Compiler Optimization for Quantum Computing Using Reinforcement Learning

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 quantum computing application, once encoded as a quantum circuit, must be compiled before being executable on a quantum computer. Similar to classical compilation, quantum compilation is a sequential process with many compilation steps and numerous possible optimization passes. Despite the similarities, the development of compilers for quantum computing is still in its infancy—lacking mutual consolidation on the best sequence of passes, compatibility, adaptability, and flexibility. In this work, we take advantage of decades of classical compiler optimization and propose a *reinforcement learning* framework for developing optimized quantum circuit compilation flows. Through distinct constraints and a unifying interface, the framework supports the combination of techniques from different compilers and optimization tools in a single compilation flow. Experimental evaluations show that the proposed framework— set up with a selection of compilation passes from IBM's Qiskit and Quantinuum's TKET—significantly outperforms both individual compilers in 73% of cases regarding the expected fidelity. The framework is available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT).

I. Introduction

In classical computing, each *Central Processing Unit* (CPU) comes with its own *Instruction Set Architecture* (ISA)—describing the native operations the CPU can process. High-level programming languages such as *C++*, *Rust*, or *Go* are used to develop programs in a platform-agnostic way. These high-level descriptions are then *compiled* to a specific device's ISA by a *compiler*. Prominent examples are *LLVM* [1] or *GCC* for C++, *rustc* for Rust, and *gc* or *gccgo* for Go. In each of these compilers, a plethora of passes for analyzing and transforming the source code is available. The "quality" of the code generated by compilers heavily depends on the selection of the individual passes applied during the compilation process as well as their execution order. Furthermore, one particular combination is highly unlikely to be the most effective sequence for every program on a particular machine. Even nowadays, after decades of classical compiler research, determining good sequences of compilation passes turns out to be a challenging task. In the literature, this problem is commonly referred to as *Phase Ordering*.

Research on the phase ordering problem already started roughly four decades ago [2]–[4] and is still very actively pursued. In the recent years, *autotuning* [5] (trying out different compilation parameters and evaluating the results with respect to a particular metric) and machine learning [6] have shown promising results. A particularly promising branch of research is the application of *Reinforcement Learning* (RL, [7], [8]) for this purpose. In [9], the authors show that their RL-based approach to phase ordering outperforms LLVM's optimization scheme. Reinforcement learning has also successfully been applied for inlining optimization [10] and vectorization [11].

The compilation of quantum circuits/programs follows a very similar scheme as the compilation of classical programs. All major quantum SDKs (such as IBM's Qiskit [12], Quantinuum's TKET [13], AWS Braket [14], or Microsoft's Azure Quantum [15]) implement some sort of compilation flow for making circuits executable on particular devices (see, e.g., [16]). However, while research on classical compilers has been conducted for decades, compilers for quantum computing are still in early development. This manifests in several ways:

1) There is no mutual consolidation on what the best methods and combinations thereof are.
2) It is not straightforward to mix and match different options from various frameworks—creating something similar to a vendor lock-in.
3) Major SDKs can be quite slow in adopting the newest research findings—effectively restraining users from taking advantage of state-of-the-art methods.
4) Users can hardly customize the optimization goal or the target metric for the compilation process. Instead, they have to rely on predefined combinations of compilation passes provided by the SDKs—typically in the form of various optimization levels.

As a consequence, we are quickly heading towards the same problems as in classical compiler development and, despite the similarities, techniques from the classical realm cannot be straightforwardly applied in the quantum realm due to the inherent differences of both computational paradigms.

In this work, we aim to mitigate the drawbacks discussed above. Instead of trying to reinvent the wheel, the solution proposed in this work builds upon the decades of research on compiler optimization in the classical realm. To this end, we adapt an established classical technique to the quantum domain—modeling the compilation flow as a *Markov Decision Process* (MDP, [17]) and applying *Reinforcement Learning* to learn the best order in which the different compilation passes and optimizations can be applied.

The resulting framework (which is available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT)) is platform agnostic and easily extendable with further compilation methods from different SDKs and software tools. This is accomplished by defining distinct constraints that can be efficiently checked for each state in the process and a unifying interface for each compilation step. In contrast to slowly-adapting compilers in major SDKs, this allows to develop optimized compilation flows that can keep up with the fast-paced development and research in this domain.

Experimental evaluations demonstrate the feasibility of the proposed approach. Compared to the compilation flows offered by IBM's Qiskit and Quantinuum's TKET, the proposed framework—set up with a selection of compilation passes from both tools—is shown to outperform both individual compilers with respect to expected fidelity, critical depth, and their combination in 73%, 84%, 75% of cases.

The rest of this paper is structured as follows. Section II briefly reviews quantum circuit compilation. Then, Section III describes how compiler optimization for quantum circuits can be modeled as an MDP and how it can be tackled with reinforcement learning. Based on that, Section IV provides details on the implementation of the proposed framework and evaluates its performance. Finally, Section V concludes the paper.

II. BACKGROUND: QUANTUM CIRCUIT COMPILATION

Quantum programs are commonly described in the form of *quantum circuits*, i.e., sequences of quantum operations (also called quantum gates) acting on the qubits of a quantum system.

Example 1. *A simple quantum circuit operating on three qubits (q_0, q_1, q_2) is shown in Fig. 1a. It contains four different kinds of gates: Two NOT gates (denoted as* X*), one Hadamard gate (denoted as* H*), one Z-rotation gate (denoted as R_z), and three two-qubit controlled-NOT or CNOT gates (with their control and target qubits denoted as • and ⊕, respectively).*

Executing such programs on an actual *Quantum Processing Unit* (QPU) requires *compilation* conducted by *compilers*. They usually first employ device-independent optimizations to reduce the complexity of the original circuit description. Examples of such optimizations include gate-cancellation, gate-commutation, or high-level synthesis routines, e.g., proposed in [18]–[22].

Example 2. *Consider again the circuit shown in Fig. 1a. Employing device-independent optimizations would, e.g., lead to the cancellation of the consecutive X gates on q_2 (highlighted by the green box) since $X \cdot X = I$.*

After these initial optimizations, QPU-specific circuits are generated. To this end, each quantum processor offers a particular set of *native gates*, i.e., operations that can be directly executed on the device. Consequently, all non-native gates need to be *synthesized* using these native gates for the circuit to be executable. This synthesis step creates opportunities for further optimizations.
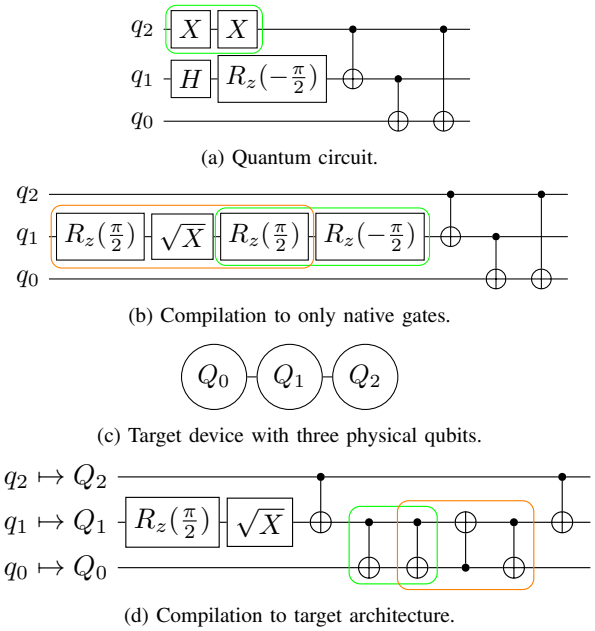

(a) Quantum circuit.

(b) Compilation to only native gates.

(c) Target device with three physical qubits.

(d) Compilation to target architecture.

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

Example 3. *Recent devices offered by IBM provide a native gate-set consisting of R_z, $\sqrt{X}$,* X, *and* CNOT *gates. To execute the circuit from Fig. 1a on such a device, e.g., the Hadamard gate needs to be expressed in terms of these gates. A suitable decomposition into two $R_z(\frac{\pi}{2})$ and a $\sqrt{X}$ gate is shown in Fig. 1b (highlighted by the orange box). Incidentally, one of the $R_z(\frac{\pi}{2})$ gates can, then, be cancelled with the existing $R_z(-\frac{\pi}{2})$ gate (highlighted by the green box).*

Some QPUs (e.g., those based on superconducting qubits or neutral atoms) only feature a limited connectivity between their qubits, so that multi-qubit gates can only be applied to qubits that are connected on the device. In these cases, a *mapping* step is required. Mapping is frequently split into *layouting*, i.e., assigning the circuit's logical qubits to the device's physical qubits, and *routing*, i.e., ensuring the connectivity constraints are satisfied by dynamically changing the assignment of logical to physical qubits. Again, optimizations may be applied after mapping the circuit to reduce the induced overhead.

Example 4. *Assume the circuit from Fig. 1b shall be executed on an IBM device with three physical qubits (Q_0, Q_1, Q_2) arranged in a line (as depicted in Fig. 1c), i.e., only interactions between Q_0 and Q_1 as well as Q_1 and Q_2 are possible. Then, no layout exists that makes this circuit executable right away since the circuit contains interactions between all pairs of qubits. Thus, a SWAP gate (eventually realized as a sequence of three CNOT gates highlighted by the orange box) needs to be inserted during the routing step to make the circuit executable (as shown in Fig. 1d). In this case, some of the induced overhead can be reduced by cancelling two CNOT gates (highlighted by the green box).*
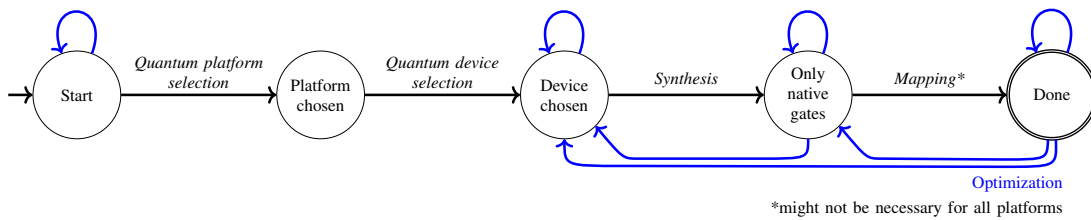
Fig. 2. Quantum compilation flow modeled as an MDP.

With the number of different synthesis, mapping, and optimization techniques for quantum circuits rapidly increasing, the landscape of compiler passes is becoming more and more complex—leading to several obstacles as discussed in Section I. In this work, we propose a framework that aims to overcome these obstacles.

III. PROPOSED FRAMEWORK

Fortunately, the underlying problem is not new per-se, since classical compilers have seen a similar trend in the past and lots of research has been conducted to find efficient solutions there. In the following, we describe how a classically-inspired reinforcement learning technique can be adapted to quantum circuit compilation. To this end, we first describe how to programmatically model quantum circuit compilation in order to, then, apply reinforcement learning to try to learn the best sequence of compilation passes depending on characteristics of the initial circuit—resulting in an *optimized compiler* for quantum circuits.

A. Modeling as a Markov Decision Process

The quantum circuit compilation flow reviewed above can be seen as a sequential procedure that takes an initial (high-level) quantum circuit and, step by step, applies certain actions/transformations to eventually arrive at a circuit representation that can be executed on a target device. Such a procedure can be formally modeled as a *Markov Decision Process* (MDP, [17]). To this end, it is split into states and corresponding actions that can be performed in each of the states. The model proposed in this work is illustrated in Fig. 2—with states being denoted as circles and actions as arrows. In the following, the process with its respective actions is described one by one.

Initially, the process starts in the "Start" state and stores the original quantum circuit to be compiled. At this point, there are two possible types of actions. On the one hand, device-independent *optimizations* can be applied to the circuit (indicated by the blue arrow). As reviewed in Section II, this includes techniques such as single-qubit gate fusion, gate cancellation and commutation, as well as high-level synthesis methods, e.g., proposed in [18]–[22]. After such an optimization, the process remains in the same state as before, but now stores a new optimized quantum circuit. On the other hand, the process can proceed to a different state by *Quantum platform selection* actions, e.g., choosing *IBM*, *Rigetti*, or *IonQ*—effectively fixing the targeted native gate-set. In the classical domain, this roughly corresponds to deciding, e.g., between *x86* and *ARM* architectures.

Once a platform has been chosen (i.e., the process is in the "Platform chosen" state), only a single type of action can be applied. Namely, the *Quantum device selection* actions offered by the selected platform. This step fixes the number of qubits and their topology, i.e., which qubits can interact with each other. Depending on the choice of the platform, a varying range of devices with different characteristics is available. At the time of writing, IBM for example offers 23 different quantum computers that each vary in their qubit count, their quantum volume, and their *Circuit Layer Operations Per Second* (CLOPS). In the classical domain, this roughly corresponds to targeting a specific CPU model family, such as *Intel Alder Lake* or *AMD Zen 4*.

After a particular device has been selected (i.e., the process is in the "Device chosen" state), the device-dependent compilation begins. The goal of the remaining steps is to produce a circuit that can be executed on the chosen device. To this end, the circuit needs to satisfy two conditions:

1) it must only use gates native on the selected platform, and
2) it must conform to the topology of the selected device.

While the first condition can be ensured by using *Synthesis* methods, e.g., as proposed in [23]–[26], the second condition is realized by corresponding *Mapping* methods, e.g., as proposed in [27]–[29]. Both of these tasks are highly non-trivial and there is a vast variety of methods available in major quantum SDKs as well as individual research tools. At the time of writing, IBM's Qiskit alone offers four different layout and five routing algorithms—leading to 20 different mapping schemes. Once both conditions are satisfied, the process is in the "Done" state, i.e., the circuit is executable. Optimizations may be applied in any state during the device-dependent compilation. Depending on whether the resulting circuit satisfies none, the native-gates, or both constraints, the process respectively continues in the "Device chosen", "Only native gates", or "Done" state (again, indicated by the blue arrows).

Overall, the formulation described above leads to a modular framework with two characteristic traits:

1) Each state has *distinct constraints* that can be efficiently checked, e.g., whether the circuit only contains certain native gates or only acts on qubits connected on the device. Thus, it is straight-forward to determine the state after a certain action has been applied.
2) All actions have a *unified interface*, i.e., all of them use a quantum circuit as the main representation for their input and output. Moreover, the same format is used independent of the SDK or tool the actions originate from.

As a consequence, it is possible to mix and match compiler passes from different SDKs and tools and integrate them into a single *optimized* compiler. Such a framework also allows to mitigate vendor lock-ins and can help to keep up with the rapid pace of the field by allowing one to quickly integrate new methods into the overall compilation flow.

B. Compiler Optimization Using Reinforcement Learning

After a framework as described above is set up, it can be used as the basis for learning the best sequence of actions (i.e., compilation passes) depending on characteristics of the initial circuit. Motivated by its success in classical compiler optimization [9]–[11], we propose to apply reinforcement learning for this task. RL-methods aim to learn an *action policy* that describes which action to apply based on certain *observations* on the current state such that some cumulative *reward function* is maximized. In contrast to supervised machine learning methods, no training data must be provided. Instead, only an environment describing the underlying MDP (i.e., the set of states and corresponding actions), the features used as observations, and a corresponding reward function needs to be provided. Based on the rewards for certain actions in certain states, the action policy improves through the training process and learns how to maximize the expected cumulative reward.

In the proposed framework, observations are realized by extracting certain features from the quantum circuits in each state. In addition, a sparse reward function is used that reflects whether a final state has been reached and, then, quantifies the "quality" of the resulting circuit, e.g., in terms of the resulting gate count, circuit depth, or expected fidelity. This flexibility in the target metric allows one to design specialized compilers for dedicated use cases. After successful training, the learned action policy can be used as an *optimized compiler* for any quantum circuit.

C. Related Work

Machine learning has been successfully applied to quantum circuit design/compilation in the past. In [30], the authors propose a machine learning based layout and routing scheme and claim to be on-par with state-of-the-art methods. Another possible application is given in [31], where the graph structure of quantum circuits is used to predict their expected fidelities for certain devices considering their respective noise characteristics. An idea related to the one proposed in this work, which follows a more coarse-grained approach, has been demonstrated in [32]. There, a supervised machine learning model is used to predict good combinations of devices and existing compilation flows. In [33], RL has been applied to learn a strategy for applying individual gate transformation rules to optimize quantum circuits.

While these works either optimize one specific type of action or predict compilation options on a high abstraction level within one quantum SDK, the framework proposed in this work results in fine-grained steps that allow one to stitch together methods from multiple SDKs with a customizable optimization objective into a fully-fledged compilation flow.

IV. FEASIBILITY STUDY

In the following, the feasibility of the proposed framework is demonstrated. To this end, we first describe the particular instantiation that has been developed as part of this work. Then, the performance of the resulting optimized compiler is evaluated and compared against the state-of-the-art compilers available in IBM's Qiskit [12] and Quantinuum's TKET [13]. All developments and results are made publicly available on GitHub (https://github.com/cda-tum/MQTPredictor) as part of the Munich Quantum Toolkit (MQT).

A. Instantiation

The basis of the framework is formed by the open-source *OpenAI Gym* [34] library, which is used to set up an environment for the MDP described in Section III (as illustrated in Fig. 2). In this regard, the main endeavor is the selection and integration of all compilation passes that should be considered as actions in the learning process. As modeled in Fig. 2, we distinguish five kinds of actions: Quantum platform selection, Quantum device selection, Synthesis, Mapping, and Optmization. Due to the unified action interface of the proposed framework described earlier, methods from more than one SDK or toolkit can be incorporated. For each kind of action, suitable representatives from IBM's Qiskit (version 0.39.2) and/or Quantinuum's TKET (version 1.8.1) have been chosen—leading to the following list of available actions.

- *Platforms* and corresponding *Devices*:
    - IBM (superconducting): *ibmq_montreal* (27 qubits) and *ibmq_washington* (127 qubits)
    - Rigetti (superconducting): *Aspen-M-2* (80 qubits)
    - IonQ (trapped-ions): *Harmony* (11 qubits)
    - OQC (superconducting): *Lucy* (8 qubits)
- *Synthesis*: Qiskit's *BasisTranslator*
- *Mapping*: 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*
    - TKET's *PeepholeOptimise2Q*
    - TKET's *CliffordSimp*
    - TKET's *FullPeepholeOptimise*
    - TKET's *RemoveRedundancies*

Inspired by the feature selection and demonstrated relevance in [32], the observations used to guide the reinforcement learning agent are based on seven features—namely 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 [35]. For the learning itself, three different reward functions have been implemented:

1) *Expected fidelity*: provides an estimate on the reliability of the circuit based on characteristics of the chosen device. A fidelity of 1 indicates an error-free result, while a fidelity of 0 indicates that the result is completely wrong.
2) *Critical Depth*: describes how many two-qubit gates in a quantum circuit lie along the critical path and contribute to the overall circuit depth. A critical depth close to 1 indicates a highly sequential quantum circuit. Since, a lower value is desirable, the corresponding reward function is defined as $1 -$ critical depth.
3) *Combination*: adds up both mentioned reward functions (divided by two to result in a value between 0 and 1).

Finally, the *Proximal Policy Optimization (PPO)* algorithm [36] provided by *Stable-Baselines3* [37] is used for the reinforcement learning.

B. Results

To evaluate the performance of the framework instantiation described above, three models—one for each reward function—were trained on a selection of 200 circuits ranging from 2 to 20 qubits taken from the *target-independent* level of the MQT Bench library [38]. In total, $100,000$ time steps were executed during the training for each reward function with runtimes of the order of hours. Then, the trained models were evaluated on the same circuits and the resulting quality in terms of the respective reward function was measured. To put this performance into perspective, all circuits are additionally compiled to the *ibmq_washington* device using Qiskit and TKET with the highest optimization level ($O3$ for Qiskit and $O2$ for TKET).

The results are summarized in Fig. 3. To this end, Fig. 3a, Fig. 3b, and Fig. 3c, respectively show the distribution of *absolute* reward differences of the proposed approach compared to Qiskit and TKET with respect to the three quality metrics. An x-value larger than zero indicates a positive improvement, e.g., a value of 0.2 for the fidelity means that the proposed approach managed to boost the fidelity by 20%. In an ideal scenario, all results would accumulate on the right-side of the zero-line.

The results demonstrate that, in the vast majority of cases, this is the case, i.e., the proposed method produces circuits of equal or better quality. In particular, the proposed method outperforms the Qiskit/TKET compiler in $73\%/80\%$, $84\%/86\%$, and $75\%/78.5\%$ of cases regarding expected fidelity, critical depth, and their combination.

The advantage of the proposed method becomes even clearer when looking at the average reward difference per benchmark algorithm (subsuming several instances with varying amounts of qubits) as shown in Fig. 3d, Fig. 3e, and

TABLE I
COMPARISON OF REINFORCEMENT LEARNING MODELS.

Model trained for...	Average result for...		
	Fidelity	Critical depth	Combination
Fidelity	**0.48**	0.27	0.37
Critical depth	0.18	**0.47**	0.33
Combination	0.45	0.33	**0.39**

Fig. 3f respectively. There, y-values higher than zero indicate a net gain in quality. Again, in the vast majority of cases, a substantial improvement can be observed. Compared to Qiskit/TKET, an average *absolute* increase in fidelity, critical depth, and their combination of $4.9\%/10.7\%$, $22.6\%/22.8\%$, and $5.5\%/8.5\%$ is achieved.

To further underline the proposed framework's ability to adapt to different target metrics, we additionally calculated the average reward for each metric by all three trained models—not only the one trained for the respective metric. The resulting numbers are denoted in Table I. They confirm that the model trained for a particular purpose indeed results in the compiled circuits with the highest rewards for all three cases.

V. CONCLUSIONS

Quantum circuit compilation flows are becoming more and more complex. This leads to a similar situation observed in classical compilation where there are so many options and it is unclear which options to choose and in which order to apply them. In this work, we proposed a framework that builds on the decades of research on classical compiler optimization to overcome these obstacles. To this end, the quantum circuit compilation flow was modeled as a Markov Decision Process with distinct constraints for each state and a unified interface for all actions. Based on that, reinforcement learning was applied to derive a complete compilation flow out of actions from multiple SDKs and toolkits. Experimental evaluations on the developed framework and three different optimization objectives demonstrate the advantage of the proposed method over Qiskit's and TKET's most optimized compilation sequences and the general feasibility of automatically creating compilations flows targeting a customizable objective. In 73%, 84%, 75% of the cases, the trained models outperformed Qiskit/TKET with an average *absolute* improvement of $4.9\%/10.7\%$, $22.6\%/22.8\%$, and $5.5\%/8.5\%$ regarding expected fidelity, critical depth, and their combination respectively. These results clearly demonstrate that adapting techniques from the classical realm can, indeed, yield substantial improvements in the quantum realm. By making the developed framework publicly available as open-source, we hope to lay the foundation for further exploration of this important area of research.
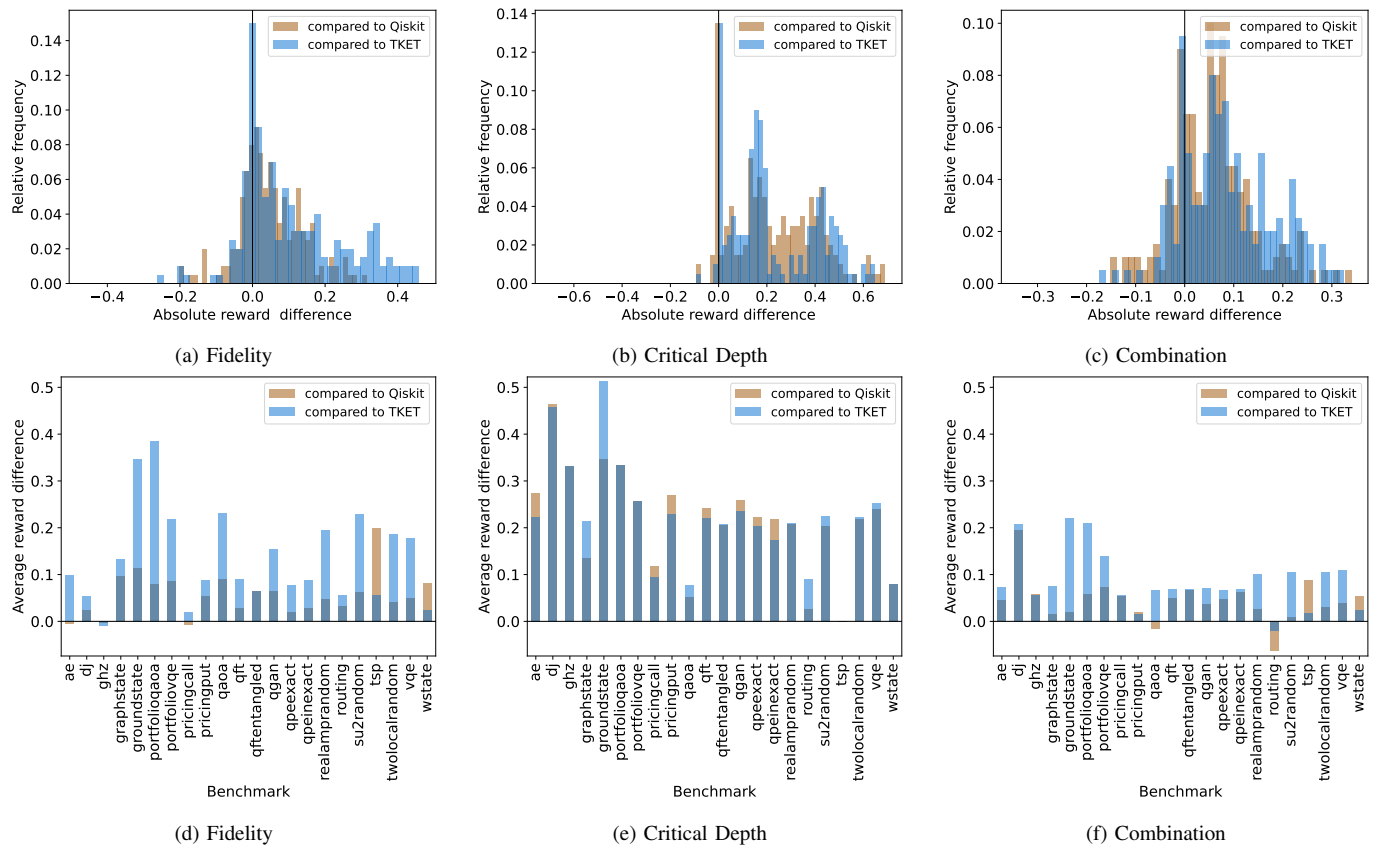
Fig. 3. Experimental evaluation.

REFERENCES

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

[2] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *Workshop on Microprogramming*, 1982.

[3] D. B. Loveman and R. A. Faneuf, "Program optimization - theory and practice," *SIGPLAN Not.*, 1975.

[4] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Trans. Program. Lang. Syst.*, 1997.

[5] A. H. Ashouri *et al.*, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, 2018.

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

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, 1996.

[8] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

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

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

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

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

[13] S. Sivirajah *et al.*, "TKET: A Retargetable Compiler for NISQ devices," *Quantum Science and Technology*, 2020.

[14] *Amazon Braket Python SDK*, Amazon Web Services, 2022. [Online]. Available: https://github.com/aws/amazon-braket-sdk-python.

[15] *Azure Quantum documentation*, Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/quantum/.

[16] R. Wille, R. Van Meter, and Y. Naveh, "IBM's Qiskit tool chain: Working with and developing for real quantum computers," in *Design, Automation and Test in Europe*, 2019.

[17] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, 1957.

[18] 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.

[19] P. Niemann, R. Wille, and R. Drechsler, "Advanced exact synthesis of clifford+t circuits," *Quantum Information Processing*, 2020.

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

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

[22] D. Maslov, G. Dueck, D. Miller, and C. Negrevergne, "Quantum circuit simplification and level compaction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2008.

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

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

[25] 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.

[26] S. Schneider, L. Burgholzer, and R. Wille, "A SAT encoding for optimal Clifford circuit synthesis," in *Asia and South Pacific Design Automation Conf.*, 2023.

[27] M. Y. Siraichi, V. F. dos Santos, S. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Int'l Symp. on Code Generation and Optimization*, 2018.

[28] 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.

[29] 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.

[30] A. Paler, L. M. Sasu, A.-C. Florea, and R. Andonie, "Machine learning optimization of quantum circuit layouts," *ACM Transactions on Quantum Computing*, 2022.

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

[32] N. Quetschlich, L. Burgholzer, and R. Wille, *Predicting good quantum circuit compilation options*, 2022. arXiv: 2210.08027.

[33] T. Fösel, M. Y. Niu, F. Marquardt, and L. Li, *Quantum circuit optimization with deep reinforcement learning*, 2021. arXiv: 2103.07585.

[34] G. Brockman *et al.*, *OpenAI Gym*, 2016. arXiv: 1606.01540.

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

[36] J. Schulman *et al.*, *Proximal policy optimization algorithms*, 2017. arXiv: 11707.06347.

[37] A. Raffin *et al.*, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, 2021.

[38] 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.