languages, gate sets, and hardware access. As a result, it cannot simply be tested which compiler generates the most suitable compilation of a given quantum circuit and not every circuit can be compiled and executed on every existing quantum computer, because (i) the programming languages are not compatible and (ii) the gate sets differ between the different vendors. Thus, for comparing the results of quantum compilers deep technical knowledge about the SDKs and large manual effort for the translation of the quantum circuit into the different supported programming languages and gate sets is required. Several works, such as [15,16,30], already compare quantum compilers. However, they focus on presenting the overall strength of their compilation approaches with certain prepared benchmarks.

In this paper, we present a framework to compare the compilation results of different quantum compilers for a particular quantum circuit and quantum computer. Therefore, the framework enables to *(i) translate* the quantum circuit into the required programming languages of the SDKs of the given quantum compilers, *(ii) execute* the compilation with the selected compilers, and *(iii) analyze* the compilation results. Thus, the user can select the most suitable compilation. With the translation, we enable the decoupling of the SDK used for implementing a certain quantum circuit and the vendor used for execution. As basis for our framework, we analyze existing SDKs and their compilers, their supported programming languages, specification features, and quantum computers. For comparing the compiled circuits, metrics, such as width and depth, are chosen. To demonstrate the practical feasibility of the framework, a prototypical implementation is presented supporting the t$|$ket$\rangle$ compiler [30], the Qiskit Transpiler [1], and the Quilc compiler [23]. The framework is plug-in based, such that the support of further metrics, programming languages, compilers, and SDKs can be added.

The remainder of the paper is structured as follows: In Sect. 2 the fundamentals and the problem statement are introduced. In Sect. 3 existing SDKs and their compilers are analyzed and compared. Furthermore, the suitability of compilations is discussed. Sect. 4 describes the concept of the framework. Sect. 5 presents the system architecture and prototype. Sect. 6 validates the framework with a case study. Sect. 7 discusses the concept of the framework. Sect. 8 presents related work and Sect. 9 concludes the work and gives an outlook.

2 Fundamentals and Problem Statement

In this section, we introduce the fundamentals about quantum compilation and present current challenges using quantum compilers. Finally, we present the problem statement and the research questions of this work.

2.1 Quantum Circuits and Quantum Compilers

For gate-based quantum computing, quantum circuits represent implemented quantum algorithms [18]. A quantum circuit is described by qubits and quantum gates manipulating the states of the qubits as well as the order in which the gates are performed. For the execution on a certain quantum computer, also called *quantum processing unit (QPU)*, a quantum compiler has to map the defined quantum circuit to the properties of the quantum computer [13]. In general, gate-based quantum computers differ in their implemented gate set, the number of qubits, the qubit topology, and errors that can occur during the execution of a quantum circuit [4,22,25]. In a first step, the quantum compiler replaces non-implemented gates of a defined quantum circuit by a sequence of implemented gates [11,13]. Moreover, due to the non-complete topology graph of the selected quantum computer, further gates and qubits are required for the usage of multi-qubit gates [5,33]: The state of non-directly connected qubits has to be transferred over connected qubits. During the compilation (i) the qubits limiting the maximum width and (ii) the error rates of the gates and the decoherence time of the qubits limiting the depth of a quantum circuit have to be considered [30,33]. Therefore, quantum compilers perform several optimizations, such as parallelizing gates and replacing gates with high error rates [30]. The mapping procedure is NP-complete [29]. Also, it can lead to a significant increase of the depth of a circuit challenging a successful execution on today's NISQ computers with high error rates and small qubit numbers [5,11,19].

Today, a large number of quantum compilers exist and is still growing [14]. Each compiler performs another mapping and optimization algorithm. Hence, the resulting compiled quantum circuits of a given input circuit for a certain quantum computer differ in their properties. This makes it difficult to predict which compiler suits best for a certain quantum circuit and quantum computer. However, as we are still in the NISQ era, it is important to optimize a quantum circuit by reducing its depth and width [25], to minimize the impact of increasing errors and use the quantum resources efficiently.

Besides the width and depth, further circuit properties, i.e. metrics, exist and are commonly used to compare compiled circuits describing the inner structure of a circuit, e.g., the number of two-qubit gates with their high error rates or the total number of gates [30]. Another proposed metric roughly estimates if an execution of a circuit with a certain width and depth could be successful dependent on the effective error rate of the given quantum computer [4,11,19,25]. Nevertheless, for sharpening and applying this metric the effective error rate of the quantum computer has to be determined which is challenging as the exact composition of the effective error rate is not yet known [21,25,33]. Thus, the two metrics width and depth build a solid expandable set for describing the decisive size of a circuit enabling a concise overview to compare between multiple compilations of different quantum compilers [25,30].

2.2 Quantum Software Development Kits

A quantum compiler is often accessed via an SDK. An SDK provides tools and libraries for developing a quantum circuit, to compile it, and, finally, execute it on a quantum computer [10]. Most of the current vendors of quantum computers, such as IBM and Rigetti, provide proprietary SDKs, e.g. Qiskit [1] and Forest SDK [23], for accessing their quantum compilers and quantum computers [24]. However, a few SDKs exist supporting several vendors, such as pytket [30] and ProjectQ [32]. Each SDK offers a different set of supported programming languages, customizable quantum computer specifications, and gate sets [34]. Thus, a circuit is not necessarily interchangeable between different SDKs. This denies the flexibility of testing several compilers and executing on a wide range of quantum computers. An overview about the features of existing SDKs and their compilers is given in Sect. 3. Thereby, many properties of the SDKs are taken into account as they determine, e.g., import and export programming languages.

2.3 Problem Statement

It is hard to estimate in advance which quantum compiler returns the most suitable compilation, in terms of smallest width and depth, for a given quantum circuit and quantum computer. However, simply testing every compiler with a certain quantum circuit and quantum computer is challenging. First, the circuit has to be written in a programming language and gate set supported by the SDK providing the specific quantum compiler, otherwise it has to be rewritten. In addition, for a compilation on the desired quantum computer, its vendor has to be supported by the SDK. Hence, the first research question (RQ) is as follows:

RQ 1: *How can potentially relevant quantum compilers be identified for the compilation of a certain quantum circuit on a certain quantum computer?*

While a detailed analysis of programming languages and vendors supported by existing quantum compilers and their SDKs builds a good basis, comparing existing quantum compilers to find the most suitable compiled circuit for a specific quantum computer demands a lot of manual effort and deep technical knowledge from the user. Therefore, the second RQ is as follows:

RQ 2: *How can the comparison of compiled circuits of different quantum compilers for a given quantum circuit and quantum computer be automated?*

For addressing the formulated RQs, we analyzed quantum compilers and SDKs and realized a compilation comparison framework, presented in the following.

3 Analysis of Quantum Compilers

In this section, we present an analysis of SDKs and their quantum compilers, as seen in Table 1. Furthermore, we discuss the suitability of compiled circuits.

Table 1. Compiler-specific Criterion on SDKs.

	SDKs (Compilers)	Import Languages	Export Languages	Backend Vendors	Custom QPUs	Custom Compilation Gate Set
Propr. SDKs	Cirq	OpenQASM, Cirq-JSON, Quirk-JSON	OpenQASM, Quil, Cirq-JSON	Google, AQT, Pasqal	yes	yes
	Forest (Quilc)	PyQuil*, Quil	Quil	Rigetti	yes	no
	Qiskit (Transpiler)	OpenQASM, Qiskit*	OpenQASM	IBM	yes	yes
Independent SDKs	pytket (t|ket⟩)	OpenQASM, Qiskit*, PyZX*, PyQuil*, Cirq*, Quipper	OpenQASM, Qiskit*, PyZX*, PyQuil*, Cirq*, Qulacs*, Q#, ProjectQ*	AQT, Amazon Braket, Honeywell, Rigetti, IBM, Microsoft QDK	yes	yes
	staq	OpenQASM	OpenQASM, ProjectQ*, Quil, Q#, Cirq*	no	no	no
	ProjectQ	ProjectQ*	ProjectQ*	IBM, AQT, Amazon Braket, IonQ	no	yes

3.1 Compiler-specific Analysis of SDKs

To address **RQ 1**, different SDKs and their quantum compilers are analyzed (see Table 1). This analysis focuses on the integration and configuration properties for certain compilers. Thus, the support of different programming languages, vendors, custom quantum computers, and custom compilation gate sets is considered. Note that vendors of simulators are excluded. The selected set is only an excerpt of open-source accessible SDKs and serves for exemplary purposes showing their distinct properties. Detailed analyses of SDKs have been shown, e.g., by [7,10,34].

Most of the analyzed SDKs support the import and export of multiple programming languages. In general, the supported languages can be split in high-level programming languages, such as Python, and assembly languages, such as Quil [31] and OpenQASM [6,34]. However, most SDKs offer their own Python libraries (marked with *), such that implemented circuits are not interchangeable. Most SDKs, except Forest SDK [23] and ProjectQ [32], support OpenQASM.

From the vendor-specific SDKs, only Cirq [20] supports multiple vendors, as shown in Table 1. Pytket [30] supports a great variety of vendors. Staq [2] supports many export languages but considers only mocked quantum computers.

Many of the considered SDKs support the specification of custom quantum computers, their qubit topologies, gate sets, and error rates for experimental compilation in their proprietary format. But less quantum compilers natively support customizing the target gate set, such that the previously specified gates cannot be retrieved in the compiled quantum circuit.

Thus, for comparing the outcomes of a wide range of existing quantum compilers, the given quantum circuit has to be translated into several programming languages. In addition, tackling **RQ 1**, the SDK of a considered quantum compiler must natively support the given quantum computer for comparison. Otherwise, the SDK needs to offer the possibility to specify custom quantum computers. However, a complete specification is often associated with a lot of manual effort. The analysis is the basis of the framework introduced in the following Sect. 4.

3.2 Suitability of Compiled Quantum Circuits

After compiling a quantum circuit with a compiler, its compilation result is assumed to be *suitable* if it is successfully executable such that the correct result is clearly identifiable and not too much interfered by errors [11,25]. Therefore, the number of qubits, decoherence times, and error rates of the quantum computer determine the maximum width and depth of an executable and, thus, suitable quantum circuit. However, a general prioritization of width or depth cannot be determined. Quantum computers can have a great number of stable qubits but only support the execution of circuits with small depths, or vice versa [11]. Moreover, if, e.g., two compiled circuits are suitable, i.e. have a smaller or equal width and depth than the number of qubits and the maximum executable depth of the quantum computer, it cannot simply be determined which circuit is to be preferred. Thus, selecting the most suitable compilation depends on multidimensional aspects and, currently, cannot be solved by automation.

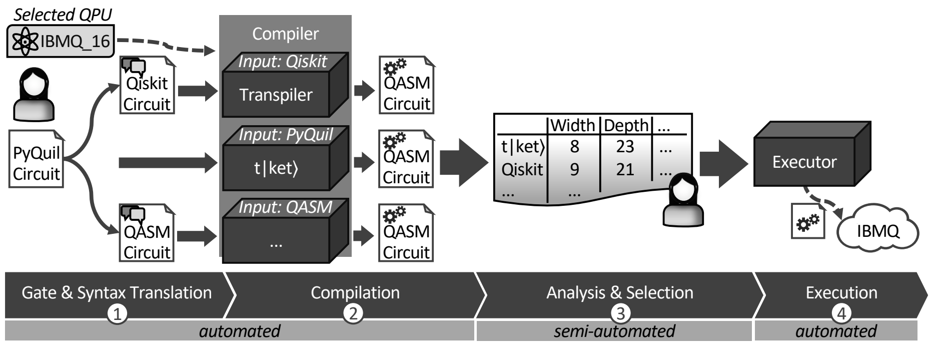

Fig. 1. Automated compilation and comparison of circuits with multiple compilers.

4 Framework for Compilation Comparison

To address **RQ 2**, we present a framework to compare the compiled circuits of different quantum compilers for a certain quantum circuit and quantum computer. The overall approach is shown in Fig. 1. The phases 1,2, and 4 are completely automated while phase 3 is semi-automated. First, the given circuit is translated into the programming languages and gate sets supported by the SDKs of the selected compilers. The translated circuits and the information about the selected quantum computer serve as input for the compilation in the second step. After compilation, the widths and depths of the resulting circuits are analyzed and suitable results are recommended to the user. The user, then, manually selects the most suitable compilation. Finally, the selected circuit is executed. The details about each phase are provided in the following.

4.1 Gate and Syntax Translation

The basis for translation is the quantum circuit and selected quantum computer. Based on Sect. 3, the SDKs of suitable quantum compilers (i) have to support the selected quantum computer and (ii) determine the required programming languages. Thus, in the *(1) Gate and Syntax Translation* phase, the circuit for compilation, its programming language, e.g. Python for *PyQuil*, and the desired quantum computer, e.g. *IBMQ_16*, serve as input for the automated translation, compilation, and comparison, as seen in Fig. 1. For each SDK, the supported vendors are specified. Thus, based on the selected quantum computer the matching quantum compiler can be determined. Then, it is identified which languages are supported by the selected SDKs. For an SDK not supporting the language of the given circuit, the syntax of the circuit is mapped to the syntax of the supported language. For example, the circuit is mapped to the Python library Qiskit, as the SDK *Qiskit* and its Transpiler does not support PyQuil. Thereby, an intermediate format reduces the number of possible translation combinations. For the gate translation, a mapping table is defined for each language. Gates that are not supported by the SDK, e.g. hardware-specific or custom gates, are either replaced by a subroutine of supported gates or unitary matrices defining the specific gates as custom gates. A simple example is the SWAP gate, that, if not supported, can be replaced by three CNOT gates [18]. For defining a matrix, the target SDK has to support custom gates. In general, an arbitrary gate can be approximated by using a universal set of quantum gates [18].

4.2 Compilation

In the *(2) Compilation* phase, the circuit in the respective input language as well as the information about the quantum computer, i.e. its name, are taken as input for each compiler. Then, the selected quantum compilers compile in parallel. The returned compilations are in the programming language required for the later execution on the selected quantum computer, e.g. for IBM machines OpenQASM is required [6]. As SDKs and their compilers are selected dependent on their vendor support, they natively support their required languages and gates.

4.3 Analysis and Selection

In the *(3) Analysis and Selection* phase, the compiled quantum circuits are analyzed to determine their widths and depths. Therefore, the *NISQ Analyzer* is used to compare the depth of the compiled quantum circuits with the estimated maximum executable depth of the quantum computer [24]. The NISQ Analyzer is a tool that selects suitable quantum circuit implementations based on the chosen quantum algorithm and input values and determines if they are executable on available quantum computers. The maximum executable depth of a quantum computer is estimated by dividing the average decoherence time of all available qubits through the maximum gate time [26]. In general, if the width of a quantum circuit is greater than the number of qubits of the quantum computer, existing quantum compilers automatically reject the circuit. If width and depth of a compilation is less than or equal to the maximum depth and the number of qubits of the quantum computer, it is executable and recommended to the user as suitable compiled circuit. All other compilations are filtered out automatically beforehand in the analysis. If multiple compiled circuits are suitable, the user, then, can manually select the most suitable compiled quantum circuit based on their comparable widths and depths, as discussed in Sect. 3.2.

4.4 Execution

In the *(4) Execution* phase, the most suitable compilation selected by the user is executed on the quantum computer, as shown in Fig. 1. As the SDKs containing the quantum compilers also provide access to the considered quantum computer, the respective SDK is used for the automated execution. Therefore, the user does not have to manually deploy the required SDK and compiled circuit for execution. Instead, the *Executor* supports all SDKs used for compilation and automates the execution of compilations. Finally, the execution result is presented to the user.

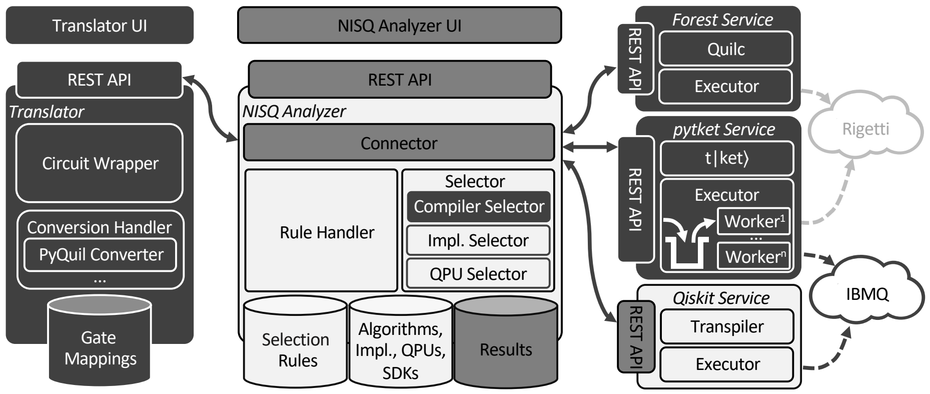

Fig. 2. Architecture for translating, compiling, and comparing quantum circuits.

5 System Architecture and Prototype

In this section, the overall system architecture is presented, as seen in Fig. 2. Furthermore, the prototypical implementation of the framework is described.

5.1 System Architecture

In Fig. 2 the overall architecture of the comparison framework is shown. It consists of an extension of the NISQ Analyzer [24] as well as new components for the translation and a set of compilation and execution services. Thereby, new components are dark, extended components are middle and already existing components are light grey. The *Translator* on the left side of Fig. 2 provides an HTTP REST API used by the *Translator User Interface (UI)* offering the modeling and translation of quantum circuits in multiple programming languages. The *Circuit Wrapper* component inside the Translator controls the import, conversion, and export of a given quantum circuit in the defined programming language. For the conversion between the import and the export languages an intermediate format is used. Therefore, the *Conversion Handler* component invokes the language-specific converter plug-in for converting from the language of the given circuit into the intermediate format and vice versa. For example, the *PyQuil Converter* supports the import and export of circuits written with PyQuil and in Quil. For converting the language and gate set supported by an SDK to the other, the defined gate mappings are stored in a repository.

For the compilation, analysis, and execution, the *NISQ Analyzer*, in the middle of Fig. 2, is extended. The HTTP REST API and the NISQ Analyzer UI are extended triggering the comparison and selection process. The new *Compiler Selector* component in the NISQ Analyzer backend identifies available SDKs and, thus, compilers supporting the vendor of the selected quantum computer and required languages. It also verifies the executability of compilations on the quantum computer. The *Connector* enables the interaction with the Translator and the *Compilation and Execution (Com&Ex) Services*. The analysis and execution results, including the transpiled circuits, are stored as provenance data in a repository enabling the learning of compilations and executions in the future [13].

On the right side of Fig. 2, three exemplary Com&Ex Services are shown. Each supported SDK is wrapped by an interface enabling the compilation and execution with the accessible quantum compiler on a supported quantum computer.

To start the compilation with available quantum compilers, the user inserts the quantum circuit using the NISQ Analyzer UI, selects its programming language and chooses the desired quantum computer. The Compiler Selector identifies the Com&Ex Services supporting the vendor of the quantum computer. Then, the Compiler Selector invokes the Connector to call the Translator if a Com&Ex Service requires another language than the circuit is written in. The circuit, information about its own language and the required language are passed to the Translator. The Circuit Wrapper invokes the Conversion Handler for the translation into the syntax and gate set of the required language and returns the resulting circuit to the NISQ Analyzer. The Compiler Selector passes the circuits to the respective Com&Ex Services for compilation. After compilation, each Com&Ex Service analyzes the width and depth of its compiled circuit and returns the information to the NISQ Analyzer [24]. Then, the Compiler Selector compares the depth of the compilations with the estimated maximum executable depth of the quantum computer proofing executability. Thereby, non-suitable circuits are filtered out and all suitable circuits, their widths and depths are returned to the user. Then, the user can select the most suitable compilation for execution. Therefore, the compiled circuit is passed to its respective Com&Ex Service, where it is sent to the cloud service of the vendor. Finally, the execution result is shown. The NISQ Analyzer can be extended by plug-ins to support further metrics besides comparing width and depth. The Connector can be extended to support further Com&Ex Services implementing the defined interfaces. Additional languages can be supported by implementing respective converters.

5.2 Prototype

The Translator UI and the NISQ Analyzer UI are implemented in TypeScript. The Translator and the Com&Ex Services are written in Python with the framework Flask. The NISQ Analyzer is written in Java with the framework Spring Boot. A detailed discussion of the implementation of the NISQ Analyzer and the Com&Ex Services can be found in [24]. The entire prototype is available open-source[1,2,3].

The intermediate format of the Translator is based on *QuantumCircuit*[4] from Qiskit. As shown in Sect. 3, Qiskit enables defining custom gates. It also supports a great set of standard gates natively. Thus, the import and export of OpenQASM and the import of Python for Qiskit is natively supported, as the SDK Qiskit

[1]https://github.com/UST-QuAntiL
[2]https://github.com/UST-QuAntiL/nisq-analyzer-content
[3]https://youtu.be/I5l8vaA-zO8
[4]https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html

provides respective functions. Exporting a circuit to Python for Qiskit requires additional functions for extracting the *QuantumCircuit* instructions into the required Python-based syntax. Additionally, as proof-of-concept, Quil and Python for PyQuil are supported by the Translator. For example, when importing a circuit written with PyQuil to *QuantumCircuit*, first, the implemented PyQuil Converter uses the Forest SDK returning a PyQuil-specific *Program*[5]. Then, the PyQuil Converter iterates over the *Program* constructing an equivalent *QuantumCircuit*. Exporting from a circuit defined as *QuantumCircuit* to a circuit written with PyQuil works vice versa. By iterating over the instructions, defined qubit registers and classical bits of a circuit are transferred. For transferring a gate, possible gate parameters and the qubits on which the gate is applied are extracted. The gate mapping itself is stored in a dictionary. Thereby, (i) an equivalent gate, (ii) a subroutine of gates, or (iii) a matrix is defined supported by the target language.

For proof-of-concept, currently three SDKs and their quantum compilers are supported for compilation and execution: Forest SDK with its Quilc compiler [23], pytket with its t$|$ket$\rangle$ compiler [30], and Qiskit with its Transpiler [1].

6 Case Study

In this section, the prototype of the framework from Sect. 5 is validated. Thus, two use cases are presented. In the first use case, a Quil circuit implementing the Shor algorithm [27] is used for the compilation on the *ibmq_16_melbourne* [9] offering 15 qubits and an estimated maximum gate depth of 32 levels. In the second use case, a Qiskit circuit implementing the Grover algorithm [8] is compiled on a mocked Rigetti quantum computer supporting a fully connected 9-qubit-topology with a maximum depth of 120 levels. Since we cannot access real quantum computers of Rigetti, the usage of the Quilc compiler is shown with a mocked Rigetti quantum computer. Each circuit is provided in our GitHub repository[2].

6.1 Compilation on IBM Hardware

The Shor algorithm factorizes an odd integer and computes its prime factors exponentially faster than a classical counterpart [27]. The considered circuit *shor-15-quil* in Quil factorizes 15. As *ibmq_16_melbourne* is the target, the Qiskit Service and the pytket Service are automatically selected for compilation, as Forest SDK [23] does not support IBM. Both Com&Ex Services do not natively support Quil. Thus, for the pytket Service, the Quil circuit is translated into PyQuil which is one of the supported languages of pytket presented in Sect. 3. For the Qiskit Service, it is translated into Python for Qiskit. Then, the quantum computer and the translated circuits are passed to the respective Com&Ex Services. The resulting depth and width of the compilations can be seen in Table 2. Both are executable on the *ibmq_16_melbourne* according to the NISQ Analyzer and are recommended to the user. Based on the widths and depths of the compilations, the user can select which should be executed.

[5]https://pyquil-docs.rigetti.com/en/latest/apidocs/program.html

Table 2. Compilation results for *shor-15-quil*.

Compiler	Width	Depth	Backend	Executable
Qiskit Transpiler	4	10	ibmq_16_melbourne	yes
t|ket⟩	5	8	ibmq_16_melbourne	yes

6.2 Compilation on Rigetti Hardware

The Grover algorithm, in general, searches items in unsorted lists and has a quadratic speed up compared to a classical algorithm [8]. The algorithm is also applied to more specific problems, such as the *Boolean satisfiability problem (SAT)*. The implemented circuit *grover-SAT-qiskit* in Qiskit solves the SAT problem $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B)$. The target is a mocked Rigetti quantum computer. Thus, the Forest Service and the pytket Service are automatically selected. The quantum computer and the circuit are directly passed to the pytket Service. For the Forest Service, the circuit is translated into Quil. The mocked Rigetti and the translated circuit are, then, passed to the Forest Service. As we set the maximum depth of the mocked Rigetti to 120, the compilation of the Quilc compiler is not executable according to the NISQ Analyzer, see Table 3. Thus, only the compilation of the t|ket⟩ compiler is recommended to the user. Finally, the user can trigger the automated execution of the recommended compilation.

Table 3. Compilation results for *grover-SAT-qiskit*.

Compiler	Width	Depth	Backend	Executable
Quilc	8	164	mocked Rigetti	no
t|ket⟩	8	117	mocked Rigetti	yes

7 Discussion and Limitations

The presented framework enables the decoupling of quantum circuits and their SDKs and the comparison of compilation results of quantum compilers for arbitrary circuits. Thereby, the automated (i) translation, (ii) compilation, and (iii) analysis of a given quantum circuit and quantum computer is provided to support the selection of the most suitable compilation. As discussed in Sect. 3, automatically prioritizing a compiled circuit if, e.g., one executable compilation has the smallest depth but another has the smallest width, is a multidimensional challenge and is currently not yet supported. However, all non-suitable compilation results in terms of width and depth are filtered out and, thus, only suitable compilation results are returned to the user for the final selection.

At present, we only consider quantum compilers if the selected quantum computer is natively supported. However, as shown in Sect. 3, some quantum compilers support the specification of custom quantum computers. The framework could be further extended to retrieve quantum computer specifications from a certain vendor and to translate them into the format to custom define this quantum computer for the compiler. By iterating over the individual instructions for translating a defined circuit into another programming language, non-optimal replacements can occur which can also affect the resource efficiency of the compiled circuit. Furthermore, no automated equivalence verification of translated circuits is currently supported. Nevertheless, the translated circuits presented in Sect. 6 were verified by comparing their execution results with those of the initial quantum circuits. Currently, only the programming languages OpenQASM and Quil and the Python libraries Qiskit and PyQuil are supported. However, as our framework bases on a plug-in based system, additional plug-ins can be implemented supporting further languages, SDKs, metrics, and compilers.

We currently use Com&Ex Services to support all SDKs and their compilers utilized for the compilation of quantum circuits and the automated execution of their compilations. For the automated deployment and execution also existing deployment automation technologies can be used [35].

8 Related Work

Comparing quantum compilers is covered in several works. Sivarajah et al. [30] present their software development platform to access the quantum compiler $t|ket\rangle$ supporting multiple programming languages and vendors, as shown in Sect. 3. They compare their compiler with the Quilc compiler and the Qiskit Transpiler. Therefore, several test circuits are used as benchmarks and executions on quantum computers were performed. Also, in the work of [2,15,16] the performance of the presented compilation and optimization algorithms were compared with the Qiskit Transpiler and the Quilc compiler, by several test circuits and certain quantum algorithm implementations. Thus, their focus is on showing the overall performance of their compilers using defined benchmarks. The focus of our framework is on supporting the selection of the most suitable compilation situation-based for an arbitrary quantum circuit on a certain quantum computer.

In the work of Mills et al. [14] the $t|ket\rangle$ compiler and the Qiskit Transpiler are compared. Thereby, they are investigating their different compilation and optimization algorithms in consideration of the physical properties of implemented qubits on quantum computers. Nevertheless, their focus is on aspects considered during the compilation process on quantum computers. Thus, they do not support the generic approach of comparing and selecting compilations of quantum compilers for arbitrary quantum circuits.

Arline[6] introduces a framework for the automated benchmarking of quantum compilers[7]. For the implementation of quantum circuits, the framework offers

[6]https://www.arline.io

[7]https://github.com/ArlineQ

its own programming language but also supports the import and export of OpenQASM circuits. Thereby, gate sets from quantum computers of IBM, Rigetti, and Google are supported. Currently, compilers of Qiskit and Cirq are considered and can be combined for compiling quantum circuits. The benchmarking analysis of the quantum compilers and its compilations bases on metrics, e.g. the depth of the compiled circuit, the gate count, and the compiler runtime. However, analyzing the executability on a certain quantum computer afterward is not considered. Instead of accessing real quantum computers, properties are hard-coded. Thus, no execution is supported. Furthermore, no support of several programming languages and its translations is given.

9 Conclusion and Future Work

In this paper, we presented a framework to compare the compilations of different quantum compilers for a certain quantum circuit and quantum computer. Answering **RQ 2**, the framework automatically (i) translates the quantum circuit into the programming languages required by the SDKs of the available compilers, (ii) compiles with selected quantum compilers, and (iii) analyzes the compilations. Thereby, the dependency between quantum circuits and their SDKs are solved, such that the execution on an arbitrary quantum computer is enabled. For the comparison of the compilations, the widths and depths of the resulting circuits are determined and their executability on the chosen quantum computer is examined. The most suitable quantum circuit can, then, directly be executed on the respective quantum computer. As basis of our framework, several SDKs and their compilers are analyzed towards compilation answering **RQ 1**.

The framework extends the NISQ Analyzer [24] which is part of the platform PlanQK[8] focusing on sharing and executing quantum software [12,13]. In the future, we plan to implement further Com&Ex Services wrapping additional SDKs to increase the variety of available quantum compilers. Additionally, we plan to support an automated equivalence verification framework to verify the Translator component systematically and ensure the equivalence of its translations [28]. Furthermore, we plan to investigate in an intermediate format to support the specification of arbitrary quantum computers for SDKs supporting custom quantum computers. We also plan to offer further metrics, e.g. the count of multi-qubit gates and the estimation of a successful executability [25], to improve the comparison of compiled quantum circuits.

Acknowledgements

We would like to thank Thomas Wangler for the implementation of the Translator.

This work was partially funded by the BMWi project *PlanQK* (01MK20005N) and the DFG's Excellence Initiative project *SimTech* (EXC 2075 - 390740016).

[8]https://planqk.de/en/

References

1. Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., et al.: Qiskit: An Open-source Framework for Quantum Computing (2019). https://doi.org/10.5281/zenodo.2562111
2. Amy, M., Gheorghiu, V.: staq—a full-stack quantum processing toolkit. Quantum Science and Technology **5**(3), 034016 (2020). https://doi.org/10.1088/2058-9565/ab9359
3. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**(7779), 505–510 (2019). https://doi.org/10.1038/s41586-019-1666-5
4. Bishop, L., Bravyi, S., Cross, A., Gambetta, J., Smolin, J., March: Quantum volume (2017)
5. Cowtan, A., Dilkes, S., Duncan, R., Krajenbrink, A., Simmons, W., et al.: On the Qubit Routing Problem. In: 14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 135, pp. 5:1–5:32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.TQC.2019.5
6. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open quantum assembly language (2017)
7. Fingerhuth, M., Babej, T., Wittek, P.: Open source software in quantum computing. PLOS ONE **13**(12), 1–28 (2018). https://doi.org/10.1371/journal.pone.0208561
8. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 212–219 (1996)
9. IBMQ team: 15-qubit backend: IBM Q 16 Melbourne backend specification V2.3.6 (2021), `https://quantum-computing.ibm.com`
10. LaRose, R.: Overview and Comparison of Gate Level Quantum Software Platforms. Quantum **3**, 130 (2019). https://doi.org/10.22331/q-2019-03-25-130
11. Leymann, F., Barzen, J.: The bitter truth about gate-based quantum algorithms in the NISQ era. Quantum Science and Technology **5**(4), 1–28 (2020). https://doi.org/10.1088/2058-9565/abae7d
12. Leymann, F., Barzen, J., Falkenthal, M.: Towards a Platform for Sharing Quantum Software. In: Proceedings of the 13th Advanced Summer School on Service Oriented Computing (2019). pp. 70–74. IBM Technical Report (RC25685), IBM Research Division (2019)
13. Leymann, F., Barzen, J., Falkenthal, M., Vietz, D., Weder, B., et al.: Quantum in the Cloud: Application Potentials and Research Opportunities. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020). pp. 9–24. SciTePress (2020)
14. Mills, D., Sivarajah, S., Scholten, T.L., Duncan, R.: Application-motivated, holistic benchmarking of a full quantum computing stack (2021). https://doi.org/10.22331/q-2021-03-22-415
15. Murali, P., Baker, J.M., Javadi-Abhari, A., Chong, F.T., Martonosi, M.: Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 1015–1029. ASPLOS '19, ACM (2019). https://doi.org/10.1145/3297858.3304075
16. Murali, P., Linke, N.M., Martonosi, M., Abhari, A.J., Nguyen, N.H., et al.: Full-stack, real-system quantum computer studies: Architectural comparisons and design

insights. In: Proceedings of the 46th International Symposium on Computer Architecture. p. 527–540. ISCA '19, ACM (2019). https://doi.org/10.1145/3307650.3322273

17. National Academies of Sciences, Engineering, and Medicine: Quantum Computing: Progress and Prospects. The National Academies Press (2019)
18. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, 10th edn. (2011)
19. Preskill, J.: Quantum Computing in the NISQ era and beyond. Quantum **2**, 79 (2018). https://doi.org/10.22331/q-2018-08-06-79
20. Quantum AI team and collaborators: Cirq (2020)
21. Resch, S., Karpuzcu, U.R.: Benchmarking quantum computers and the impact of quantum noise (2019)
22. Rieffel, E., Polak, W.: Quantum Computing: A Gentle Introduction. The MIT Press, 1st edn. (2011)
23. Rigetti: Docs for the Forest SDK (2021), `https://pyquil-docs.rigetti.com/`
24. Salm, M., Barzen, J., Breitenbücher, U., Leymann, F., Weder, B., et al.: The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms. In: Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020). pp. 66–85. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-64846-6_5
25. Salm, M., Barzen, J., Leymann, F., Weder, B.: About a Criterion of Successfully Executing a Circuit in the NISQ Era: What $wd \ll 1/\epsilon_{\text{eff}}$ Really Means. In: Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS 2020). pp. 10–13. ACM (2020). https://doi.org/10.1145/3412451.3428498
26. Sete, E.A., Zeng, W.J., Rigetti, C.T.: A functional architecture for scalable quantum computing. In: 2016 IEEE International Conference on Rebooting Computing (ICRC). pp. 1–6 (2016). https://doi.org/10.1109/ICRC.2016.7738703
27. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997). https://doi.org/10.1137/S0036144598347011
28. Singhal, K., Rand, R., Hicks, M.: Verified translation between low-level quantum languages. The First International Workshop on Programming Languages for Quantum Computing (2020)
29. Siraichi, M.Y., Santos, V.F.d., Collange, S., Quintão Pereira, F.M.: Qubit Allocation. In: CGO 2018 - International Symposium on Code Generation and Optimization. pp. 1–12 (2018). https://doi.org/10.1145/3168822
30. Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., et al.: t|ket⟩: A retargetable compiler for nisq devices. Quantum Science and Technology **6** (2020). https://doi.org/10.1088/2058-9565/ab8e92
31. Smith, R.S., Curtis, M.J., Zeng, W.J.: A practical quantum instruction set architecture (2017)
32. Steiger, D.S., Häner, T., Troyer, M.: Projectq: an open source software framework for quantum computing. Quantum **2**, 49 (2018). https://doi.org/10.22331/q-2018-01-31-49
33. Tannu, S.S., Qureshi, M.K.: Not all qubits are created equal: A case for variability-aware policies for nisq-era quantum computers. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 987–999. ASPLOS '19, ACM (2019). https://doi.org/10.1145/3297858.3304007

34. Vietz, D., Barzen, J., Leymann, F., Wild, K.: On Decision Support for Quantum Application Developers: Categorization, Comparison, and Analysis of Existing Technologies. In: Computational Science – ICCS 2021. pp. 127–141. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-77980-1_10
35. Wild, K., Breitenbücher, U., Harzenetter, L., Leymann, F., Vietz, D., Zimmermann, M.: TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications. In: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC). IEEE Computer Society (2020). https://doi.org/10.1109/EDOC49727.2020.00024
36. Zhong, H.S., Wang, H., Deng, Y.H., Chen, M.C., Peng, L.C., et al.: Quantum computational advantage using photons. Science (2020)