MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing

NILS QUETSCHLICH, Technical University of Munich, Germany
LUKAS BURGHOLZER, Technical University of Munich, Germany
ROBERT WILLE, Technical University of Munich, Germany and Software Competence Center Hagenberg GmbH, Austria

Fueled by recent accomplishments in quantum computing hardware and software, an increasing number of problems from various application domains are being explored as potential use cases for this new technology. Similarly to classical computing, realizing an application on a particular quantum device requires the corresponding (quantum) circuit to be *compiled* so that it can be executed on the device. With a steadily growing number of available devices—each with their own advantages and disadvantages—and a wide variety of different compilation tools, the number of choices to consider when trying to realize an application is quickly exploding. Due to missing tool support and automation, especially end-users who are not quantum computing experts are easily left unsupported and overwhelmed.

In this work, we propose a methodology that allows one to *automatically select* a suitable quantum device for a particular application *and* provides an *optimized compiler* for the selected device. The resulting framework—called the *MQT Predictor*—not only supports end-users in navigating the vast landscape of choices, it also allows *mixing and matching* compiler passes from various tools to create optimized compilers that transcend the individual tools. Evaluations of an exemplary framework instantiation based on more than 500 quantum circuits and seven devices have shown that—compared to both Qiskit's and TKET's most optimized compilation flows for all devices—the MQT Predictor produces circuits within the top-3 out of 14 baselines in more than 98% of cases while frequently outperforming any tested combination by up to 53% when optimizing for *expected fidelity*. Additionally, the framework is trained and evaluated for *critical depth* as another *figure of merit* to showcase its flexibility and generalizability—producing circuits within the top-3 in 89% of cases while frequently outperforming any tested combination by up to 400%. MQT Predictor is part of the *Munich Quantum Toolkit* (MQT) and publicly available as open-source on GitHub (https://github.com/cda-tum/mqt-predictor) and as an easy-to-use *Python* package (https://pypi.org/p/mqt.predictor).

1 INTRODUCTION

Quantum computing is an emerging computational technology and has seen considerable accomplishments in both hardware and software—especially in recent years. New quantum devices with an increasing number of qubits and higher fidelity rates are emerging almost on a daily basis. A similar trend can be observed for the development of quantum *Software Development Kits* (SDKs) such as, e.g., IBM's Qiskit [1], Quantinuum's TKET [2], Google's Cirq [3], Xanadu's PennyLane [4], or Rigetti's Forest [5]. This sparks interest not only in academia but also in industry—with promising applications of quantum computing emerging in various domains, such as finance [6], optimization [7], machine learning [8], or chemistry [9]. However, realizing such quantum computing-based solutions remains a challenge that is not straight-forward to solve and, to date, requires several manual steps [10].

First, a suitable quantum device must be *selected* for the execution of the developed quantum algorithm. This alone is non-trivial since new quantum devices based on various underlying technologies emerge almost daily—each with their own advantages and disadvantages. There are hardly any practical guidelines on which device to choose based on the targeted application. As such, the best guess in many cases today is to simply try out many (if not all) possible devices and, afterwards, choose the best results—certainly a time- and resource-consuming endeavor that is not sustainable for the future.

Once a target device has been selected, the quantum circuit, which is typically designed in a device-agnostic fashion that does not account for any hardware limitations (such as a limited gate-set or limited connectivity), must be *compiled* accordingly so that it actually becomes executable on that device. This is similar to classical computing, where high-level, platform-agnostic code (e.g., written in C++) is being *compiled* to low-level, platform-specific machine code before being executed on a particular classical computer. Compilation itself is a sequential process consisting of a sequence of *compilation passes* that, step-by-step, transform the original quantum circuit so that it eventually conforms to the limitations imposed by the target device. Since many of the underlying problems in compilation are computationally hard (e.g., satisfying the connectivity limitations of a certain device with the least amount of overhead being NP-complete [11]), there is an ever-growing variety of compilation passes available across several quantum SDKs and software tools—again, each with their own advantages and disadvantages. As a result of the sheer number of options, choosing the best sequence of compilation passes for a given application is nearly impossible. Consequently, most quantum SDKs (such as Qiskit and TKET) provide easy-to-use high-level function calls that encapsulate "their" sequence of compilation passes into a single compilation flow. While this allows to conveniently compile circuits, it has several drawbacks. For one, it creates a kind of *vendor lock* that limits the available compilation passes to those available in the SDK offering the compilation flow. Furthermore, the respective compilation flows are designed to be broadly applicable and, hence, are neither device-specific nor circuit-specific. On top of that, no means are provided to optimize for a specific *figure of merit*—only the degree of desired optimization can be specified without any explicit optimization criterion.

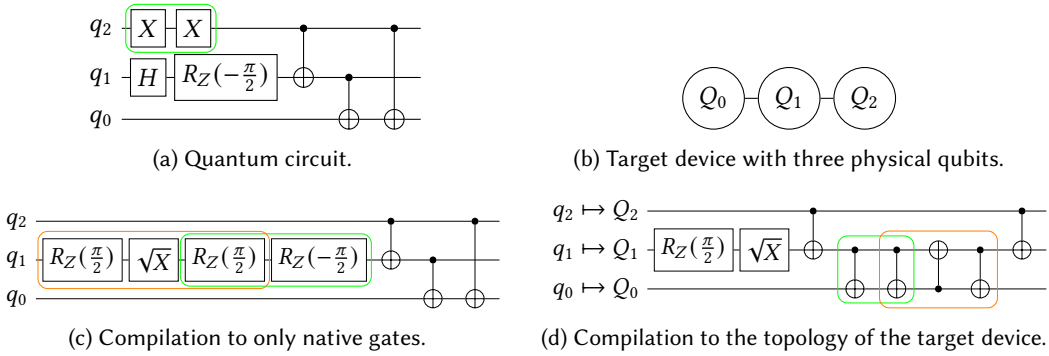
Overall, this leaves anyone trying to realize a quantum computing application with more options than they could ever feasibly explore. What makes things worse is that the choice of the quantum device and the subsequent compilation flow heavily affect the quality of the resulting circuit and its execution. Choosing the right combination can often make the difference between a useful result and pure noise. This is especially unfortunate for end-users who are not experts in quantum computing and just want to use the technology to solve their application domain problem.

In this work[1], a new approach to *automate* the quantum device selection and according compilation is proposed to free end-users from this task and to prevent a scenario where quantum computers can only be utilized by quantum computing experts—hindering the overall adoption of that technology. To this end, we tackle this problem from two angles:

(1) We propose a *prediction* method (based on *Supervised Machine Learning*) that, without performing any compilation, automatically predicts the most suitable device for a given application. This completely eliminates the manual and laborious task of determining a suitable target device and guides end-users through the vast landscape of choices without the need for quantum computing expertise.
(2) We propose a method (based on *Reinforcement Learning*) that produces device-specific quantum circuit compilers by combining compilation passes from various compiler tools and learning optimized sequences of those passes with respect to a customizable *figure of merit*. This *mix-and-match* of compiler passes from various tools allows one to eliminate vendor locks and to create optimized compilers that transcend the individual tools.

Combining both of these methods into a holistic framework—the *MQT Predictor*—allows one to *automatically select* a suitable quantum device for a particular application while also providing an *optimized compiler* for the selected device and a customizable figure of merit.

[1]Preliminary versions of this work have been published in [12], [13].



(a) Quantum circuit.

(b) Target device with three physical qubits.

(c) Compilation to only native gates.

(d) Compilation to the topology of the target device.

Fig. 1. Compilation of a quantum circuit to a targeted device.

To show its benefits, an instantiation of the proposed approach is implemented based on more than 500 quantum circuits ranging from 2 to 90 qubits and considering seven different quantum devices based on two qubit technologies. As a baseline to compare against, the most optimized compilation flows available in Qiskit and TKET have been used—leading to a total of $7 \times 2 = 14$ possible combinations of device and compiler. Compared to these reference results, the MQT Predictor produces circuits with an expected fidelity that is *on par with the best* possible result that could be achieved by trying out all combinations of devices and compilers. Specifically, the *automatic device selection* and *optimized circuit compilation* produced circuits within the top-3 in more than 98% of cases while frequently outperforming any tested combination by up to 53%. Additionally, MQT Predictor is trained and evaluated for *critical depth* as another figure of merit to showcase its flexibility and generalizability. The respective results show that the proposed method also achieves a similar performance—producing circuits within the top-3 in 89% of cases while frequently outperforming any tested combination by up to 400%. The corresponding framework (which is part of the *Munich Quantum Toolkit* (MQT) [14]) including the pre-trained models and classifiers is publicly available as open-source on GitHub (https://github.com/cda-tum/mqt-predictor) and as an easy-to-use *Python* package (https://pypi.org/p/mqt.predictor).

The rest of this work is structured as follows: In Section 2, quantum circuit compilation flows and related work are reviewed. Based on that, Section 3 introduces the proposed approach and the underlying methodologies with details given in Section 4 for the compilation using reinforcement learning and Section 5 for the device selection using supervised machine learning. The resulting combined approach is then described in Section 6, evaluated in Section 7, and discussed in Section 8. Section 9 concludes this work.

2 BACKGROUND AND MOTIVATION

To keep this work self-contained, this section gives a brief introduction to quantum circuit compilation, discusses its challenges, and reviews the related work.

2.1 Quantum Circuit Compilation

Solving any problem using quantum computing requires encoding the problem as a quantum algorithm. These algorithms are typically described as sequences of quantum operations (often referred to as *quantum gates*) that form a *quantum circuit*. Usually, one distinguishes between *single-qubit* gates affecting only one qubit and *multi-qubit* gates affecting more than one qubit at the same time.

Example 1. *Fig. 1a shows a small exemplary quantum circuit with three qubits (q_0, q_1, q_2) and four different kinds of quantum gates acting on them. There are two single-qubit gates on q_2 (two* NOT *gates denoted as* X*) and two single-qubit gates on q_1 (a Hadamard gate and a Z-rotation gate denoted as* H *and R_Z, respectively). Additionally, there are multi-qubit gates between all possible qubit pairs, namely* controlled-NOT *(CNOT) gates denoted as $\bullet$ and $\oplus$ for their control, respectively, target qubit.*

The actual compilation can only be started as soon as a suitable quantum device is *selected* and, by that, its native gate-set and connectivity constraints are known. The device selection itself is already challenging due to several factors:

- Various underlying quantum technologies are currently pursued in parallel with no clear winner yet and all of them have their own advantages and disadvantages. *Ion trap*-based devices, e.g., feature an all-to-all connectivity but provide rather slow gate execution times while, e.g., *superconducting*-based devices provide fast gate execution times but usually come with a restricted connectivity.
- Even devices based on the same technology significantly vary in their hardware characteristics, such as qubit count, their respective error rates, coherence times, qubit connectivity, and gate execution times.
- 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.

Example 2. *Fig. 1b shows the topology of a three-qubit quantum device. This topology indicates that two-qubit gates may only be applied to (Q_0, Q_1) or (Q_1, Q_2), but not to (Q_0, Q_2).*

As introduced in Section 1, executing such quantum circuits on an actual quantum device requires compilation, which is usually conducted by *compilers* that are part of quantum SDKs as easy-to-use high-level function calls. However, compilation actually is a sequential process that can be roughly divided into three kinds of *compilation passes*: *Synthesis* passes compile a given quantum circuit to the native gate-set of a selected quantum device, *mapping* passes enforce its connectivity constraints, and *optimization* passes exploit various possibilities to reduce the circuit's complexity. Quantum SDKs usually offer a wide range of device-independent optimization passes with the goal of creating a more efficient quantum circuit which still realizes the same functionality by, e.g., gate-cancellation, gate-commutation, or high-level synthesis routines.

Example 3. *Since $X \cdot X = I$, the circuit shown in Fig. 1a can be simplified by canceling the two subsequent* X *gates (denoted by the green box).*

Usually, those device-independent optimization passes are followed by the *device-dependent* compilation passes. Every quantum device can only directly execute a subset of all possible quantum gates—its so-called native gates. These native gates usually comprise a handful of single-qubit gates and one multi-qubit gate to become a *universal* gate-set. Therefore, all non-native gates must be *synthesized* into a sequence of native gates of that respective quantum device—a procedure that is NP-complete to solve in an optimal fashion [15]. Consequently, this alters the quantum circuit—leading to further room for optimization.

Example 4. *Assume that the targeted device from Example 2 has a native gate-set consisting of R_Z, $\sqrt{X}$,* X*, and* CNOT *gates and that the circuit from Fig. 1a shall be executed on it. Only the Hadamard gate on qubit q_1 is not a native gate of the device and, hence, must be synthesized into a sequence of such gates. The orange box in Fig. 1c indicates one such decomposition into two $R_Z(\frac{\pi}{2})$ and a $\sqrt{X}$ gate. Again, this circuit alteration leads to further optimization potential since the two subsequent R_Z gates in the green box, again, cancel each other out.*

In addition to the limited gate-set, many quantum devices (e.g., based on such as superconducting qubits) only have a *limited connectivity* between their underlying qubits—requiring that all multi-qubit gates must be placed on qubits which are actually connected on the respective device. To adapt a quantum circuit with gates operating between arbitrary qubits to such a limited connectivity, a *mapping* pass is necessary. This pass is often split into *layouting* and *routing*. The former assigns each (logical) qubit of the circuit to a (physical) qubit on the device and the latter ensures that all connectivity constraints are satisfied by changing the qubit assignment throughout the circuit, e.g., using *SWAP* gates—a procedure that is, again, NP-complete to solve in an optimal fashion [11]. Then, again, optimization passes can be applied to reduce the complexity of the resulting quantum circuit.

Example 5. *Consider again the circuit shown in Fig. 1c which shall be executed on the three-qubit quantum device shown in Fig. 1b. Since the quantum circuit contains multi-qubit gates between all qubit pairs, there is no trivial layout that satisfies the connectivity constraints throughout the entire circuit. Therefore, the circuit must be mapped by introducing a* SWAP *gate as denoted in the orange box in Fig. 1d (again, compiled into its native gate-set sequence of three* CNOT *gates). Although the resulting quantum circuit is now fully executable, a successive optimization pass can, again, eliminate two subsequent* CNOT *gates.*

Although the compilation process is rather straight-forward conceptually—consisting of *synthesis*, *mapping*, and *optimization* passes—it comes with practical challenges. For each step, a variety of different techniques and approaches have been proposed, such as [15]–[20] for synthesis, [21]–[29] for mapping, and [30]–[33] for optimization. Therefore, end-users are easily overwhelmed with selecting the *best* combination of provided SDKs and respective compilation passes—not even mentioning the effort of becoming acquainted with multiple (often poorly documented) SDKs and, respectively, needed conversions between them. Every SDK usually implements a subset of these compilation passes with varying degrees of effectiveness and efficiency. Those passes are provided as pre-configured, fixed sequences of compilation passes (for various levels of desired optimization)—independent of the given quantum circuit.

2.2 Related Work

A handful of techniques have already been explored to support the quantum device selection and compilation. In [34]–[37], approaches that compile a given circuit for *all* devices in a brute-force fashion and compare their resulting quality (also referred to as *autotuning*) are proposed. A similar approach to select the best sequence of compilation passes by evaluating all possible combinations has been proposed in [38]. Furthermore, in [39], a methodology has been proposed to determine whether compiled circuits can be reliably executed on certain devices to give some guidance. The same authors have further explored how *Multi-Criteria Decision Analysis* methods can be used to select the most suitable compiled circuit from a given set of all compiled circuits [40] and how machine learning can be used to predict the goodness of a device and compiler combination [41].

Supervised Machine Learning (ML) techniques have also been applied to quantum compilation, e.g., in [42], an ML-based layout and routing scheme is proposed, while in [43], an ML model is trained to predict the optimization potential of a given quantum circuit to decide whether to apply (potentially costly) optimization passes or not. Furthermore, *Reinforcement Learning* (RL) has been explored in this context, e.g., in [44] where a strategy for applying individual gate transformation rules to optimize quantum circuits is learned or in [45] where an RL model is trained to learn how to estimate the circuit fidelity for specific devices. Moreover, environments to train RL models for various compilation tasks are proposed in [46]—similar to the environment proposed in [13].

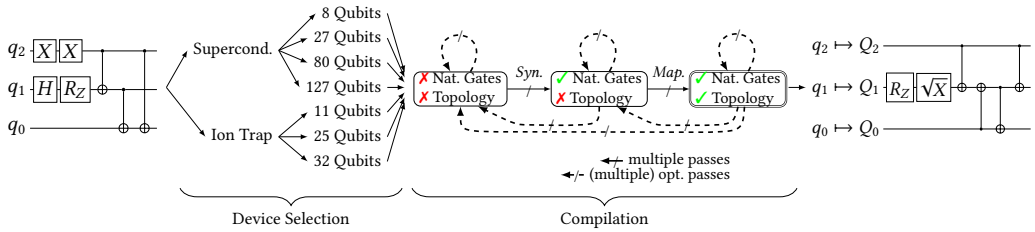


Fig. 2. The device selection and compilation process.

In classical compilation, machine learning has been explored for a much longer time which led to sophisticated compiler optimization techniques with overviews given in, e.g., [47], [48] and established these methods as state-of-the-art techniques in this domain. Additionally, even a combination of these techniques has been explored in [49] by using a machine learning model to determine promising areas of optimization space which then can be further investigated using autotuning. On top of that, reinforcement learning approaches targeting the LLVM [50] compiler emerged especially in recent years with encouraging results [51]–[53].

So far, many of the approaches toward both quantum device selection and compilation [34]–[40] follow a naive brute-force approach that, obviously, is not a long-term solution due to the steadily increasing number of quantum devices and compilers. Furthermore, existing ML-based tools to support end-users [41]–[45] do not consider the entire compilation or do not provide a broad coverage in terms of supported devices, range of qubits, or considered algorithms, and frequently lack the necessary means to scale to significantly more devices and compilers. On top of that, most approaches focus on using current compilers *as-is* and hardly make use of the sophisticated ML-based approaches for classical compiler optimization—leading to lots of untapped potential.

3 GENERAL IDEA

In this work, we propose a new approach to *automate* the device selection and compilation process that learns from previously conducted compilations and *predicts* the most suitable target device for a given quantum circuit *as well as* provides an *optimized compiler* for the selected device. To this end, this section motivates the proposed approach and describes its general ideas, the chosen methodology, and its utilization.

3.1 Proposed Approach

In an ideal scenario, a quantum circuit would be compiled for all possible sequences of compilations on all possible devices to determine the best compilation flow. However, since compilation comes with considerable complexity both in time and resources, this is practically infeasible.

Example 6. *Assume that we want to find the best compiled version of the quantum circuit shown in Fig. 1a given access to seven different devices (based on two qubit technologies) and various compiler passes provided by multiple SDKs as illustrated in Fig. 2. Manually determining the best possible combination of options is not only challenging, but close to impossible due to the number of possible combinations—which, in theory, is infinitely large due to backward loops in the compilation process that are introduced by optimization passes. End-users could just choose* any *(random) path from the left-hand side of Fig. 2 to the right-hand side. However, this is highly unlikely to yield a sufficiently good result (not to dare, the best possible result).*

In the proposed approach, we first consider finding the best sequence of *compilation* passes for a particular device as an isolated problem that is independent of the device selection task itself. To this end, compilation is modeled as a sequential (*Markov Decision)* process of synthesis, mapping, and optimization passes for a given native gate-set and qubit connectivity. Using *Reinforcement Learning* (RL), respective models can be trained to *learn* the sequences that lead to the most promising compilation result for each supported device according to some *figure of merit* such as expected fidelity or critical depth. These models can then be used as optimized black-box compilers for the respective devices.

Based on these optimized compilers, the task of choosing the most promising device is tackled. To this end, *Supervised Machine Learning* (ML) is applied by training an agent that *learns* the best suited device for a given circuit assuming that it will be compiled using the trained RL models. Thus, a holistic approach is created that takes a to-be-compiled quantum circuit as input and predicts which quantum device to use, compiles it accordingly using the trained RL model, and returns the compiled circuit with detailed compilation information. All of this is conducted in a fully automated fashion which is especially helpful for end-users who are not experts in quantum computing. Additionally, compilation passes from various sources can be integrated—effectively eliminating vendor locks by mixing and matching compiler passes from various SDKs.

To learn how to *best* compile quantum circuits, it is necessary to give the notion of "best" a meaning by defining a *figure of merit* to optimize for. As this is an essential part of the proposed approach, more details on this way of customizing the compilation are provided in the following.

3.2 Figures of Merit

In general, the figure of merit can be completely customizable and consider *anything* from (compiled) circuit characteristics such as gate counts to *soft* factors—applicable to the device selection—such as actual execution costs or the queue length for a specific device vendor. Nevertheless, a comprehensive figure of merit should consider at least the characteristics of the compiled quantum circuit and the characteristics of the selected quantum device. Therefore, the respective hardware information needs to be collected, such as, e.g., gate/readout fidelities, gate execution times, or decoherence times. While for some devices, this information may be publicly available, for other devices, it may be estimated from comparable devices, previous records, or insider knowledge.

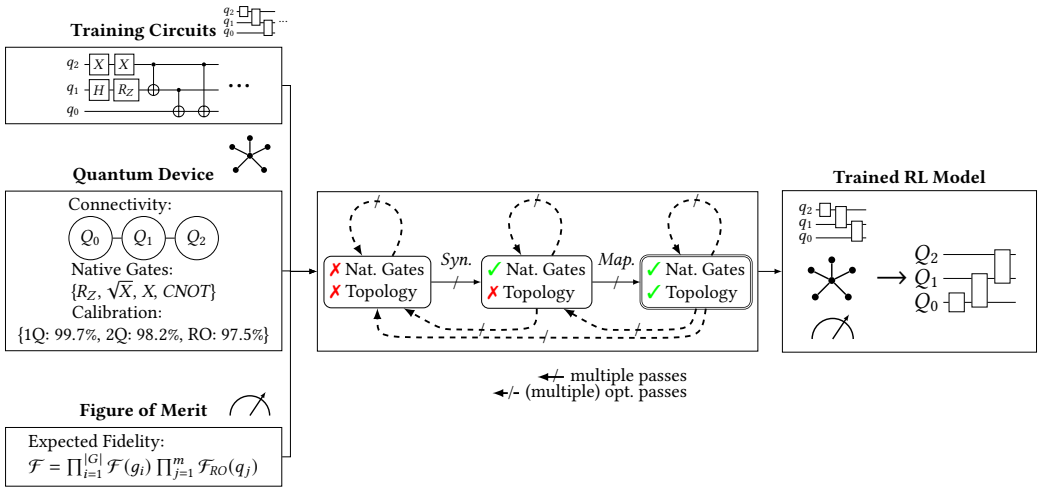
Example 7. *Consider the figure of merit that takes into account the following two aspects:*

(1) *If the selected device is not large enough to execute a given quantum circuit with respective to its number of qubits, the worst-possible score is assigned.*
(2) *If the device is large enough, the evaluation score is calculated using the formula:*

$$\mathcal{F} = \prod_{i=1}^{|G|} \mathcal{F}(g_i) \prod_{j=1}^{m} \mathcal{F}_{RO}(q_j)$$

with $\mathcal{F}(g_i)$ being the expected execution fidelity of gate g_i on its corresponding qubit(s), $\mathcal{F}_{RO}(q_j)$ being the expected execution fidelity of a measurement operation q_j on its corresponding qubit and $|G|$ respectively m being the number of gates and measurements in the compiled circuit.

This figure of merit determines an estimate of the probability that a quantum circuit will return the expected result, the so-called expected fidelity*, which ranges between* 0.0 *and* 1.0 *with higher values being better.*



(a) Input. (b) MDP for quantum compilation. (c) Output.

Fig. 3. Training process of an RL model for quantum compilation.

However, the expected fidelity introduced above is just one exemplary figure of merit. A potential alternative could be the *critical depth* (taken from [54])—a measure to describe the percentage of multi-qubit gates on the longest path through a compiled quantum circuit (determining the depth). A respective value close to 1 would indicate a very sequential circuit while a value of 0 would indicate a highly parallel one. On top of that, even combinations with customizable weighting are possible, such as 25% of the expected fidelity and 75% critical depth. Furthermore, different figures of merit could be used for the device selection and the respective compilation itself. While for the former, soft factors such as actual execution costs or queue lengths might be sensible to consider as well, the latter should only focus on technical factors that describe the quality of the compilation itself.

In the following, detailed descriptions of both the RL-based approach to the compilation and the ML-based approach to the device selection are given in Section 4 and Section 5, respectively. Afterwards, Section 6 describes the combination of both techniques into a holistic framework.

4 COMPILATION USING REINFORCEMENT LEARNING

Compilation, fortunately, is not new per-se, since classical compilers have seen a similar trend of increasing complexity and variety in the past. Instead of reinventing the wheel, we propose to make use of the decades of research on classical compiler optimization and model quantum circuit compilation in a similar fashion. Based on that, classical reinforcement learning is used to learn optimized sequences of compilation passes for a given figure of merit. The resulting model can then be used as a compiler.

4.1 Quantum Circuit Compilation Modeled as a Markov Decision Process

Quantum circuit compilation as described in Section 2.1 can be interpreted as a *Markov Decision Process* (MDP, [55]) that transforms a high-level circuit into an executable circuit by applying synthesis, mapping, and optimization passes. For that, the compilation process is broken down into *states* and corresponding *actions* that can be executed in those states to advance the process. Each

state captures which of the requirements for a circuit to be executable are satisfied—specifically, whether the circuit only contains native gates and whether it matches the topology of the device. This results in three states[2]:

(1) The circuit contains non-native gates and does not respect the device topology,
(2) The circuit only contains native gates but does not respect the device topology, and
(3) The circuit only contains native gates and respects the device topology, i.e., it is executable.

The resulting MDP is depicted in Fig. 3b—with states being denoted as rounded rectangles and actions as arrows. In the following, the individual states and actions will be described in more detail alongside an example.

Given a high-level quantum circuit and a target device (determining the native gate-set and qubit topology), compilation typically begins in the state where the circuit contains non-native gates and does not respect the device topology. In this state, two kinds of actions are possible: Either a *synthesis action* that transforms the circuit so that it only contains gates native to the target device, or an *optimization action* that applies any kind of optimization to the circuit.

Example 8. *To make this modeling more tangible, consider again the compilation flow shown in Fig. 1 with the initial circuit depicted in Fig. 1a. Since it comprises a non-native gate, the compilation starts from the left-most state in Fig. 3b. First, an optimization pass is applied that cancels the two redundant* X *gates. Since the circuit still contains non-native gates, this does not change the overall state—as indicated by the dashed self-loop above the state. Then, a synthesis pass is applied, which yields the circuit depicted in Fig. 1c and advances the state of the MDP to the next state.*

Whenever the circuit only contains native gates but does not yet respect the device topology, three different actions are possible: First, the circuit could be mapped using any *mapping action*—making the circuit executable and, thus, advancing the MDP to the final state. Additionally, the circuit may be optimized again. Depending on whether the optimization pass preserves the native gates, the state after the action either stays the same (*gate-set-preserving optimization action*) or goes back to the previous, non-native gates state (*general optimization action*).

Example 9. *Consider again the synthesized quantum circuit from Fig. 1c. As before in Example 8, a similar optimization is conducted to cancel the two rotation gates. Since the resulting circuit still only consists of native gates, this optimization does not change the state of the MDP—again represented by a dashed self-loop in Fig. 3b. Subsequently, a mapping action is applied that leads to the quantum circuit depicted in Fig. 1d and advances the state of the MDP to the final state.*

In the final state, both the device's native gate-set and qubit connectivity constraints are fulfilled and any circuit in this state is already executable. However, additional *optimization* actions can be applied to further reduce complexity. Again, those actions might violate any of the two constraints and the circuit might end up in a state where it is not executable any more.

Example 10. *The quantum circuit resulting from Example 9 is already executable. However, another optimization action is applied to the circuit depicted in Fig. 1d to cancel two further gates and, by that, reducing the complexity. Since this does not violate the device's connectivity, the circuit remains executable and the MDP state does not change—again represented by a dashed self-loop in Fig. 3b.*

[2]We purposely omit the state where the circuit respects the topology but contains non-native gates as there are hardly any compilation passes available that take advantage of this specific combination of conditions.

This MDP formulation enables a modular framework based on two properties:

(1) Each of the constraints that characterize an MDP state is *efficiently computable* (via a single traversal of the circuit).
(2) All actions have a *unified interface* based on a common intermediate representation (IR) for their input and output—independent of the origin of a certain action.

The former ensures that, at any stage during the compilation, state transitions within the MDP can be computed efficiently. The latter ensures interoperability between different SDKs and makes it possible to *mix and match* compilation passes provided by multiple SDKs. Consequently, they can be combined and integrated in a single compilation framework—mitigating vendor lock-ins, allowing one to quickly adapt and integrate further compilation passes, and helping to keep up with the fast-paced development in quantum computing software.

4.2 Learning Optimal Compilation Sequences Using Reinforcement Learning

In classical compilation, different problems have been tackled very successfully using *Reinforcement Learning* (RL) [51]–[53]. In general, these techniques aim to learn an *action policy* that determines which actions should be applied based on *observations* of the current state with the goal of maximizing a cumulative *reward function*. In contrast to providing labeled training data (as in *Supervised Machine Learning*, ML), a *training environment* representing the underlying MDP including its states and actions must be defined.

To avoid reinventing the wheel, we also propose using such an RL approach for quantum circuit compilation based on the MDP shown in Fig. 3b. To this end, three things must be provided as input (which are depicted in Fig. 3a):

(1) *Training Circuits*: Used in the training process to learn an action policy.
(2) *Quantum Device*: Defines the native gate-set, the qubit connectivity, and certain characteristics (such as gate fidelities) for the compilation process.
(3) *Figure of Merit*: Defines the evaluation score to guide the learning process.

An *RL agent* can then be employed to learn a corresponding action policy that maximizes the expected cumulative reward. For the training itself, a *sparse* reward function is used, i.e., as long as the circuit is not executable, the reward is always zero, and whenever the circuit becomes executable, the chosen figure of merit (e.g., expected fidelity as described in Section 3.2) is used to assign a score to the compiled circuit.

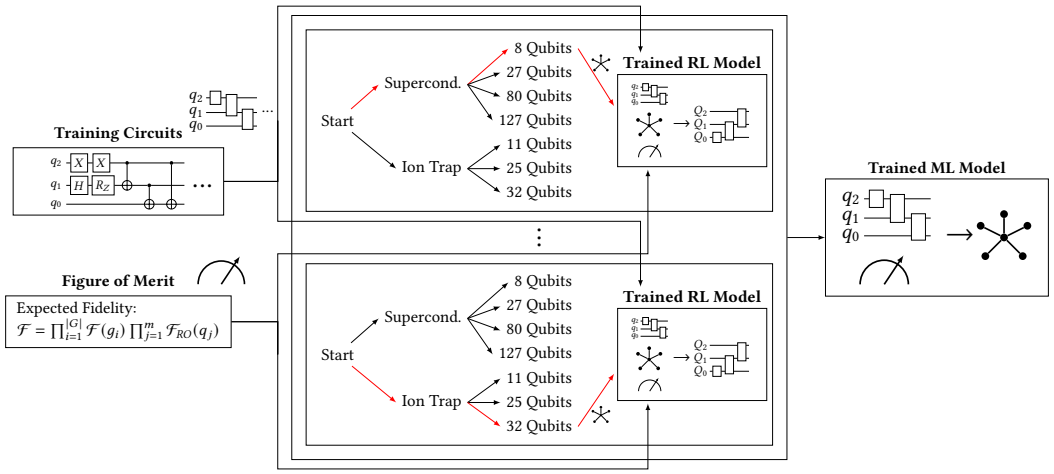
The resulting trained RL model then acts as *compiler* itself that can compile any given quantum circuit for the target device it was trained for—as illustrated in Fig. 3c. This creates a compiler that is fine-tuned for a specific device, optimizes for a customizable figure of merit, and factors in the input circuit when determining which compilation passes to apply—three major advances over current quantum circuit compilers.

5 DEVICE SELECTION USING SUPERVISED MACHINE LEARNING

As discussed in Section 2.1, selecting the best quantum device for a particular application is itself a challenging task. This section describes how that task can be interpreted as a classification problem and how supervised machine learning can predict the most promising qubit technology and device for a given quantum circuit—therefore, automatically make this decision for an end-user.

5.1 Quantum Device Selection Modeled as a Classification Task

A naive approach to selecting the best quantum device for a given quantum circuit would be to first compile it for all devices using a specific compiler, e.g., the trained RL models from Section 4. Afterwards, the resulting compiled circuits must be evaluated according to some *figure of merit*



(a) Input. (b) Classification task for the quantum device selection. (c) Output.

Fig. 4. Training process of an ML model for the quantum device selection.

to identify the most promising device. However, doing this for each and every to-be-compiled quantum circuit is practically infeasible since compilation is a time-consuming task and the number of available devices is steadily growing.

Example 11. *There are various underlying technologies how to realize quantum computers such as, e.g., superconducting- and ion trap-based qubits. Even within a technology, respective devices may significantly vary in their characteristics such as the connectivity scheme, native gate-set, and fidelity rates. Assuming that an end-user has access to seven devices based on two technologies, the task of determining the best device for a given circuit and figure of merit already becomes challenging since seven different compilations would have to be conducted—as insinuated in Fig. 4b.*

Therefore, we propose an approach to *learn* from previously conducted compilations and interpret the selection of the most promising device for a circuit and figure of merit as a statistical *classification* task—a perfect fit for supervised machine learning. By that, end-users are freed from answering the following questions themselves:

- Which *qubit technology* is best suited for the application at hand?
- Which particular *device*, e.g., from IBM or Rigetti, fits the quantum algorithm best?

5.2 Predicting a Quantum Device Using Supervised Machine Learning

Supervised Machine Learning (ML) has proven a promising methodology in classical compilation (as described in [47], [48]) but also in various other domains, such as, e.g., medicine, cyber security, and predictive analytics [56]. A crucial part to utilize this technology is gathering suitable training data which are representative for the whole problem domain space to facilitate generalizability—in this case, representative training circuits covering a broad range of applications. Subsequently, each training circuit must be compiled for *all* possible devices—using the trained RL models that act as compilers—to be able to identify the most promising according to the chosen figure of merit as shown in Fig. 4b. That device then acts as the *classification label* and, together with the training circuit itself, constitutes one *training sample*. To generate all the *labeled training data* necessary

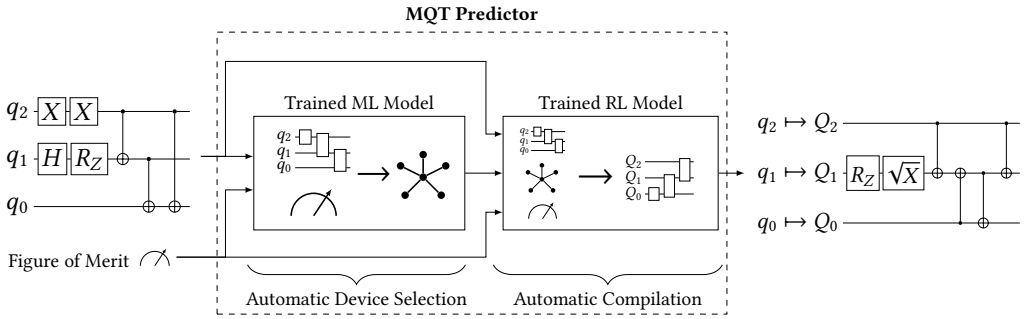


Fig. 5. Automatic compilation flow selection and execution using MQT Predictor.

to train respective ML models, this process is repeated for each training circuit and requires two things that must be provided (which are depicted in Fig. 4a):

(1) *Training Circuits*: Similar to the RL models, used here to generate labeled training data[3].
(2) *Figure of Merit*: Determines the most promising device based on all conducted compilations.

Each labeled circuit must then be transformed into a *feature vector* to be suitable for training a classifier. There are many degrees of freedom when choosing the respective feature and it is very much an active field of research to determine features that best characterize a given quantum circuit. In practice, these features should at least include the number of qubits and some statistics about the interactions of gates in the circuit, such as depth, parallelism, or interaction frequency.

Subsequently, the actual ML model can be trained. This methodology with the respectively trained ML model (as illustrated in Fig. 4c) then acts as a tool to automatically predict the most promising quantum device for a given circuit and figure of merit. As a result, the user gets a recommendation that is circuit-specific, tuned for a specific figure of merit, and requires no compilation at all—again, three major advances over the established workflow.

6 PROPOSED APPROACH: MQT PREDICTOR

Together with the trained RL models from Section 4, the trained ML model automatically solves the device selection and compilation process—forming a holistic framework. In the following, this framework, called the *MQT Predictor*, and its implementation are described in more detail.

6.1 Approach

The proposed approach is shown in Fig. 5. It takes a quantum circuit and a customizable figure of merit as input. Then, the framework itself predicts which quantum device is the most promising using the trained ML model—in *realtime* without conducting any compilation at all. The outcome of this is the predicted device. Based on that, the respectively trained RL model compiles the given circuit for the predicted device optimizing for the chosen figure of merit. Finally, it returns the compiled quantum circuit. By that, the MQT Predictor completely frees end-users from the daunting tasks of selecting a proper quantum device and finding a good compiler and can be used with just a single *Python* function call—providing the same ease of use as the state-of-the-art compilers such as Qiskit [1]. On top of that, it combines the advantages of both individual techniques into a single powerful tool.

Example 12. *Consider again the original scenario shown in Fig. 2, where an end-user wants to execute the circuit shown again on the left-hand side of Fig. 5. Instead of manually finding the*

[3]To reduce the effort needed for regenerating labeled training data after changes, all compiled circuits including their evaluation score are persistently stored in a database.

most promising device and compiler passes, the circuit can be simply fed into the MQT Predictor framework together with the expected fidelity (as defined in Section 3.2) as its figure of merit. Then, the MQT Predictor predicts a respective device and compiles the circuit accordingly using the trained RL model—returning the compiled circuit shown on the right-hand side of Fig. 5.

The following sections describe the particular instantiation of the framework that has been implemented as part of this work.

6.2 Implementation of Automatic Compilation Using RL

The training environment described in Section 4 and shown in Fig. 3b is built on top of the open-source library *OpenAI Gym* [57]. For that, all compilation passes from different sources that should be considered in learning promising compilation sequences must be implemented through a *unified interface*.

To this end, *Qiskit's QuantumCircuit* is used as the *intermediate representation*. Therefore, a compilation pass can be incorporated if the following requirement is fulfilled: Either it must provide an interface for both the input and the compiled output circuit that supports *Qiskit's QuantumCircuit* or a parser that supports *OpenQASM* (version 2 or 3)—which, then, can be used to create and serialize a *Qiskit's QuantumCircuit* again.

As modeled in Fig. 3, three different types of actions are considered: synthesis, mapping, and optimization passes. Using the described unified interface for both input and output of a compilation action, compilation passes from various quantum SDKs can easily be incorporated. To demonstrate this, representative compilation passes from IBM's Qiskit (version 0.43.3) and Quantinuum's TKET (version 1.17.1) have been integrated—leading to the following list of available actions.

- *Synthesis*: Qiskit's *BasisTranslator*
- *Mapping*: Qiskit's *Sabre* mapping or any combination of the following methods
  - Layout:
    - Qiskit's *TrivialLayout*
    - Qiskit's *DenseLayout*
    - Qiskit's *SabreLayout*
  - Routing:
    - Qiskit's *BasicSwap*
    - Qiskit's *StochasticSwap*
    - Qiskit's *SabreSwap*
    - TKET's *RoutingPass*
- *Optimization*:
  - Qiskit's *Optimize1qGatesDecomposition*
  - Qiskit's *CXCancellation*
  - Qiskit's *CommutativeCancellation*
  - Qiskit's *CommutativeInverseCancellation*
  - Qiskit's *RemoveDiagonalGatesBeforeMeasure*
  - Qiskit's *InverseCancellation*
  - Qiskit's *OptimizeCliffords*
  - Qiskit's *Collect2qBlocks* + *ConsolidateBlocks*
  - Qiskit's *O3 Fixpoint Optimization Routine*
  - TKET's *PeepholeOptimise2Q*
  - TKET's *CliffordSimp*
  - TKET's *FullPeepholeOptimise*
  - TKET's *RemoveRedundancies*

For the training process, more than 500 training circuits from *MQT Bench* [58] on the target-independent level have been used ranging from 2 to 30 qubits. Lastly, a maskable version of the *Proximal Policy Optimization (PPO)* algorithm [59] provided by *Stable-Baselines3* [60] is used for the reinforcement learning training process itself and also takes care of potential infinite sequences due to the backward loops within the MDP.

6.3 Implementation of Automatic Device Selection Using ML

For the quantum device selection, the ML model described in Section 5 and shown in Fig. 4b is trained using scikit-learn [61]. In this regard, seven different underlying algorithms are implemented and evaluated in [12]—with grid-searched and 5-fold cross-validated parameter values—to determine the most promising one:

- *Random Forest* [62]
- *Gradient Boosting* [63]
- *Decision Tree* [64]
- *Nearest Neighbor* [65]
- *Multilayer Perceptron* [66]
- *Support Vector Machine* [67]
- *Naive Bayes* [68]

In [69], an overview of today's use of those techniques is given. All seven machine learning classifiers have been experimentally evaluated. Since the *Random Forest* classifier has given the best results, it is used in the following as the ML algorithm.

As indicated in Fig. 2, seven devices from two qubit technologies are considered as representatives:

- Superconducting: four devices with 8, 27, 80, and 127 qubits
- Ion Trap: three devices with 11, 25, and 32 qubits

To generate respective training data, more than 500 training circuits from *MQT Bench* [58] on the target-independent level are used (ranging from 2 to 90 qubits) with a 70/30 train-test-split. The compilations to generate the labeled training data are conducted based on the trained RL models acting as respective compilers.

6.4 Quantum Circuit Representation

Both the RL and the ML training process require quantum circuits to be represented as so-called *feature vectors*–a vectorized representation based on integer and floating point values that makes them suitable for training a respective model. Similar to the figure of merit described in Section 3.2, these features can be arbitrarily complex. In this instantiation, various characteristics are used to describe a quantum circuit for both models: the *number of qubits*, the *depth* of the circuit, and the five composite features of *program communication*, *critical-depth*, *entanglement-ratio*, *parallelism*, and *liveness* originally proposed in [54]:

- *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.
- *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.

Furthermore, for the ML model, the number of gates for each gate type according to the *OpenQASM 2.0* specification [70] are used as features. For the RL model, this was omitted to improve the learning efficiency, since the gate count features have been shown to be less representative than the composite ones [12].

7 EVALUATION

Based on the instantiation described above, this section demonstrates the feasibility of the MQT Predictor framework by means of numerical evaluation. All used pre-trained models and classifiers are publicly available as open-source on GitHub (https://github.com/cda-tum/mqt-predictor) and as an easy-to-use *Python* package (https://pypi.org/p/mqt.predictor).

7.1 Setup

To assess the quality of the proposed framework, it is evaluated on more than 150 benchmarks from 2 to 90 qubits taken from *MQT Bench* [58] using the target-independent level circuits and the MQT Predictor instantiation described in Section 6. For each of those evaluation circuits—which have not been part of the (ML) training circuits—the tool predicts the most promising target device and its optimized compiler to compile the given circuit to the selected device. To assess quality, two figures of merit are used:

(1) The expected fidelity as introduced in Section 3.2 and
(2) The critical depth describing the ratio of multi-qubit gates on the longest path of the compiled quantum circuit[4].

Note that the used figures of merit are examples and the MQT Predictor framework allows end-user to provide *any* metric to optimize for. In this case, the focus rather lies on technical factors to determine both the best device and compilation sequence while it might be sensible to also factor in soft factors such as the execution costs or the queue length for the device selection. Since the focus of this work is to provide a framework that optimizes for *any* given figure of merit, the following evaluations focus on how well the framework optimizes for these exemplary figures of merit and not on how suited these figures of merit are to assess the compilation task itself.

To generate baseline values, all evaluation circuits are compiled for all seven currently supported devices using both Qiskit's and TKET's most optimized (pre-configured and fixed) compilation settings *O3* respectively *O2*—leading to 14 different compiled circuits as baselines.

7.2 Results

Fig. 6 shows the results obtained by using the proposed framework to optimize the expected fidelity. Each tick on the x-axis denotes one evaluation circuit that is assessed using the 14 baseline compilation flows (7 devices × 2 compilers) and the MQT Predictor. Benchmarks leading to only zero values have been excluded from all evaluations. Since (maximally) 15 values per tick would be hard to read, the graph is reduced to only show the

- MQT Predictor evaluation score in blue,
- best evaluation score in green,
- median evaluation score in yellow, and
- worst evaluation score in red.

[4]Those ratios are actually subtracted from 1 such that higher values indicate a more parallel quantum circuit that is assumed to be better.

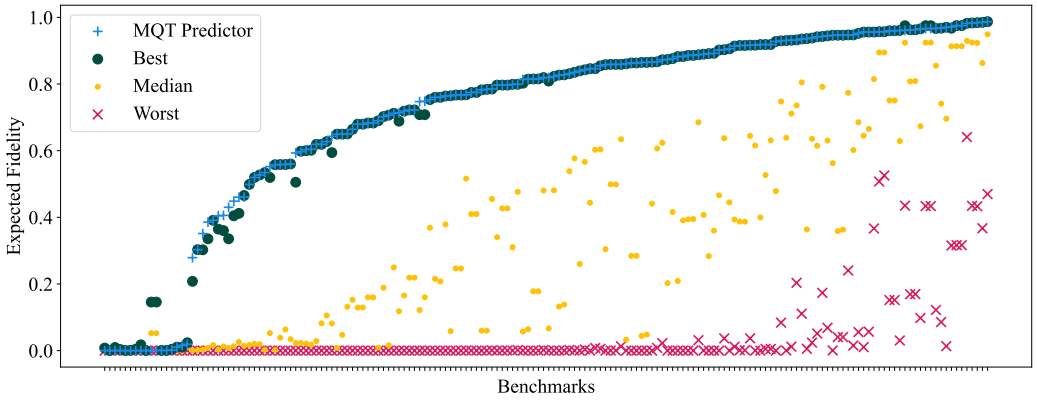


Fig. 6. Evaluation of the expected fidelity. All benchmarks are sorted by their MQT Predictor score for better readability.

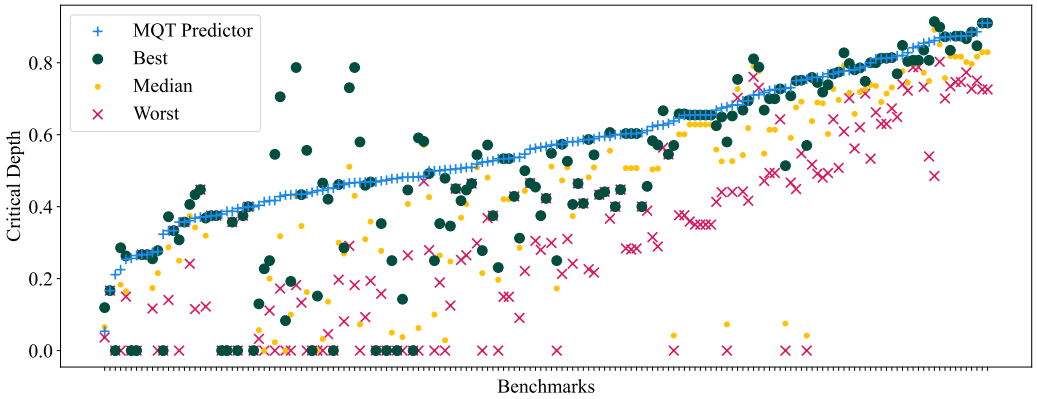


Fig. 7. Evaluation of the critical depth. All benchmarks are sorted by their MQT Predictor score for better readability.

In this setup, the proposed method is capable of producing compiled quantum circuits that are within the top-3 of all 14 baselines in over 98% of cases while frequently outperforming all baselines by up to 53%. Therefore, especially end-users who are not experts in quantum computing can use it to reliably and automatically compile quantum circuits whose quality otherwise can only be achieved by many manual experiments—often infeasible due to the necessary time effort and expert knowledge. These results also clearly highlight that choosing the right device and compiler can make the difference between a successful execution and obtaining completely random results.

To demonstrate the ability to optimize for customizable figures of merit, we additionally set up the framework to optimize for critical depth and show the respective results in Fig. 7. Again, benchmarks only leading to zero values have been excluded from the evaluations. Since it is not possible to define a custom figure of merit/optimization criterion for both Qiskit and TKET, the same compiled baseline evaluation circuits are used. The respective graph has been reduced in a similar fashion as described above. For 89% of cases, the compiled circuits produced by MQT Predictor are within the top-3 of the 14 baselines while frequently outperforming all baselines by up to 400%.

Table 1. Comparison of the influence of the figure of merit.

	Average result for...	
Model trained for...	Exp. Fidelity	Critical depth
Exp. Fidelity	**0.67**	0.42
Critical depth	0.12	**0.55**

These results describe the performance from the end-users' perspective, who use the MQT Predictor framework to automatically select the most promising device and compile accordingly. To provide more insight into the underlying training data, the ideal device distribution for the evaluated benchmarks is explored in Appendix A.1. Additionally, the impact of both the device selection as well as the compilation has been examined in an isolated fashion in Appendix A.2 and Appendix A.3, respectively.

To further investigate the frameworks ability to optimize for customizable figures of merit, the average evaluation scores for both figures of merit are calculated for both respectively trained frameworks. The resulting numbers are denoted in Table 1 and confirm that the framework trained for a particular figure of merit indeed results in the compiled circuits with the highest evaluation scores for both cases.

7.3 Generalizability

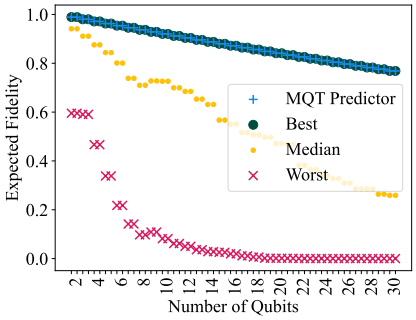
The performance of machine learning models, in general, is highly dependent on the quality of the training data. Therefore, training data should be as heterogeneous as possible to cover a wide variety of circuits with different characteristics to achieve a high *generalizability*. For that purpose, a multitude of different algorithms are considered in the training data for both the ML and RL models described above—ranging from quantum circuit building blocks such as, e.g., the *quantum fourier transform* and *amplitude estimation* towards rather complex quantum circuits such as, e.g., ground state estimation—all taken from *MQT Bench* [58].

To estimate how well the trained MQT Predictor works for not only the quantum circuits that are part of the training data but also for other algorithms that have not been considered during training, a further evaluation has been conducted. To evaluate generalizability, quantum circuits for the *Greenberger–Horne–Zeilinger* (GHZ) state, which have not been considered during training, are evaluated in a similar fashion as before in Section 7.2. The resulting scores for expected fidelity and critical depth are denoted in Fig. 8a and Fig. 8b, respectively.

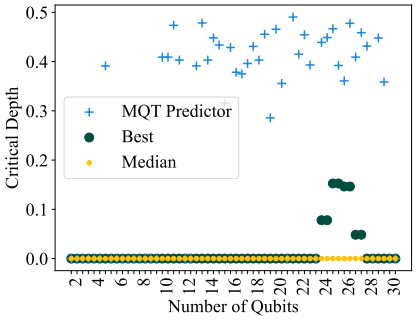
Taking into account both figures of merit, a performance similar to the results described in Section 7.2 is achieved. Considering the expected fidelity, MQT Predictor leads to the same performance as the best baseline for all cases. In case of the critical depth as a figure of merit, MQT Predictor even outperforms all baselines in most cases. It should be noted that the best baseline results for critical depth frequently are 0.0 (since *all* two-qubit gates are on the longest path), while the MQT Predictor achieves significant improvements in many of these cases. Although this is only a small evaluation of the generalizability of the proposed methodology, it indicates that it also works well in untrained scenarios.

8 DISCUSSION

Classical machine learning has shown tremendous capabilities to solve problems from various domains such as classical compilation and beyond. However, its potential success inherently depends on gathering a sufficiently large amount of high-quality training data and, by that, it is not per-se well suited for changes, which often require to re-train the underlying machine learning models. This section specifically gives guidance how the MQT Predictor deals with such changes in various ways.



(a) Expected fidelity for GHZ circuits.



(b) Critical depth for GHZ circuits.

Fig. 8. Generalizability evaluation for GHZ circuits which were not part of the training processes.

8.1 Changes in Hardware Information used in Figure of Merit

Quantum computers are frequently calibrated to maximize the actual gate fidelities, e.g., used in the figure of merit introduced in Section 3.2. Thus, these calibration updates affect the fidelity data used throughout both the RL and ML training as part of the cost function. Consequently, the models need to be re-trained. Unfortunately, it is not feasible to always generate all training data from scratch whenever such a calibration is conducted since it may occur on a daily (or even hourly) basis, while the training of the proposed framework instantiation took approximately 24 hours on a consumer MacBook Pro. As with any AI solution currently, a natural way to speed up the training time would be to utilize supercomputers, e.g., using *High Performance Computing* clusters. Future work may focus on machine learning techniques to adjust existing models to new circumstances that do not involve re-training from scratch.

8.2 Additional Compilation Passes and Figures of Merit

The quantum software ecosystem is moving fast and quantum SDKs frequently release new versions of their toolchains. In addition, many new research methods for quantum circuit compilation are proposed regularly. In order to keep up with this pace, the MQT Predictor has been designed with a unified and modular interface that allows one to easily add and/or update compilation passes. The good news is that the degree of change—whether just another optimization pass has been added or whether all of it changed—does not affect the overall effort. The bad news is that any such change still requires re-training of all RL and ML models. Also here, future work may focus on finding ways to re-train models without starting from scratch.

A similar effort is necessary when changing or adapting the figure of merit. Since both the RL and the ML models optimize for it during training, again, the whole MQT Predictor framework must be re-trained and the persistently stored data must be refreshed.

8.3 Changes in Supported Quantum Devices

New quantum devices become available on a regular basis. Fortunately, adding a new device to the proposed framework is rather easy. It is only necessary to train the respective RL model and to determine the respective evaluation score based on the chosen figure of merit for all training circuits. Afterwards, the ML model can be re-trained in minutes. This is enabled by persistently storing the compiled circuits and evaluation scores of previously conducted compilations as part of the training data generation process. On the contrary, no additional effort is needed to exclude devices—either in case of a temporary downtime due to, e.g., calibration, or a permanent deprecation. These devices can just be masked out without any further effort necessary.

9 CONCLUSION

Using quantum computing as a technology to *just* solve a problem from any kind of application domain is challenging, since a suitable quantum device must be selected to solve a problem encoded as a quantum circuit and the circuit must be compiled accordingly. Deciding which is the best device and how to best compile for a certain application—according to a *figure of merit*—currently requires expert knowledge in quantum computing. So far, especially end-users from application domains are often left unsupported and overwhelmed due to missing automation and tool support. This situation is even worsened, since the result quality heavily fluctuates with the particular choice of options and often makes the difference between a useful result and a useless one due to too much noise.

In this work, we proposed the MQT Predictor framework to automate the selection of suitable target devices and the subsequent compilation. By that, end-users are freed from this task with the goal of preventing a scenario where quantum computers can only be utilized by quantum computing experts—hindering the overall adoption of the technology. The proposed methodology allows learning on the basis of previously conducted compilations of quantum circuits for various devices. For that, the problem is tackled from two angles: Using *Reinforcement Learning*, optimal compilation pass sequences are learned for all available devices—mixing and matching compilation passes from various sources and quantum SDKs. Furthermore, using *Supervised Machine Learning*, a prediction of a quantum device is made for a given quantum circuit according to customizable figures of merit such as, e.g., the *expected fidelity* of the to-be-compiled quantum circuit.

The proposed framework has been implemented using more than 500 quantum circuits from 2 to 90 qubits for seven devices from two qubit technologies and has shown to yield compiled circuits that are within the top-3 in more than 98% of cases while frequently outperforming any tested combination by up to 53% when considering the expected fidelity as the figure of merit. Furthermore, trained and evaluated for *critical depth* as another figure of merit, the performance was within the top-3 in 89% of cases while frequently outperforming any tested combination by up to 400%. The corresponding framework (which is part of the *Munich Quantum Toolkit* (MQT)) including its pre-trained models and classifiers is publicly available on GitHub (https://github.com/cda-tum/mqt-predictor) and as an easy-to-use *Python* package (https://pypi.org/p/mqt.predictor).

These results clearly demonstrate that adapting technologies from the classical realm and applying them to quantum computing can significantly foster the progress of this new and promising technology. Especially since quantum computers are mostly utilized by experts at the moment due to the challenges within the current workflow for realizing quantum application on real machines. With this work, we hope to take a step forward in simplifying the utilization of quantum computers for end-users from application domains and the development of optimized compilers within the community.

ACKNOWLEDGMENTS

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, the State of Upper Austria in the frame of the COMET program, and the QuantumReady project within Quantum Austria (managed by the FFG).

REFERENCES

[1] A. Javadi-Abhari *et al.*, "Quantum computing with Qiskit," 2024. arXiv: 2405.08810.

[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] N. Stamatopoulos *et al.*, "Option Pricing using Quantum Computers," *Quantum*, 2020.

[7] N. Quetschlich *et al.*, "A Hybrid Classical Quantum Computing Approach to the Satellite Mission Planning Problem," in *Int'l Conf. on Quantum Computing and Engineering*, 2023.

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

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

[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] M. Y. Siraichi *et al.*, "Qubit allocation," in *Int'l Symp. on Code Generation and Optimization*, 2018.

[12] N. Quetschlich, L. Burgholzer, and R. Wille, "Predicting Good Quantum Circuit Compilation Options," in *Int'l Conf. on Quantum Software*, 2023.

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

[14] R. Wille *et al.*, "The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing," in *Int'l Conf. on Quantum Software*, 2024, A live version of this document is available at https://mqt.readthedocs.io.

[15] T. Peham *et al.*, "Depth-optimal synthesis of Clifford circuits with SAT solvers," in *Int'l Conf. on Quantum Computing and Engineering*, 2023.

[16] B. Giles and P. Selinger, "Exact synthesis of multiqubit Clifford+T circuits," *Physical Review A*, 2013.

[17] M. Amy *et al.*, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2013.

[18] D. M. Miller, R. Wille, and Z. Sasanian, "Elementary quantum gate realizations for multiple-control Toffoli gates," in *Int'l Symp. on Multi-Valued Logic*, 2011.

[19] A. Zulehner and R. Wille, "One-pass design of reversible circuits: Combining embedding and synthesis for reversible logic," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.

[20] P. Niemann, R. Wille, and R. Drechsler, "Advanced exact synthesis of Clifford+T circuits," *Quantum Information Processing*, 2020.

[21] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the IBM QX architectures," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019.

[22] A. Matsuo and S. Yamashita, "An efficient method for quantum circuit placement problem on a 2-D grid," in *Int'l Conf. of Reversible Computation*, 2019.

[23] R. Wille, L. Burgholzer, and A. Zulehner, "Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations," in *Design Automation Conf.*, 2019.

[24] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for NISQ-era quantum devices," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019.

[25] T. Peham, L. Burgholzer, and R. Wille, "On Optimal Subarchitectures for Quantum Circuit Mapping," *ACM Transactions on Quantum Computing*, 2023.

[26] S. Hillmich, A. Zulehner, and R. Wille, "Exploiting Quantum Teleportation in Quantum Circuit Mapping," in *Asia and South Pacific Design Automation Conf.*, ACM, 2021.

[27] A. Zulehner and R. Wille, "Compiling SU(4) quantum circuits to IBM QX architectures," in *Asia and South Pacific Design Automation Conf.*, 2019.

[28] R. Wille and L. Burgholzer, "MQT QMAP: Efficient quantum circuit mapping," in *Int'l Symp. on Physical Design*, 2023.

[29] L. Schmid, S. Park, and R. Wille, "Hybrid Circuit Mapping: Leveraging the Full Spectrum of Computational Capabilities of Neutral Atom Quantum Computers," in *Design Automation Conf.*, 2024.

[30] W. Hattori and S. Yamashita, "Quantum circuit optimization by changing the gate order for 2D nearest neighbor architectures," in *Int'l Conf. of Reversible Computation*, 2018.

[31] G. Vidal and C. M. Dawson, "Universal quantum circuit for two-qubit transformations with three controlled-NOT gates," *Physical Review A*, 2004.

[32] T. Itoko *et al.*, "Quantum circuit compilers using gate commutation rules," in *Asia and South Pacific Design Automation Conf.*, 2019.

[33] D. Maslov *et al.*, "Quantum circuit simplification and level compaction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2008.

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

[35] Y. Kharkov *et al.*, "Arline benchmarks: Automated benchmarking platform for quantum compilers," 2022. arXiv: 2202.14025.

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

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

[38] S. Dangwal *et al.*, "Clifford assisted optimal pass selection for quantum transpilation," 2023. arXiv: 2306.15020.

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

[40] M. Salm *et al.*, "Optimizing the prioritization of compiled quantum circuits by machine learning approaches," in *Service-Oriented Computing*, Cham: Springer International Publishing, 2022.

[41] M. Salm *et al.*, "How to select quantum compilers and quantum computers before compilation," in *Int'l Conf. on Cloud Computing and Services Science*, 2023.

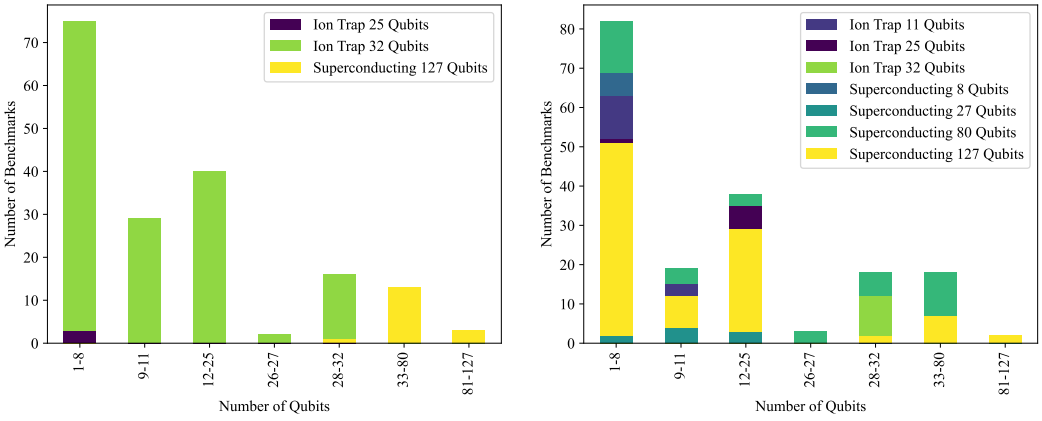
[42] A. Paler *et al.*, "Machine learning optimization of quantum circuit layouts," *ACM Transactions on Quantum Computing*, 2023.

[43] B. Mete, M. Schulz, and M. Ruefenacht, "Predicting the optimizability for workflow decisions," in *Int'l Workshop on Quantum Computing Software*, 2022.

[44] T. Fösel *et al.*, "Quantum circuit optimization with deep reinforcement learning," 2021. arXiv: 2103.07585.

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

[46] S. van der Linde *et al.*, "qgym: A gym for training and benchmarking RL-based quantum compilation," 2023. arXiv: 2308.02536.
[47] H. Leather and C. Cummins, "Machine learning in compilers: Past, present and future," in *Forum on Specification and Design Languages*, 2020.
[48] A. H. Ashouri *et al.*, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, 2018.
[49] F. Agakov *et al.*, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization*, 2006.
[50] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004.
[51] M. Trofin *et al.*, "MLGO: A machine learning guided compiler optimizations framework," 2021. arXiv: 2101.04808.
[52] Q. Huang *et al.*, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," 2020. arXiv: 2003.00671.
[53] A. Haj-Ali *et al.*, "Neurovectorizer: End-to-end vectorization with deep reinforcement learning," 2019. arXiv: 1909.13639.
[54] T. Tomesh *et al.*, "Supermarq: A scalable quantum benchmark suite," 2022. arXiv: 2202.11045.
[55] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, 1957.
[56] I. H. Sarker, "Machine Learning: Algorithms, Real-World Applications and Research Directions," *SN Computer Science*, 2021.
[57] G. Brockman *et al.*, "OpenAI Gym," 2016. arXiv: 1606.01540.
[58] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing," *Quantum*, 2023, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.
[59] J. Schulman *et al.*, "Proximal policy optimization algorithms," 2017. arXiv: 1707.06347.
[60] A. Raffin *et al.*, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, 2021.
[61] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.
[62] L. Breiman, "Random forests," *Machine learning*, 2001.
[63] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of statistics*, 2001.
[64] L. Breiman *et al.*, "Classification and Regression Trees." 1984.
[65] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, 1967.
[66] S. Haykin, "Neural Networks: A Comprehensive Foundation." Prentice Hall PTR, 1998.
[67] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, 1995.
[68] 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.
[69] 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.
[70] A. W. Cross *et al.*, "Open Quantum Assembly Language," 2017. arXiv: 1707.03429.



(a) Expected fidelity.

(b) Critical depth.

Fig. 9. Device distributions.

A APPENDICES

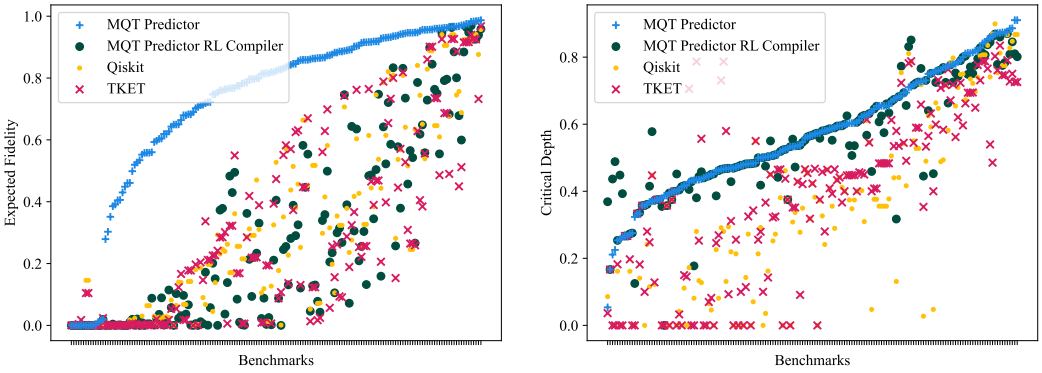
A.1 Device Distribution

To better understand the problem of selecting the best device, the underlying ground truth data used to train the MQT Predictor is further investigated. To this end, the frequency with which each device is selected as the best performing one is investigated. For that, each benchmark is compiled for all supported devices using the respectively trained RL compilers and the one leading to the best result is determined. The results are visualized in Fig. 9. Furthermore, the benchmarks are grouped by their number of qubits such that all benchmarks within one group can be executed on the same number of devices—leading to seven groups since currently seven devices are supported with a decreasing number of available devices for an increasing number of qubits.

When using the expected fidelity as the figure of merit, the results are rather distinct as visualized in Fig. 9a. For most benchmarks, the ion trap device with 32 qubits results in the best performance. Therefore, the rule of thumb known by quantum computing experts—taking an ion trap device when possible—has been confirmed. However, when all ion trap devices' capacities are exceeded, the largest superconducting device is best even when a smaller device would have been available as well. This showcases that another quantum expert rule of thumb—taking always the smallest but fitting device—cannot be confirmed by our experimental data.

When using the critical depth as the figure of merit, the device distribution is significantly more diverse as visualized in Fig. 9b[5]. Here, all seven devices are chosen at least sometimes and there is no apparent correlation between the best performing device and the benchmark groups since most devices result in the best performance for more than just one group. Consequently, no clear guidance can be derived from that—underlining the importance of software tools to guide this decision automatically.

[5]The benchmarks used for the evaluation differ between both figures of merit and, therefore, the number of benchmarks in each group differs as well.



(a) Expected fidelity.

(b) Critical depth.

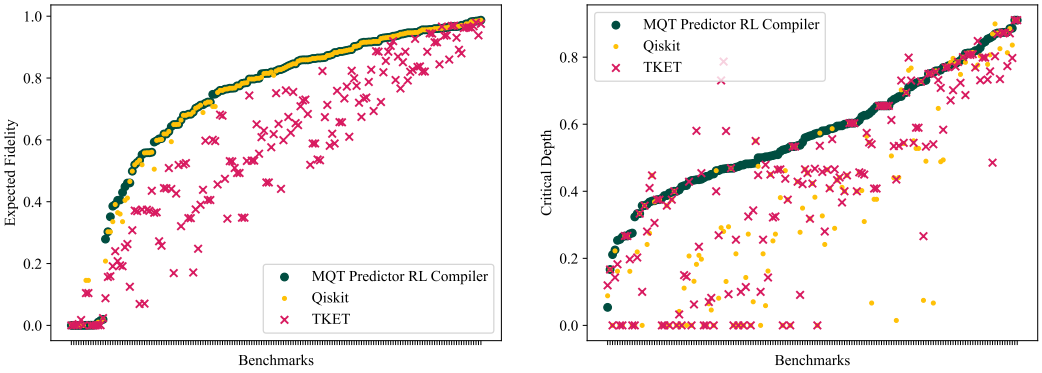
Fig. 10. Isolated evaluation of the device selection using ML.

A.2 Detailed Evaluation: Device Selection Using ML

The MQT Predictor framework automatically selects the most promising device and compiles accordingly. To assess the effect each of the two tasks has, they are evaluated in an isolated fashion—starting with the device selection and its respective results shown in Fig. 10. To this end, the largest device (superconducting device with 127 qubits) is used as the default device to fit all benchmarks. Then, the benchmarks are compiled for it using Qiskit, TKET, and the MQT Predictor RL compiler and, similarly to Fig. 6 and Fig. 7, the benchmarks are sorted by their MQT Predictor score for better readability. The resulting performances are then compared against the full MQT Predictor approach including the device selection.

For the expected fidelity, the results are visualized in Fig. 10a—showing a distinct gap between the results of the full MQT Predictor approach including the device selection compared to all three baselines using the default device. This highlights the importance of selecting the most suitable device, since that can make the difference between a successful execution and obtaining completely random results. Furthermore, it is interesting to see that there is no clear winner among Qiskit, TKET, and the MQT Predictor RL compiler.

In the case of critical depth, the situation is different as visualized in Fig. 10b since the full MQT Predictor approach less frequently leads to the best result. This suggests that the default device is a good choice for critical depth and the influence between it and the best one is smaller than it is the case for the expected fidelity. Also, this matches with Fig. 9b since often the largest device is the most suitable in the ground truth data. Furthermore, the MQT Predictor RL compiler approach outperforms both Qiskit and TKET in most cases—showcasing, that both are not optimizing for critical depth but optimize for a different implicit underlying criterion closer to the expected fidelity.



(a) Expected fidelity.

(b) Critical depth.

Fig. 11. Isolated evaluation of the compilation using RL.

A.3 Detailed Evaluation: Compilation Using RL

To assess the compilation quality of the RL part of the MQT Predictor framework, it is evaluated in a similar and isolated fashion as the device selection described in Appendix A.2. To this end, the device is set to be the one predicted for each benchmark and the effect of the different compilers is evaluated in Fig. 11. For that, the same compilers as before are used: Qiskit, TKET, and the MQT Predictor RL compiler—while, in this case, the latter corresponds to the full MQT Predictor approach because the default device is the predicted one. Therefore, this evaluation setup results in plots that show a subset of the data points plotted in Fig. 6 respectively Fig. 7 in which the compilers are evaluated for all seven devices while here only the predicted one is used. The benchmarks are sorted by their MQT Predictor RL compiler scores for better readability (which, in this case, corresponds to the scores of the full MQT Predictor approach).

For the expected fidelity visualized in Fig. 11a, the MQT Predictor RL compiler outperforms TKET and Qiskit, although there is only a small performance gap to the latter since Qiskit performs better than TKET for the predicted device. Compared to Fig. 10a where the default device was the largest superconducting one, this was not the case. Therefore, apparently the compilation quality significantly depends on the device—either by Qiskit being better for the predicted one (which is often an ion trap one), TKET being worse, or both. Nevertheless, the plot shows the (small) improvement of the MQT Predictor compilation—potentially growing when including more compilation passes from further providers.

In the case of the critical depth visualized in Fig. 11b, the performance differences between Qiskit and TKET are not as distinct as before. Furthermore, the performance difference of the MQT Predictor RL compiler compared to both of them is even larger—highlighting the impact it has.