Predicting Good Quantum Circuit Compilation Options

Nils Quetschlich* Lukas Burgholzer† Robert Wille*‡
*Chair for Design Automation, Technical University of Munich, Germany
†Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
‡Software Competence Center Hagenberg GmbH (SCCH), Austria
nils.quetschlich@tum.de lukas.burgholzer@jku.at robert.wille@tum.de
https://www.cda.cit.tum.de/research/quantum

***Abstract*—Any potential application of quantum computing, once encoded as a quantum circuit, needs to be compiled in order to be executed on a quantum computer. Deciding which qubit technology, which device, which compiler, and which corresponding settings are best for the considered problem—according to a measure of *goodness*—requires expert knowledge and is overwhelming for end-users from different domains trying to use quantum computing to their advantage. In this work, we treat the problem as a statistical classification task and explore the utilization of supervised machine learning techniques to optimize the compilation of quantum circuits. Based on that, we propose a framework that, given a quantum circuit, predicts the best combination of these options and, therefore, *automatically* makes these decisions for end-users. Experimental evaluations show that, considering a prototypical setting with 3000 quantum circuits, the proposed framework yields promising results: for more than three quarters of all unseen test circuits, the *best* combination of compilation options is determined. Moreover, for more than 95% of the circuits, a combination of compilation options within the top-three is determined—while the median compilation time is reduced by more than one order of magnitude. Furthermore, the resulting methodology not only provides end-users with a prediction of the best compilation options, but also provides means to extract explicit knowledge from the machine learning technique. This knowledge helps in two ways: it lays the foundation for further applications of machine learning in this domain and, also, allows one to quickly verify whether a machine learning algorithm is reasonably trained. The corresponding framework and the pre-trained classifier are publicly available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT).**

I. Introduction

The capabilities of quantum computers are steadily evolving—achieving more physical qubits, lower error rates, and faster operations. Devices are increasingly made available through manufacturers, such as IBM Quantum, or third-party cloud providers, such as Amazon Web Services. Furthermore, several *Software Development Kits* (SDKs) for programming these devices have been developed, e.g., IBM's Qiskit [1], Quantinuum's TKET [2], Google's Cirq [3], Xanadu's Pennylane [4], and Rigetti's Forest [5]. As a result of this progress, academia and industry have started to elaborate possible applications for this new technology. In fact, the potential of quantum computing is currently being explored for several application domains such as chemistry (e.g., [6]), finance (e.g., [7]), optimization (e.g., [8]), and machine learning (e.g., [9]). Even dedicated workflows describing the steps required to solve a classical problem using quantum computing start to emerge (e.g., [10]).

The process towards executing corresponding quantum algorithms on quantum computers has a lot of similarities to the process of executing classical algorithms or programs on classical computers. In the classical world, a program must be transformed so that it can be executed on a specific *Central Processing Unit* (CPU) which provides its own *Instruction Set Architecture* (ISA). This transformation process is called *compilation* and is conducted by *compilers*—usually coming with numerous settings and optimization parameters. Similarly, once an application for quantum computing has been encoded as a quantum circuit, it needs to be *compiled* to the targeted device's native gate-set while obeying all restrictions imposed by the device, e.g., limitations on the interaction of qubits.

For classical compilation, guidelines and best-practices have been developed over the previous decades such that even end-users without a background in computer science can compile and execute their programs. In quantum compilation, however, this is not yet the case and classical compilation schemes cannot be adopted in a one-to-one fashion. The reason for that lies within the constraints (such as the restricted connectivity of quantum devices) to be satisfied by a compiled quantum circuit compared to classical software. Since comparable guidelines and best-practices have not yet been developed for quantum circuit compilation, expert knowledge is required and especially end-users without a background in quantum computing are easily overwhelmed with a flood of different qubit technologies, devices, compilers, and (quite frequently, poorly documented) settings to choose from. Without actionable advice, it is extremely difficult for an end-user to decide on a combination of all these options for a correspondingly considered application.

In this work, we treat the problem as a statistical classification task and explore the utilization of supervised machine learning techniques to optimize the compilation of quantum circuits. We propose a framework that, given a quantum circuit, predicts the best combination of these options and, by that, *automatically* decides for end-users which qubit technology, which specific device, which compiler, and which corresponding settings to choose for realizing their applications. By this, a similar kind of comfort is provided to the end-users of quantum computers as is taken for granted in the classical world.

In order to demonstrate the effectiveness of the proposed methodology, we trained a machine learning classifier on 3000 training circuits (taken from the MQT Bench library [11]) considering two qubit technologies, five different devices, two compilers, and six corresponding settings. The resulting classifier, which is publicly available along with the proposed framework, is shown to yield the best possible combination of compilation options in more than three quarters of all unseen test circuits. For more than 95% of the circuits, a combination of compilation options within the top-three is determined—with the median compilation time being reduced by more than one order of magnitude compared to manually compiling for all possible combinations of compilation options and choosing the best result. Moreover, rather than functioning as a black-box, the underlying model additionally provides means to extract explicit knowledge from the classifier—potentially guiding further works towards exploiting the full potential of machine learning in this domain and, also, to quickly verify whether a machine learning algorithm is reasonably trained. The corresponding framework and the pre-trained classifier are publicly available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT).

The rest of this work is structured as follows: In Section II, we describe the process of determining a good combination of compilation options and review the related work. Based on that, Section III details the methodology to predict good combinations of options. Section IV describes the resulting framework and summarizes our experimental evaluations. Section V discusses how explicit knowledge can be extracted from the proposed methodology and how changes in the underlying data can be incorporated before Section VI concludes this work.

II. Considered Problem: Determining Good Compilation Options

In this section, we review the spectrum of options to choose from when realizing an application on an actual quantum computer, discuss the resulting dilemma for end-users, and the related work available on this topic so far.

A. Motivation

Due to its promising applications and the steady evolution of the corresponding technologies, quantum computing is no longer a niche topic. Researchers in application domains very different from quantum computing seek to utilize this technology as a tool to solve their domain-specific problems in a more efficient fashion. As a consequence, quantum computers are no longer exclusively used by physicists or computer scientists, but by an interdisciplinary range of end-users.

After a quantum algorithm for solving a particular problem from an application domain has been developed in terms of a quantum circuit, the end-user is confronted with a flood of possible options and design decisions:

- Which *qubit technology*, e.g., superconducting qubits or ion traps, is best suited for the application at hand?
- Which particular *device*, e.g., from IBM or Rigetti, fits the quantum algorithm best?
- Which *compiler*, such as IBM's Qiskit or Quantinuum's TKET, is most efficient for compiling the algorithm to the respective device?
- Which *settings and optimizations* offered by the compilers, such as different levels of optimization, are adequate for the problem considered?

Figuring out good combinations of options (according to some evaluation metric) for a particular application is a highly non-trivial task due to several factors:

- Different qubit technologies have their own advantages and disadvantages, e.g., devices based on ion-traps have an all-to-all connectivity but suffer from slow gate execution times, while devices based on superconducting qubits have limited qubit connectivity but fast gate execution times [12].
- Individual devices greatly vary in their characteristics such as qubit count, error rates, coherence times, qubit connectivity, and gate execution times.
- Various compilers have been proposed by industry and academia, i.e., in [1]–[5], [13]–[15]—each of which is particularly suited for certain classes of circuits and architectures.
- Compiler settings and optimizations, quite frequently, are hardly or insufficiently documented.
- On top of all that, the domain of quantum computing is fast-paced and constantly changing—quickly and frequently redefining the state of the art.

This leaves end-users without expert knowledge faced with more options than they can feasibly explore. Since quantum computing is still in its infancy, this diversity of options is only going to increase in the future. Thus, automated tools for predicting good combinations of options are absolutely necessary in order to provide the same kind of comfort that is taken for granted in the classical world today. Otherwise, we might end up in a situation where we have powerful quantum computers and tools, but only a selected group of people know how to use them.

B. Related Work

Decades of work on classical compilers have led to many sophisticated compiler optimization techniques. *Autotuning* (which describes the process of trying different compilation parameters and comparing their result against some metric) and machine learning (overviews are given in, e.g., [16], [17]) have shown promising results and established themselves as state-of-the-art techniques in this domain. Even combined approaches of those two techniques are explored in [18], where a machine learning model is used not to directly determine optimal compiler settings but to identify promising areas of the optimization space. Additionally, reinforcement learning gained a lot of attention in recent years with promising results, e.g., in [19]–[21]—all targeting the LLVM [22] compiler.

In quantum compilation, a domain that is still in its infancy, the applied techniques are not that highly developed. Nevertheless, first works towards this direction have already been proposed nearly a decade ago in the form of hardware resource estimates to reliably execute various quantum algorithms [23] to provide guidance to end-users. Until today,

resource estimation has still been an open research question. Microsoft recently proposed the Azure Quantum Resource Estimator [24] which is publicly available in their SDK. Furthermore, a application-specific resource estimator in the domain of derivative pricing has been developed [25] and can be used to estimate the minimal quantum computing resources needed to achieve a real-world quantum advantage in this domain. However, resource estimation often requires end-users to manually provide a lot of information on particular hardware aspects such as gate times and fidelities.

A different, but related, approach is to evaluate whether existing architectures and compilers are capable of executing a given quantum circuit. One example is the *NISQ Analyzer* proposed in [26] which allows one to determine architectures that are expected to be capable of reliably executing a given circuit based on their number of qubits and a measure of their maximum supported circuit depth. Although this solution requires less manual input, it still provides no advice on which compiler (settings) to use and also on which of the determined architectures to pick.

In the recent past, many tools for comparing the performance of different quantum circuit compilers for different devices and/or compilation options have been proposed [27]–[30]. However, in all these approaches, each input circuit is simply compiled for every possible combination of compilation options and the best circuit according to some metric is reported, i.e., the best option is determined in a brute-force fashion. While this provides the basis for interesting case studies, such an approach cannot be feasibly employed in practical situations due to the sheer amount of time it takes to try out all options on every invocation with a particular circuit—a situation which is only going to get worse with the ongoing increase in options.

In addition to the above approaches, machine learning methods have been proposed to tackle certain quantum compilation challenges by learning from data. In [31], the influence of several circuit characteristics on the quality of the circuit execution results is studied using *Multi-Criteria Decision Analysis* methods and machine learning approaches to optimize such. Similarly, the approach proposed in [32] tries to learn an estimate of circuit fidelity for specific devices using the graph structure of quantum circuits. Complementarily, a reinforcement learning-based approach that models quantum compilation as a *Markov Decision Process* with the goal of learning optimal compilation sequences has recently been proposed in [33].

Overall, there are many interesting activities happening in this area. Nevertheless, to the best of the authors' knowledge, there is no automatic solution to determine the best combination of quantum compilation options without explicitly executing and examining all of them—which might be feasible for conducting a case study on a handful of devices and compilers but is certainly not scalable enough to support end-users in realizing their real-world applications. In this work, we apply similar techniques as in classical compiler optimization targeting this problem.

III. Proposed Solution: Predicting Good Compilation Options

As discussed in the previous section, naively executing and evaluating all possible combinations of compilation options on demand quickly becomes infeasible due to the large number of possible options. Even if an end-user was able to utilize all those different SDKs to compile the specific quantum circuit for all computers, it would have been a very time-consuming and costly endeavor. Thus, in this work, we propose a methodology that allows one to *predict* good combinations of options without the need for explicit compilation. To this end, the problem is interpreted as a statistical *classification* task which constitutes a prime task for *supervised machine learning* algorithms. In the following, each individual aspect of the proposed methodology is described and illustrated exemplary. Based on that, Section IV then covers how this results in a corresponding framework for the prediction of good combinations of options and evaluates the benefits for end-users.

A. Compilation Options and Compilation Pipeline

Given certain qubit technologies with a set of respective devices and certain compilers with various settings, the search space of all compilation options is spanned by all the possible combinations of technologies, devices, compilers, and settings. Therefore, a compilation pipeline needs to be set up to realize each combination of compilation options as a basis for the proposed methodology. This poses a significant challenge as a result of the vastly different interfaces and usability levels of existing SDKs. Overall, this results in a decision tree structure, where each path from the root node to a leaf node represents one possible combination of compilation options.

Example 1. *For illustration purposes assume that in the following, the end-user has to decide between four devices based on superconducting qubits (with* 8*,* 27*,* 80*, and* 127 *qubits, respectively) and a single ion trap-based one (with* 11 *qubits). Additionally, assume that IBM's Qiskit [1] and Quantinuum's TKET [2] are available as state-of-the-art representatives for compilers. Last but not least, assume that in the case of Qiskit four different optimization levels (called* $O0$ *to* $O3$*) are considered, while in the case of TKET two different qubit placement algorithms (called* Line Placement *and* Graph Placement*) can be used for the compilation.*

Both considered compilers are capable of compiling a quantum circuit for all the considered devices. Therefore, the corresponding search space for compilation options is structured as illustrated in Fig. 1—comprising a total of 30 *different combinations of compilation options.*

B. Evaluation Metric

Based on the constructed search space, the definition of an *evaluation metric*—determining whether a combination of compilation options is good for a certain quantum circuit—is an essential part of the proposed framework. All further steps aim to optimize the prediction quality of the resulting model according to *this* evaluation metric.

In principle, this evaluation metric can be designed to be arbitrarily complex, e.g., factoring in actual costs of executing quantum circuits on the respective platform or availability

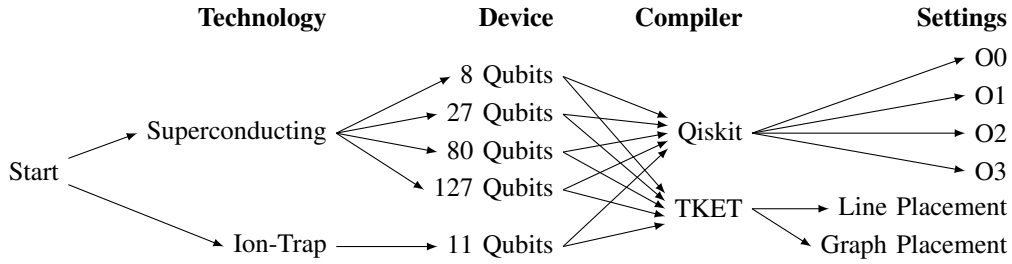


Fig. 1. Search space of compilation options for the setup described in Example 1.

limitations for certain devices. However, any suitable evaluation metric should at least consider the characteristics of the compiled quantum circuit and the respective device. Thus, hardware information for each of the devices (such as qubit coherence times and gate fidelities) needs to be gathered. For some of these devices, this information might not be publicly available from the respective hardware vendors and, hence, needs to be estimated from comparable architectures, previous records, or insight knowledge.

Example 2. *In the following, we employ an evaluation metric that considers two aspects:*

1) *The device chosen in the combination of compilation options has to have at least as many physical qubits as the compiled circuit (or else the circuit would not be executable at all). If this is not the case, the worst-possible score is assigned to this combination of compilation options.*
2) *If the first criterion is satisfied, each combination of compilation options is assigned an evaluation score defined by the formula:*

$$eval\ score = \prod_{i=1}^{|G|} gate\ fid(g_i) \prod_{j=1}^{m} readout\ fid(q_j),$$

where $gate\ fid(g_i)$ *denotes the expected fidelity of gate* g_i *on its corresponding qubit(s),* $readout\ fid(q_j)$ *denotes the expected fidelity of a measurement operation* q_j *on its corresponding qubit,* $|G|$ *denotes the number of gates in the compiled circuit and* m *denotes the number of measurements.*

This evaluation metric measures the expected fidelity of the compiled circuit due to noise, i.e., the probability of the circuit working as expected—higher values meaning less noise and, hence, better results.

C. Generation of Training Data

As with any machine learning algorithm, the availability of sufficient training data is critical for the performance of the resulting model. In this regard, the biggest challenge is to gather sufficiently many quantum circuits to start the generation of training circuits—even more so, ones that are representative for the diversity of quantum computing applications.

Once a sufficiently large set of circuits has been gathered, *all* possible combinations of compilation options are executed and evaluated using the metric from Section III-B for each of the training circuits using the pipeline from Section III-A. Then, the best combination of compilation options is stored as the classification label for this particular training circuit.

Example 3. *In order to generate training data based on the compilation options from Example 1 and the evaluation metric from Example 2, the benchmark library MQT Bench [11] has been used. It provides a large selection of benchmarks covering all kinds of quantum computing applications on various abstraction levels. Here,* 3000 *benchmarks have been utilized from the "target-independent" level from* 2 *to* 127 *qubits. Using a timeout of* 300*s for each combination of compilation options (as a trade-off between execution time and the number of successful compilations), classification labels have been determined as illustrated in Fig. 2. For each training circuit, all suitable combinations of compilation options are executed once to determine their corresponding evaluation scores. Subsequently, the combination of compilation options leading to the highest evaluation score is the best combination of compilation options and thus the classification label for that training circuit.*

After generating the labeled training data—consisting of the initial quantum circuits and the best combinations of compilation options—the training circuits are transformed into *feature vectors* to be suitable for training a classifier. On the basis of that, the classifier is able to predict good combinations of compilation options for quantum circuits not used as training data (referred to as *unseen* test circuits in the following) based on their features. As for the evaluation metric, the features extracted from the input quantum circuit can be designed to be arbitrarily complex.

Example 4. *After determining the correct classification label, the feature vector of each circuit is created. To this end, the number of qubits, the depth of the circuit, and the number of gates for each gate type according to the* OpenQASM 2.0 *specification [34] are used as features. Additionally, the following five composite features (adapted from [35]) are used:*

- Program Communication*: Metric to measure the average degree of interaction for all qubits. A value of* 1 *indicates that each qubit interacts at least once with all other qubits.*
- Critical Depth*: Metric to measure how many of all multi-qubit gates are on the longest path (defining the depth of a quantum circuit). A value of* 1 *indicates that all multi-qubit gates are on the longest path.*

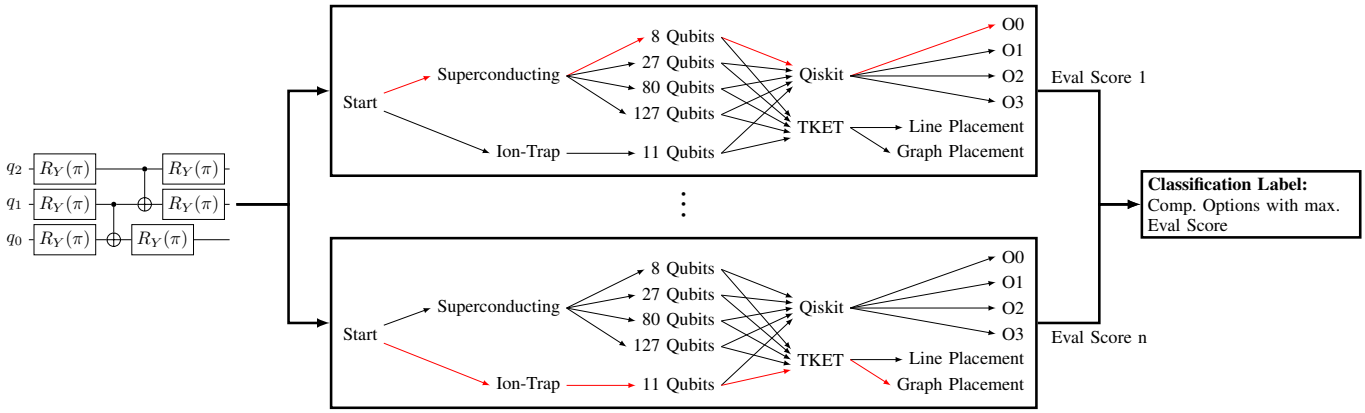


Fig. 2. Generation of a training circuit from an example circuit.

- Entanglement Ratio*: Metric to measure how many gates in a quantum circuit are multi-qubit gates. A value of* 1 *indicates that the quantum circuit consists of only multi-qubit gates.*
- Parallelism*: Metric to measure how much parallelization within the circuit is possible due to simultaneous gate execution. A value of* 1 *describes large parallelization.*
- Liveness*: Metric to measure how often the qubits are idling and waiting for their next gate execution. A value of* 1 *describes a circuit in which there is a gate execution on each qubit at each time step.*

In order to reduce the dimensionality of the resulting training circuits, the features that are zero for all training circuits are discarded. This resulted in 31 *features for each of the training circuits.*

D. Solving the Classification Task

Eventually, the goal is to predict good combinations of compilation options for unseen test quantum circuits without executing and evaluating all possible combinations. This classification task is perfectly suited for supervised machine learning classifiers. To this end, a model representing the complex factors that influence the selection of compilation options can be trained without giving explicit guidelines.

IV. Resulting Prediction Framework and its Performance

The methodology proposed above has been implemented as a proof-of-concept, open-source Python package and is available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT). It comprises three main functionalities:

1) Real-time prediction of good compilation options based on a pre-trained classifier,
2) running the compilation with the predicted best combination of compilation options itself, and,
3) the adaption and extension of the training pipeline to create custom prediction models.

To this end, scikit-learn [36] is used to train the underlying machine learning model. The resulting framework allows for automatic prediction of good combinations of compilation options out of all considered possibilities. As a result, the tedious task of choosing a combination of compilation options is shifted from the end-user without quantum computing expertise to an automatic framework that has this domain knowledge embedded. In the following, we describe the setup of the prediction framework and, then, comprehensively evaluate it in order to demonstrate the feasibility and effectiveness of the proposed methodology.

A. Setting Up the Prediction Framework

In order to instantiate the framework as described above, it needs to be set up accordingly. To this end, the following steps need to be conducted:

1) All qubit technologies, devices, compilers, and settings that should be considered need to be provided. Finally, this defines the set of possible combinations of compilation options.
2) The corresponding SDKs and software packages need to be set up and stitched together.
3) All information about the devices which shall be considered in the evaluation of the goodness of compilation options needs to be collected and, afterwards, incorporated into an evaluation metric.
4) Finally, the classifier must be trained using a sufficient number of suitable training circuits.

Note that using this setup procedure, even future developments can easily be considered as it only requires the incorporation and/or adjustment of the respective instantiation.

In the following and for the purpose of evaluation, we considered the following setup (which was described throughout all previous examples):

- Qubit Technology: superconducting and ion-trap-based qubits
- Devices: Archictures with 8, 27, 80, and 127 qubits for superconducting and 11 qubits for the ion-trap based qubit technology
- Compiler: IBM's Qiskit (version 0.39.2) and Quantinuum's TKET (version 1.9.0)
- Compiler settings: 4 optimization levels (Qiskit) and 2 placement strategies (TKET)

Overall, this results in a total of 30 different combinations of compilation options (as described in Example 1). As evaluation metric, we used the information and the corresponding calculation as previously described in Example 2. All circuits used for training are taken from the MQT Bench library [11] (version 0.2.2). More precisely, all circuits on the "target-independent" level from 2 to 127 qubits have been used—resulting in 3000 training data samples.

Then, each of these circuits is exhaustively compiled with every possible combination of options and subsequently evaluated according to the desired evaluation metric. Eventually, the best combination of options is recorded as the label for the training data[1]. Due to the independence of the individual generation jobs (regarding the evaluation metric as well as the combination of options), the training data generation can be easily parallelized across all available resources. The generation of training data for this particular instantiation took around 100h on a 16-threaded Intel Xeon W-1370P with 3.60 GHz and 128 GB RAM, and resulted in a total of 38672 compiled and evaluated circuits (covering all possible combinations of compilation options)—making the 3000 training circuits ready for use in any machine learning algorithm. In our evaluation, we used a 70%/30% train-test-split which resulted in 2100 training samples and 900 test samples. In the following, we summarize and discuss our evaluations on seven different supervised machine learning classifiers applied to the generated data.

B. Resulting Performance

Using the instantiation described above, we evaluated the performance of various supervised machine learning classifiers—all instantiated and trained with grid-searched and 5-fold cross-validated parameter values in order of minutes:

- *Random Forest* [37]
- *Gradient Boosting* [38]
- *Decision Tree* [39]
- *Nearest Neighbor* [40]
- *Multilayer Perceptron* [41]
- *Support Vector Machine* [42]
- *Naive Bayes* [43]

For an overview of today's use of those techniques, see [44].

To this end, we considered the 900 unseen test quantum circuits and predicted the best combination of compilation options for them. To compare whether predictions actually yielded the best possible options (or not), the evaluation scores produced as part of the training data generation were persistently stored and used as a ground truth. Subsequently, the resulting predictions have been ranked based on this ground truth. That is, any prediction that actually constituted the best combination of compilation options got assigned rank 1, any prediction that constituted the second-best combination of compilation options got assigned rank 2, etc. With a total of 30 combinations of compilation options, the worst prediction accordingly got assigned rank 30.

[1]In many cases, it is beneficial to not discard the computed evaluation scores and the compiled circuits right away, but rather store them persistently. This allows one to validate the performance of a trained classifier, as well as quickly re-evaluate and re-label test samples after changes to the evaluation metric or the addition of new options.

TABLE I
COMPARISON OF DIFFERENT CLASSIFIERS.

Classifier	Accuracy	Top3	Worst Rank	Best Score Diff.
Random Forest	0.77	0.96	23	-0.0021±0.0138
Gradient Boosting	0.76	0.95	23	-0.0022±0.0138
Decision Tree	0.73	0.92	12	-0.0038±0.0202
Nearest Neighbor	0.71	0.94	23	-0.0036±0.0219
Multilayer Perceptron	0.66	0.89	23	-0.0047±0.0220
Support Vector Machine	0.62	0.76	21	-0.0083±0.0365
Naive Bayes	0.33	0.55	19	-0.0184±0.0506

The results obtained correspondingly for all trained classifiers are summarized in Fig. 3. More precisely, the histograms show the relative frequency of the rankings for all the predictions made, ranging from the Random Forest classifier shown in Fig. 3a to the Naive Bayes classifier shown in Fig. 3g. The performance of all classifiers is assessed by four measures denoted in Table I:

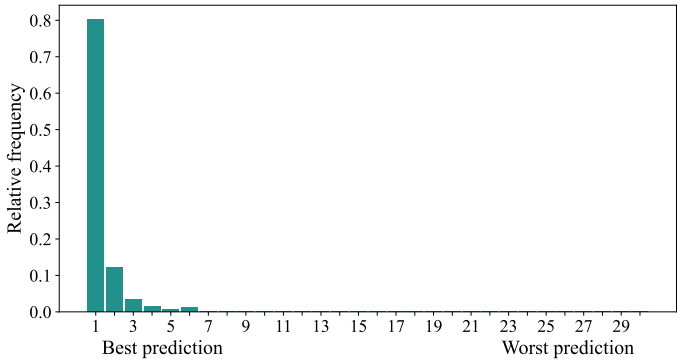
1) Accuracy: Relative frequency of predicting the best possible combination of compilation options.
2) Top3: Relative frequency of predicting one of the top-three combinations of compilation options.
3) Worst Rank: Worst predicted rank for any of the test samples (out of 30).
4) Best Score Diff.: Mean *absolute* evaluation score difference (and standard deviation) compared to the performance of the best combination of compilation options.

In this instantiation, the Random Forest classifier shown in Fig. 3a with the following parameters led to the best performance:

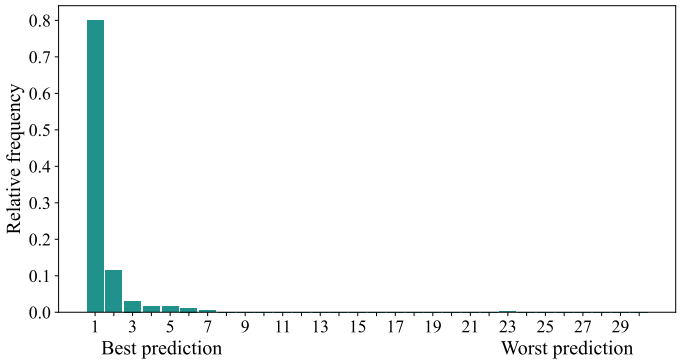
- Number of Decision Trees: 100
- Maximal depth = 26
- Minimal samples per leaf = 2
- Minimal samples per split = 2

It clearly confirms the quality of the predictions produced by the proposed prediction framework. In fact, for more than three quarters of all unseen test circuits, the *best* combination of compilation options has been determined by the prediction framework. Moreover, for more than 95% of the circuits, a combination of compilation options within the top-three is determined while the average absolute difference to the best performing options is around 0.2% in expected fidelity. Consequently, the Random Forest classifier yields the best or at least a very good prediction of the combination of compilation options in all test cases. Considering that all these predictions are made in real-time and must only be compiled once, this is a tremendous improvement compared to the state of the art where the end-users have to manually compile their circuit for all combination of compilation options to determine the best one—leading to a median runtime improvement of more than one order of magnitude.

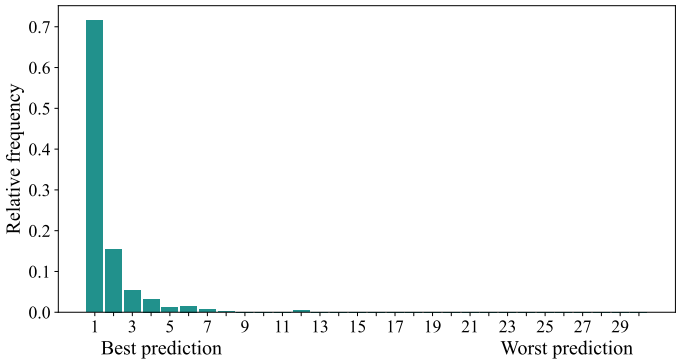
Example 5. *Consider the case that a user wants to find the best combination of compilation options for a* Deutsch-Jozsa *algorithm [45] instance with 7 qubits (also taken from the MQT Bench library [11]) with the setup described in Section IV-A. For that, the respective quantum circuit must be compiled for all possible combinations leading to 30 compilations, where each resulting compiled circuit needs to be*



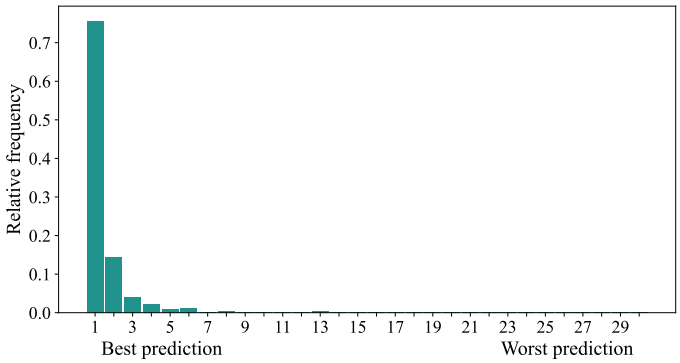
(a) Random Forest



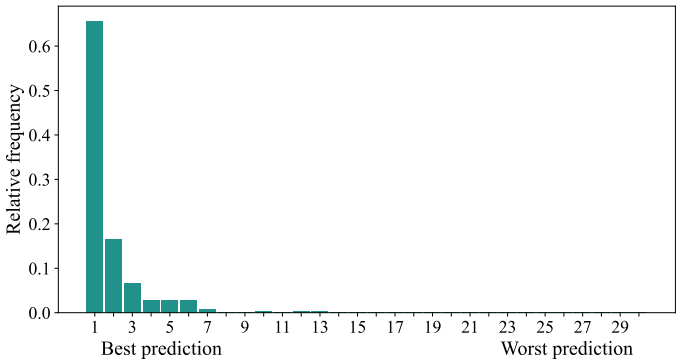
(b) Gradient Boosting



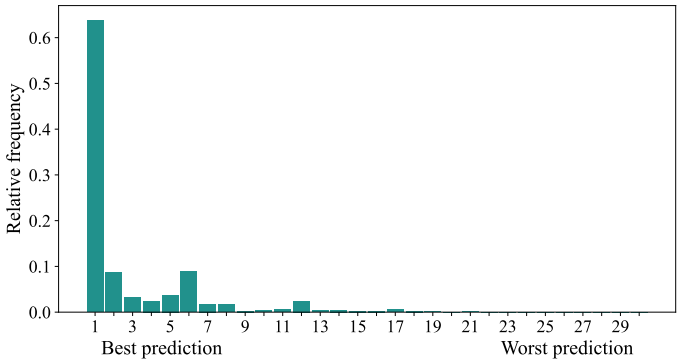
(c) Decision Tree



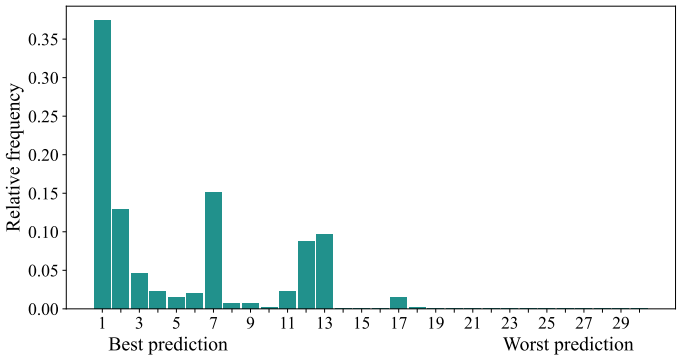
(d) Nearest Neighbor



(e) Multilayer Perceptron



(f) Support Vector Machine



(g) Naive Bayes

Fig. 3. Comparison of supervised machine learning classifiers.

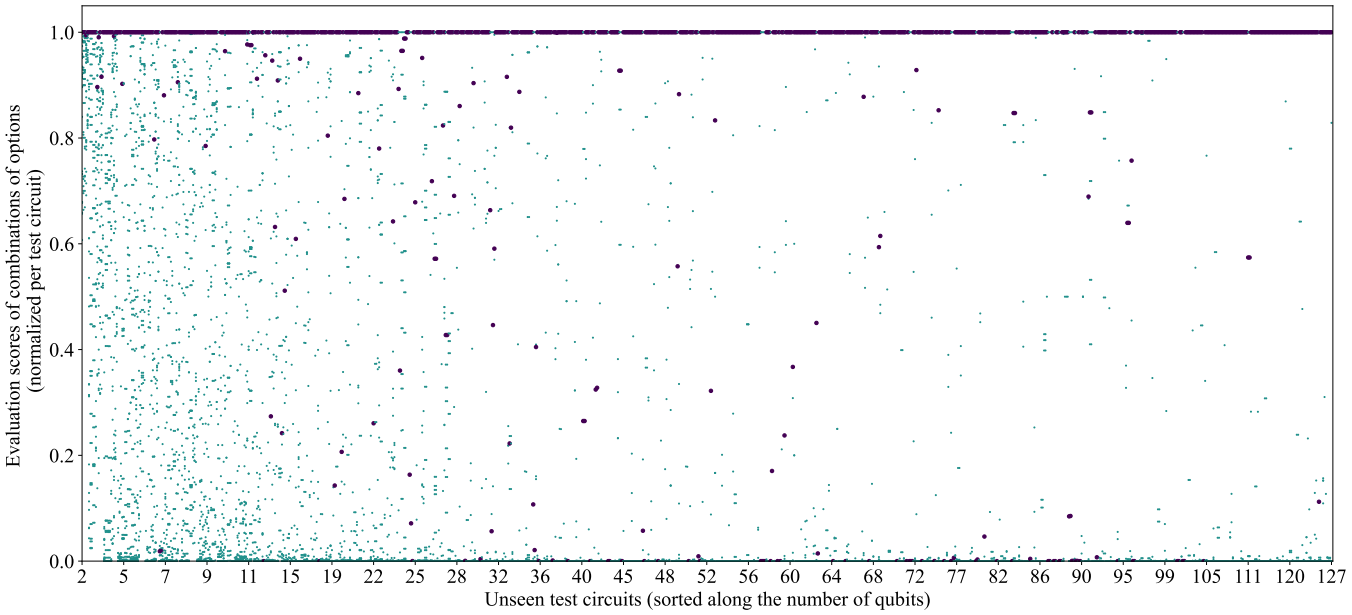


Fig. 4. Detailed evaluation of all combinations of compilation options for all unseen test circuits. All unseen test circuits are sorted by their qubit number which is indicated on the x-axis and, for most qubit numbers, multiple test circuits are evaluated. For each test circuit, all executed and evaluated combinations of compilation options are indicated by green dots (●) while the predicted combination of compilation options is indicated by a purple dot (●). Due to the increased availability of quantum circuits with small numbers of qubits, the test circuits are not evenly spaced out with regard to their number of qubits. Since each device comes with a qubit limit (e.g., 8, 11, 27, 80, and 127 qubits), the number of executed combinations of compilation options (and corresponding green dots) decreases with a growing number of qubits. For the majority of the test cases, the predicted combination of compilation options returns the best results—illustrating the performance of the proposed framework.

additionally evaluated using the evaluation metric described in Example 2—taking roughly a minute. In comparison, using the proposed framework, only a prediction of the best combination of compilation options and the respective single compilation is necessary. This is conducted in less than a second and, in this case, also leads to the best combination of compilation options while the calculation time is reduced by more than one order of magnitude.

This shows the potential of machine learning-based optimization for quantum circuit compilation—similar to the enhancement these techniques brought to the domain of classical compiler optimization. The Random Forest classifier is provided as a pre-trained model within our Python package. Next, the performance of this classifier is examined to underline the importance of selecting good combinations of compilation options.

C. Importance of Compilation Options

A more detailed insight of the complete results of the Random Forest classifier is given in Fig. 4, which shows the evaluation scores of all possible combinations of compilation options for all unseen test circuits. Here, each green dot (●) corresponds to one possible combination of compilation options while the predicted result is indicated by a purple dot (●). The resulting evaluation score, i.e., the expected fidelity of the circuit execution, is normalized and plotted on the y-axis (higher is better), such that the normalized evaluation score of the best combination of compilation options per test circuit is assigned a value of 1.0.

The results clearly demonstrate the significant impact of the chosen combination of compilation options on the expected performance/quality of the considered circuit. In other words: Whether a good or bad combination of compilation options is chosen frequently makes the difference between a reliable execution or one that does not work at all[2]. Consequently, end-users may have a brilliant quantum circuit design, but its performance can still be spoiled by choosing a bad combination of compilation options. This emphasizes the importance of providing end-users with good predictions on the compilation options. The results summarized in Fig. 4 again confirm that the proposed framework can deliver on that (in the vast majority of cases, the purple dot, i.e., the predicted combination of compilation options, is on the top of the spectrum).

V. Discussion

In general, machine learning techniques have two major drawbacks. The first is their black-box character and a lack of explainability of why a certain prediction has been made. The second one is the effort spent on the model training and necessary preparations for that and, thus, the adaptability to change. In this section, we discuss both drawbacks and propose approaches on how to mitigate and tackle the respective challenges.

[2]Of course, the scores and ranking of combinations of compilation options highly depend on the chosen evaluation metric. However, the expected fidelity used in this evaluation has proven to be suitable for judging the expected performance of a quantum circuit. In addition to that, as discussed in Section IV-A, the setting used here is just a representative and can be adjusted to correspondingly reflect other setups.

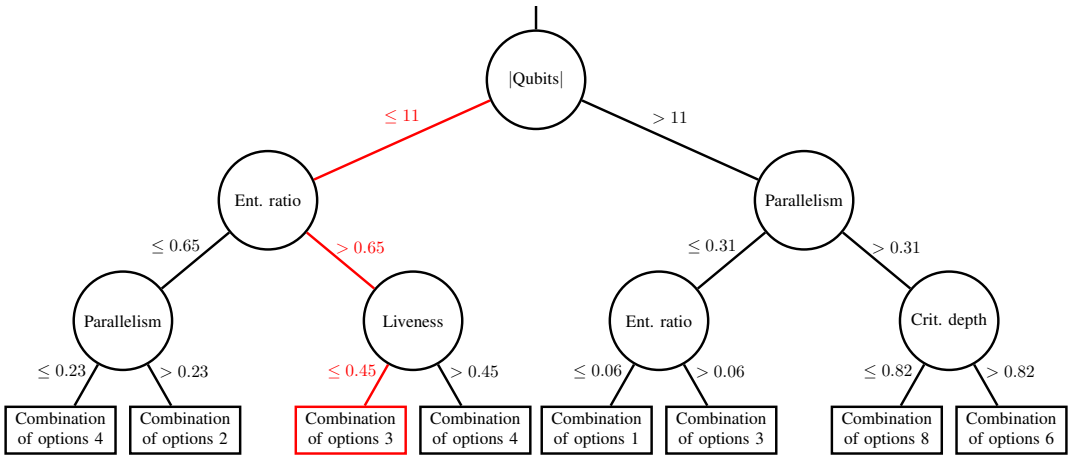


Fig. 5. Simplified decision tree classifier.

A. Knowledge Extraction

Bringing light to opaque black-box machine learning techniques and learning more about their working mechanisms is a whole research area on its own (with overviews given in, e.g., [46], [47]). Depending on the machine learning classifier itself, the methods chosen to gain insight also vary.

In the following, insights into the best performing classifier in the above evaluation, the Random Forest classifier, are given and discussed to extract explicit knowledge. While it is particularly straightforward to get insights into this type of classifier, there are similar methods to extract information for other classifiers with overviews given in the provided references. This can be used to guide further work towards exploring the potential of machine learning in predicting good combinations of compilation options and, additionally, to quickly verify the reasonableness of the trained classifier.

The Random Forest classifier is composed of an ensemble of Decision Trees (100 in the evaluated scenario) that perform a majority vote for the prediction classification label and provides a rather simple method to gain insights called *feature importance*. The feature importance describes the normalized influence of each feature of the trained model and is defined as the reduction of the misclassification probability (also called *gini impurity*) averaged over all comprised Decision Trees.

Example 6. *Fig. 5 shows a simplified illustration of a Decision Tree classifier for the scenario of determining good compilation options. Each node in such a Decision Tree corresponds to a decision depending on some of the features/characteristics of the circuit to be classified. A prediction follows a certain path from the root node according to the decisions at each node until a terminal node is reached—containing the predicted label/combination of compilation options. Using this classifier, for a circuit with* 10 *qubits that has an entanglement ratio of* 0.74 *and a liveness of* 0.32*, "Combination of options 3" (highlighted in red) would be predicted as its classification label. For some circuits, the resulting prediction might not be the correct and best classification label and, hence, they are misclassified based on the decision nodes and their thresholds. With an ensemble of* 100 *different Decision Trees, the influence of each feature on these misclassification rates can be determined and, by that, the reduction of the gini impurity.*

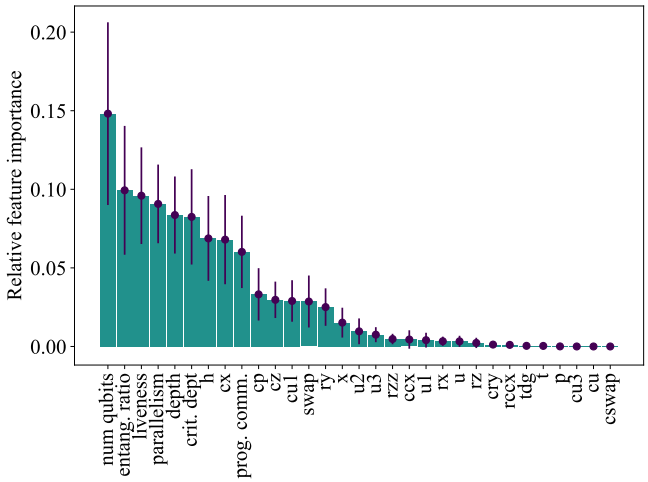


Fig. 6. Feature importance: Normalized mean reduction of the gini impurity including its standard deviation for each feature.

The feature importance for the trained Random Forest classifier is summarized and denoted in decreasing order in Fig. 6 with significant differences in the influence each feature has. While 14 of the 31 features have a considerable influence on the trained model (all features up to the *ry* gate count with a feature importance of $\geq 2.5\%$), the remaining 17 features are mostly negligible (with a feature importance of $< 2.5\%$). When focusing on the influential ones, the number of qubits feature is standing out with the highest importance. This is a strong testimony of the successful training of the underlying model and can be explained since no execution is possible at all if the device's number of qubits is smaller than needed. Furthermore, the rather complex features (such as those adapted from [35]) are considerably more influential than most of the plain gate counts. This indicates that those features characterize a quantum circuit better—with a few exemptions: the *h*, *cx*, *cp*, *cz*, *cu1*, *swap*, and *ry* gates.

Since the multi-qubit gates (*cx*, *cp*, *cz*, *cu1*, and *swap*) heavily influence the overhead during mapping, their high influence is expected. Additionally, the *cx*, *cz*, and *ry* gates are native to some devices, resulting in significantly fewer native gates after compilation. Similarly, the *h* gate is more efficiently compiled to certain native gate-sets leading to higher influence.

While it is always possible to assume the reason behind some feature's impact and deduct rules of thumb (such as "*quantum circuits with a high program communication should be mapped to ion-trap devices due to their full connectivity*" or "*circuits with a large* cx *gate count should be mapped to devices where it is a native gate*"), it is far too complex to factor in and add weight to a large number of features—even with only 30 different combinations of compilation options and 31 features. This underlines the importance and potential of machine learning approaches that aim for the same success they brought to the domain of classical compiler optimization.

Nevertheless, the extracted knowledge in the form of feature importance is helpful in two ways:

- It indicates what kind of features are helpful to characterize a quantum circuit and underlines the importance of thorough feature engineering.
- It provides insight into the trained model, allowing one to quickly verify whether it is reasonable at all.

B. Adaptability to Change

The second drawback of machine learning algorithms is the effort needed for the generation of training data and the model training itself—especially, since the model has to be re-trained whenever the evaluation metric or the available options are updated.

Quantum computers are calibrated frequently to ensure that they operate at their lowest error rate. Thus, the noise characteristics considered in the proposed evaluation metric could change from calibration to calibration.

In addition to the characteristics of existing devices that change over time, the number of devices and their underlying technologies, compilers, and respective options is steadily increasing. Naturally, adding more options increases the time to generate training data, while also potentially requiring an update to the evaluation scores and the classification label, since a new option might lead to better results than were possible before.

It is neither feasible nor realistic to always re-generate all the training data and re-train the model from scratch. Calibration might be performed daily, while the generation of the training data on our system took roughly five days (while the training itself takes only minutes). Obviously, large *High Performance Computing (HPC)* systems could be used to speed up generation and training, but the amount of required resources might not be worth the price.

Another approach is to re-use the training data of the previously trained models as much as possible. As described in Section IV-A, all compiled circuits used for the training data generation are persistently stored in the proposed framework. Thus, all these compiled circuits can be utilized whenever new calibration data is available, since they must only be re-evaluated and not re-generated. The updated calibration data just assigns a new value to each compiled circuit to determine the classification label. Similarly, the same previous training can be used, even without any re-evaluation, whenever a new combination of compilation options is added—only the newly added compilation options must be applied to compile each training circuit before the classification label is determined.

Nevertheless, in both cases, the model itself must be re-trained based on adjusted training data. There are different approaches to avoid the necessity of re-training the model from scratch. So-called *warmstart* approaches can be utilized where parameters of an existing model are used as the starting point for a new model (incorporating, for example, the latest calibration data) as, e.g., discussed in [48] for neural networks.

VI. Conclusion

In this work, we proposed a methodology that allows end-users from domains different to quantum computing to realize their applications more easily on actual hardware. This is accomplished by shifting the tedious selection of good compilation options away from the end-user and embedding the necessary expert knowledge in a prediction framework that takes a quantum circuit as input and predicts the best combination of various qubit technologies, devices, compilers, and corresponding settings. Experimental evaluations show that, out of 30 different combinations of compilation options, the most powerful classifier is capable of predicting the best combination of compilation options for more than three quarters of all unseen test circuits. For more than 95% of the circuits, a combination of compilation options within the top-three is determined—while the median compilation time is reduced by more than one order of magnitude compared to manually compiling for all possible combinations of compilation options and choosing the best result. In addition to the time savings, the underlying model provides insight on why certain decisions have been made—allowing end-users to build up expertise from the predicted results. Furthermore, the proposed methodology can easily be adapted and extended to future qubit technologies, devices, compilers, and compiler settings. The corresponding framework and the pre-trained classifier are publicly available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT). To the best of our knowledge, this work constitutes the first step towards using machine learning for quantum compilation—aiming for a similar success as achieved in classical compilation leading and, by that, simplifying and accelerating the adoption of quantum computing to solve problems from various application domains.

VII. Acknowledgements

This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 101001318), was part of the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus, and has been supported by the BMWK on the basis of a decision by the German Bundestag through project QuaST, as well as by the BMK, BMDW, and the State of Upper Austria in the frame of the COMET program (managed by the FFG).

References

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

[2] S. Sivarajah *et al.*, "T|ket⟩: A retargetable compiler for NISQ devices," *Quantum Science and Technology*, 2020.

[3] C. Developers, *Cirq*, version v0.12.0, See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors, 2021.

[4] V. Bergholm *et al.*, *PennyLane: Automatic differentiation of hybrid quantum-classical computations*, 2020. arXiv: 1811.04968.

[5] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, 2016. arXiv: 1608.03355.

[6] A. Peruzzo *et al.*, "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, 2014.

[7] N. Stamatopoulos *et al.*, "Option pricing using quantum computers," *Quantum*, 2020.

[8] S. Harwood *et al.*, "Formulating and Solving Routing Problems on Quantum Computers," *IEEE Transactions on Quantum Engineering*, 2021.

[9] C. Zoufal, A. Lucchi, and S. Woerner, "Quantum Generative Adversarial Networks for learning and loading random distributions," *npj Quantum Information*, 2019.

[10] N. Quetschlich, L. Burgholzer, and R. Wille, "Towards an Automated Framework for Realizing Quantum Computing Solutions," in *Int'l Symp. on Multi-Valued Logic*, 2023.

[11] N. Quetschlich, L. Burgholzer, and R. Wille, *MQT Bench: Benchmarking software and design automation tools for quantum computing*, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/, 2022. arXiv: 2204.13719.

[12] N. M. Linke *et al.*, "Experimental comparison of two quantum computing architectures," *Proceedings of the National Academy of Sciences*, 2017.

[13] M. Amy and V. Gheorghiu, "Staq—A full-stack quantum processing toolkit," *Quantum Science and Technology*, 2020.

[14] T. Häner *et al.*, "A software methodology for compiling quantum programs," *Quantum Science and Technology*, 2018.

[15] A. S. Green *et al.*, "Quipper: A scalable quantum programming language," *ACM SIGPLAN Notices*, 2013.

[16] H. Leather and C. Cummins, "Machine learning in compilers: Past, present and future," in *Forum for Specification and Design Languages*, 2020.

[17] A. H. Ashouri *et al.*, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, 2018.

[18] F. Agakov *et al.*, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization*, 2006.

[19] M. Trofin *et al.*, *Mlgo: A machine learning guided compiler optimizations framework*, 2021. arXiv: 2101.04808.

[20] Q. Huang *et al.*, *Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning*, 2020. arXiv: 2003.00671.

[21] A. Haj-Ali *et al.*, *Neurovectorizer: End-to-end vectorization with deep reinforcement learning*, 2019. arXiv: 1909.13639.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004.

[23] M. Suchara *et al.*, "QuRE: The Quantum Resource Estimator toolbox," in *Int'l Conf. on Comp. Design*, 2013.

[24] M. E. Beverland *et al.*, *Assessing requirements to scale to practical quantum advantage*, 2022. arXiv: 2211.07629.

[25] S. Chakrabarti *et al.*, "A Threshold for Quantum Advantage in Derivative Pricing," *Quantum*, 2021.

[26] M. Salm *et al.*, "The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms," in *Service-Oriented Computing*, 2020.

[27] M. Salm *et al.*, "Automating the Comparison of Quantum Compilers for Quantum Circuits," in *Service-Oriented Computing*, 2021.

[28] *Arline: Quantum-Applied Machine Learning*, https://www.arline.io, Accessed: 2022-05-22, Arline.

[29] D. Mills *et al.*, "Application-Motivated, Holistic Benchmarking of a Full Quantum Computing Stack," *Quantum*, 2021.

[30] T. Lubinski *et al.*, *Application-Oriented Performance Benchmarks for Quantum Computing*, 2021. arXiv: 2110.03137.

[31] M. Salm *et al.*, "Optimizing the prioritization of compiled quantum circuits by machine learning approaches," in *Service-Oriented Computing*, 2022.

[32] H. Wang *et al.*, *QuEest: Graph transformer for quantum circuit reliability estimation*, 2022. arXiv: 2210.16724.

[33] N. Quetschlich, L. Burgholzer, and R. Wille, "Compiler Optimization for Quantum Computing Using Reinforcement Learning," in *Design Automation Conf.*, 2023.

[34] A. W. Cross *et al.*, *Open Quantum Assembly Language*, 2017. arXiv: 1707.03429.

[35] T. Tomesh *et al.*, *Supermarq: A scalable quantum benchmark suite*, 2022. arXiv: 2202.11045.

[36] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[37] L. Breiman, "Random forests," *Machine learning*, 2001.

[38] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of statistics*, 2001.

[39] L. Breiman *et al.*, *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[40] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, 1967.

[41] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[42] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, 1995.

[43] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society, Series B (Methodological)*, 1977.

[44] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, *et al.*, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, 2007.

[45] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 1992.

[46] N. Burkart and M. F. Huber, "A survey on the explainability of supervised machine learning," *Journal of Artificial Intelligence Research*, 2021.

[47] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, "Explainable ai: A review of machine learning interpretability methods," *Entropy*, 2021.

[48] J. T. Ash and R. P. Adams, "On warm-starting neural network training," in *International Conference on Neural Information Processing Systems*, 2020.