MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing

Nils Quetschlich[1], Lukas Burgholzer[1], and Robert Wille[1,2]

[1]Chair for Design Automation, Technical University of Munich, Germany
[2]Software Competence Center Hagenberg GmbH (SCCH), Austria

Quantum software tools for a wide variety of design tasks on and across different levels of abstraction are crucial in order to eventually realize useful quantum applications. This requires practical and relevant benchmarks for new software tools to be empirically evaluated and compared to the current state of the art. Although benchmarks for specific design tasks are commonly available, the demand for an overarching cross-level benchmark suite has not yet been fully met and there is no mutual consolidation in how quantum software tools are evaluated thus far. In this work, we propose the *MQT Bench* benchmark suite (as part of the *Munich Quantum Toolkit*, MQT) based on four core traits: (1) cross-level support for different abstraction levels, (2) accessibility via an easy-to-use web interface (`https://www.cda.cit.tum.de/mqtbench`) and a Python package, (3) provision of a broad selection of benchmarks to facilitate generalizability, as well as (4) extendability to future algorithms, gate-sets, and hardware architectures. By comprising more than 70,000 benchmark circuits ranging from 2 to 130 qubits on four abstraction levels, MQT Bench presents a first step towards benchmarking different abstraction levels with a single benchmark suite to increase comparability, reproducibility, and transparency.

1 Introduction

Quantum computing has gained a lot of attention due to its promising applications and the advances in quantum computing hardware. But still, designing quantum algorithms often means to manually implement quantum circuits on the gate-level—similar to assembly language in classical computing. Hence, there is an urgent need to enhance the workflow of designing and testing quantum algorithms. Without software tools that aid in the design of quantum algorithms, the underlying hardware may not be efficiently utilized. Thus, researchers and developers have already proposed tools for various design tasks, such as quantum circuit simulation [1]–[13], compilation [14]–[34], or verification [35]–[50]. This already led to comprehensive quantum circuit design flows realized through toolkits such as IBM's Qiskit [51], Google's Cirq [52], or Rigetti's Forest [53]—in addition to numerous tools and methods developed by other researchers and engineers in the field.

Generally, these software tools operate on and across different levels of abstraction and all tackle computationally hard problems. As a result, they have to compromise between resource demands and result quality. Usually, when a new software tool is proposed, its performance is empirically evaluated. The question how to evaluate the performance of a software tool is very challenging and not yet fully answered—especially for tools solving NP-complete problems. The most common approach is to run certain problem instances, so-called benchmarks, and compare the performance of the newly proposed method against state-of-the-art methods regarding a certain characteristic, e.g., run-time or solution quality.

Nils Quetschlich: nils.quetschlich@tum.de
Lukas Burgholzer: lukas.burgholzer@tum.de
Robert Wille: robert.wille@tum.de

Currently, this benchmarking is conducted using a wide variety of different benchmark suites—each with a specific focus. While some benchmarks suites, e.g., [54], [55], provide benchmarks on higher abstraction levels, other benchmark suites focus on lower abstraction levels, e.g., [56], [57]. Although the intended target levels are properly covered, the demand for a cross-level benchmark suite is not fully met yet. As another consequence, there is no mutual consolidation which benchmarks to use for empirical evaluations yet—leading to lower comparability, reproducibility, and transparency.

In this paper, we propose *MQT Bench*—the benchmark suite from the *Munich Quantum Toolkit* (MQT) which explicitly aims to address those drawbacks. It aims at providing a first step towards benchmarking the whole quantum software stack with a single benchmark suite by offering the same benchmark algorithms on different levels of abstraction. To realize such a benchmark suite, several challenges have to be tackled:

- Distinct requirements on different levels: Benchmarks have to fulfill certain requirements depending on the abstraction level, e.g., only gates of a device's native gate-set are allowed.
- Accessibility: To foster adoption of the benchmark suite, it needs to be as easy to use as possible.
- Generalizability: Providing a comprehensive set of benchmark algorithms to cover as many use cases as possible.
- Extendability: Future algorithms, gate-sets, and architectures should be easy to integrate.

MQT Bench has been developed with these challenges in mind and is based on four core traits:

1. Cross-level benchmarking: Provision of benchmarks on four abstraction levels.
2. Accessibility: To simplify the usage of MQT Bench, we provide a web interface (`https://www.cda.cit.tum.de/mqtbench`) and a Python package in addition to the open-source repository on GitHub (`https://github.com/cda-tum/mqt-bench`).
3. Algorithm selection: Broad selection of different algorithms ranging from building blocks, i.e., Quantum Fourier Transform (QFT), to applications, i.e., Grover's algorithm.
4. Extendability: Algorithms, gate-sets, and hardware architectures are easily extendable and integrable into MQT Bench.

By this, MQT Bench aims to improve the comparability, reproducibility, and transparency of empirical evaluations for the whole quantum software stack.

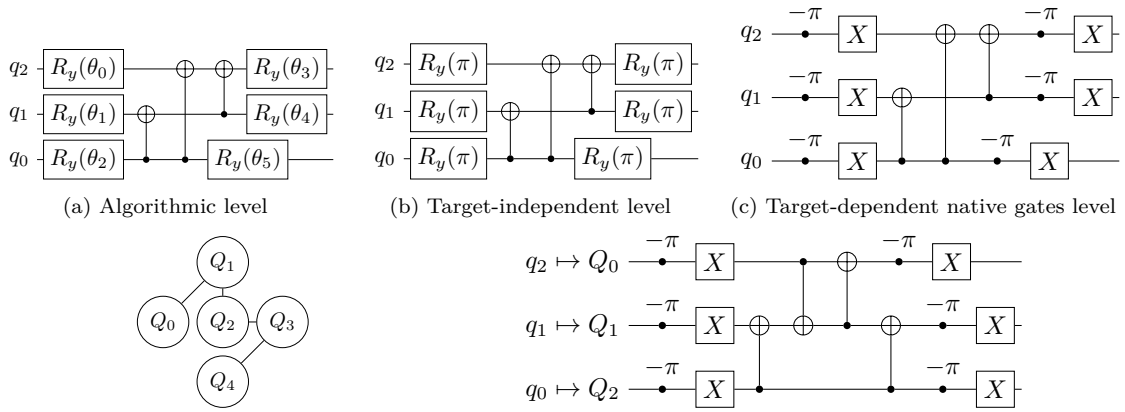
The rest of this work is structured as follows: In Section 2, we review the necessary basics of the quantum computing compilation flow and the respective software stack to keep this work self-contained. Afterwards, Section 3 reviews the state of the art on how this quantum software stack is currently evaluated and benchmarked—motivating the benchmark suite proposed in this paper. Based on that, Section 4 introduces the resulting benchmark suite and its core traits, before the generated benchmarks are evaluated in Section 5. Section 6 concludes this work.

2 Background

To keep this paper self-contained, this section gives a brief overview of the quantum circuit compilation flow with its different abstraction levels and the respective quantum software stack. The levels mentioned in this section are inspired by the structure proposed by the openQASM 3.0 specification [58] and can be found similarly in many other compilation flows as well.

2.1 Quantum Circuit Compilation Flow

Similar to the classical domain, executing a conceptual quantum algorithm on an actual device requires compiling it to a representation that adheres to all constraints imposed by the hardware. Usually, quantum algorithms are initially developed and tested on a hardware-agnostic level. Generally, this is described as a quantum circuit that consists of high-level building blocks without any restrictions to a certain gate-set or hardware architecture and is defined as the *algorithmic level*.



(a) Algorithmic level (b) Target-independent level (c) Target-dependent native gates level

(d) Target architecture: *ibmq_manila* (e) Target-dependent mapped level

Figure 1: Compilation of a Variational Quantum Algorithm.

Example 1. Variational Quantum Algorithms *(VQAs) are an emerging class of quantum algorithms with a wide range of applications. A respective circuit is depicted in Fig. 1a and shows an example of an ansatz function frequently used for* Variational Quantum Eigensolvers *(VQEs), a subclass of VQAs. On this abstraction level, the circuit is parameterized by the angles θ_i of the six single-qubit gates.*

The first step in realizing a conceptual quantum algorithm on the algorithmic level for a particular problem is to synthesize the high-level building blocks and optimize the resulting representation independently of the actual target architecture. In analogy to a classical compiler, this involves tasks such as constant propagation and folding, gate modifier evaluation, synthesis, loop unrolling, and gate simplification. The resulting representation is defined as the *target-independent level*.

Example 2. *VQAs are hybrid quantum-classical algorithms, where the parameters of the quantum ansatz are iteratively updated by a classical optimizer analogous to conventional gradient-based optimization. Consider again the circuit from Fig. 1a. Assuming that these parameters have been determined, e.g., $\theta_i = \pi$ for $i = 0, ..., 5$, they are now propagated and the resulting quantum circuit is shown in Fig. 1b.*

Today's quantum devices impose several constraints on the circuits that may be executed on them. Thus, a target-dependent compilation phase is necessary. For one, devices only provide a particular set of native gates—typically consisting of an entangling gate (like the CX gate) and some family of single-qubit gates. Therefore, the gates of a circuit need to be translated to this native gate-set, which is typically followed by an optimization pass that aims to reduce the introduced overhead. This representation level is defined as the *target-dependent native gates level*.

Example 3. *Different quantum computer realizations support different native gate-sets. In our example, we consider the* ibmq_manila *device as the target device that natively supports I, X, $\sqrt{X}$, R_z, and CX gates. Consequently, the R_y gates in Fig. 1b have to be converted using only these native gates. In this case, they are substituted by a sequence of R_z (denoted as $\bullet$ with a phase of $-\pi$) and X gates as shown in Fig. 1c.*

In addition to the limited gate-set, today's devices (at least those based on superconducting qubits) only feature limited connectivity between their qubits. Consequently, the circuit's logical qubits need to be *mapped* to the targeted device's physical qubits, so that multi-qubit gates are only applied to qubits directly connected on the device. Since such a mapping can rarely be determined in a static fashion, i.e., globally for the whole circuit, the mapping has to change dynamically throughout the circuit, which is frequently referred to as mapping or routing. Again, this is followed by a round of optimization in order to minimize the overhead caused by the compilation. This representation level is defined as the *target-dependent mapped level*.

Example 4. *Consider again the scenario from Example 3. The architecture of the* ibmq_manila *device is shown in Fig. 1d and defines between which qubits a two-qubit operation can be performed. Since the circuit shown in Fig. 1c contains CX gates operating between all combinations of qubits, there is no mapping directly matching the target architecture's layout. As a consequence, a non-trivial mapping followed by a round of optimization leads to the resulting circuit shown in Fig. 1e. This is also the reason for the different sequence of CX gates compared to the previous example.*

At this point, the circuit is ready to be sent to the quantum computer's backend for execution. From there, the individual gates are scheduled, linked to a particular calibration, and the resulting circuit is eventually passed to the target machine code generator. The resulting binaries are then forwarded to the execution engine to orchestrate the quantum computation on the actual device.

2.2 Quantum Software Stack

Quantum software is needed on all levels reviewed above in order to aid designers in eventually realizing useful quantum applications. In the following, we review three of the core design tasks for software tools.

Today, quantum computers are a scarce resource with limited availability and capability. Until more devices become available that are larger and less prone to errors, classical means to simulate quantum algorithm circuits (also known as *quantum circuit simulators*) are essential for fostering the development of quantum applications. Additionally, and in contrast to actual quantum computing devices, quantum circuit simulators allow for detailed insights into the quantum states throughout the circuit execution since the state's amplitudes are explicitly tracked during the simulation. On actual hardware, information can only be extracted in the form of measurements from the final quantum state, which each yields a classical bit string according to the distribution described by the state's amplitudes. Such simulators can be used on the highest possible level of abstraction, since they can be designed in a way that does not require restrictions on the circuit's gate-set or the qubits' connectivity, e.g., as proposed in [1]–[13]. Furthermore, they might also be used on the lowest possible level in order to perform noise-aware simulations to estimate how a circuit is likely to perform on an actual device, e.g., as proposed in [59]–[62]. Either way, on both abstraction levels, quantum circuit simulation is non-trivial since corresponding representations of states and operations, in general, require exponential space or runtime—requiring powerful software tools.

Due to the immense complexity of the tasks involved in quantum circuit compilation (as it has been illustrated in Section 2.1), e.g., mapping being NP-complete [63], manually conducting compilation often is not an option. Consequently, efficient software tools are needed across all levels of abstraction of the aforementioned flow in order to realize a conceptual algorithm on an actual device. Thus, various compilation tools have been proposed, e.g., [14]–[34].

During compilation, an algorithm's description is considerably altered and transformed. Naturally, it is of utmost importance to ensure that the originally intended functionality is preserved through all levels of abstraction. This procedure is called verification. While the underlying principle of comparing the transformations represented by different quantum circuits is conceptually simple, the exponential size of the underlying matrices makes this problem challenging—it has been shown to be QMA-complete [64]. Due to the increasing complexity of today's compilation flows, tools for verifying their results become increasingly important. Examples of verification tools have been proposed in [35]–[50].

These design tasks demonstrate the wide variety of software tools operating on and across different levels of abstraction—leading to comprehensive quantum circuit design flows realized through toolkits such as IBM's Qiskit [51], Google's Cirq [52], or Rigetti's Forest [53]. At the same time, all these tasks have in common that they have an immensely large complexity. Therefore, a multitude of techniques have been proposed for each task—each with its own trade-off between resource demand and quality of the result. It is key in the development of scientific methods to empirically evaluate and compare their performances on practical, relevant benchmarks.

3 Benchmarking

The benchmark suite proposed in this paper is not the first and certainly not the last collection of quantum circuit benchmarks. In this section, we review some of the existing suites and their respective foci. Afterwards, we discuss how MQT Bench further complements this.

3.1 Current State of the Art

Providing a comprehensive set of benchmarks to satisfy the needs across all levels of the quantum circuit compilation flow is a challenging task. Several approaches to provide benchmark suites for some of the levels within the quantum circuit compilation flow have already been proposed and a non-exhaustive overview is given in the following:

Application-Oriented Performance Benchmarks for Quantum Computing [54]: Starting with an example located on the algorithmic level, this benchmark suite rather focuses on high-level descriptions. At the moment of writing, it provides 13 benchmark algorithms which can be generated with a variable number of qubits via jupyter notebooks using multiple software compilation flows and are classified into four categories of complexity. The resulting circuits are available as high-level Python objects without any further compilation steps.

SupermarQ [55]: Similarly, SupermarQ also focuses on the algorithmic level by providing benchmarks with an adjustable qubit range for eight algorithms as high-level Python objects via a Python package. Additionally, six feature vectors are proposed to describe the characteristics of the benchmarks.

QASMbench [56]: This benchmark suite focuses on the target-independent level. It offers numerous quantum circuits with a wide but fixed range of both the number of qubits and depth. It classifies the benchmarks into three categories of sizes and three categories of algorithm classes. All circuits are available in an intermediate representation according to the openQASM 2.0 specification [65].

RevLib [57]: This benchmark suite provides a wide variety of reversible circuits in an intermediate representation format specified by the authors. Classical functions are frequently embedded into reversible circuits in order to use them in quantum algorithms, e.g., oracle functions or Boolean building blocks such as the modular exponentiation in Shor's algorithm [66]. In addition, decomposed versions of this benchmark suite have found widespread use in evaluating the performance of mapping tools, e.g., in [32]–[34]. Since reversible circuits merely form a subclass of quantum circuits and do not employ quantum mechanical effects such as superposition and entanglement, they do not serve as adequate benchmarks for the whole stack.

3.2 Motivation

All of the mentioned benchmark suites have in common that they each target a specific abstraction level within the quantum circuit compilation flow. Although the respectively intended target level is well covered, the demand for a cross-level benchmark suite is not fully met yet. Additionally, so far, there is no mutual consolidation which benchmarks to use for empirical evaluations of software tools. This results in a lower comparability, reproduceability, and transparency of results and may even cause confusion because benchmarks believed to realize the same particular task are quite frequently realized in a completely different fashion, e.g., a benchmark called "grover" could realize Grover's algorithm with any particular oracle. Consequently, the demand for a benchmark suite that spans the whole stack of abstraction levels constitutes the main motivation of this contribution. In order to realize such a benchmark suite, several challenges have to be tackled:

- Distinct requirements for benchmark circuits on different levels: The closer a level is to the actual computing hardware, the more requirements have to be fulfilled, e.g., only gates from the device's native gate-set are allowed and the connectivity of the hardware architecture must be considered.
- Accessibility: In order to facilitate its adoption, the benchmark suite needs to be as easy to use as possible. While most benchmark suites already provide open-source access, running and adapting the code to the user's needs, e.g., obtaining a selection of relevant benchmarks, frequently is tedious or impossible and a solution that keeps this hurdle low is desirable.

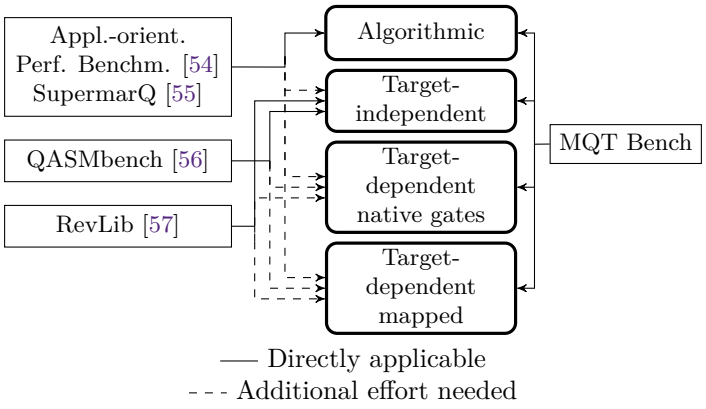


Figure 2: Benchmarking software tools with and without MQT Bench.

- Generalizability of a software tool: This can only be guaranteed if it is evaluated on a broad set of benchmarks, relating to several characteristics, e.g., the number of qubits, circuit depth, or the application domain.
- Extendability: In addition to that, quantum computing is rapidly advancing and the extendability of the benchmark suite to future algorithms, gate-sets, and architectures becomes even more important.

MQT Bench has been developed with these challenges in mind and aims to be a first step towards covering the whole quantum software stack with a benchmark suite based on the following four core traits (which are described in detail in Section 4):

1. Cross-level benchmarking: All benchmarks are provided on four abstraction levels reviewed in Section 2.1.
2. Accessibility: A website is provided (`https://www.cda.cit.tum.de/mqtbench`) to simplify the usage of MQT Bench as much as possible. Additionally, all benchmarks can also be generated on-demand using our Python package. While most users' needs should be covered by this, we also give access to the open-source repository on GitHub (`https://github.com/cda-tum/mqt-bench`).
3. Algorithm Selection: Broad selection of different benchmarks with parameterizable characteristics ranging from building blocks, i.e., QFT, to applications, i.e., Grover's algorithm.
4. Extendability: MQT Bench is easily extendable with respect to available benchmarks, native gate-sets, and hardware architectures.

Benchmarking quantum software tools and compilation flows with the same benchmarks aims to aid comparability, reproducibility, and transparency of empirical evaluations. How this leads to more consistent testing is shown in the following example.

Example 5. *Consider a full quantum compilation flow across all four levels which shall be developed and tested as depicted in Fig. 2. On the left-hand side of the levels, the current state of the art of benchmarking is sketched. To comprehensively test each level, different benchmark suites have to be adapted and used. In contrast, MQT Bench replaces all of those and enables testing software tools with the same benchmarks throughout the whole quantum circuit compilation flow as shown on the right-hand side.*

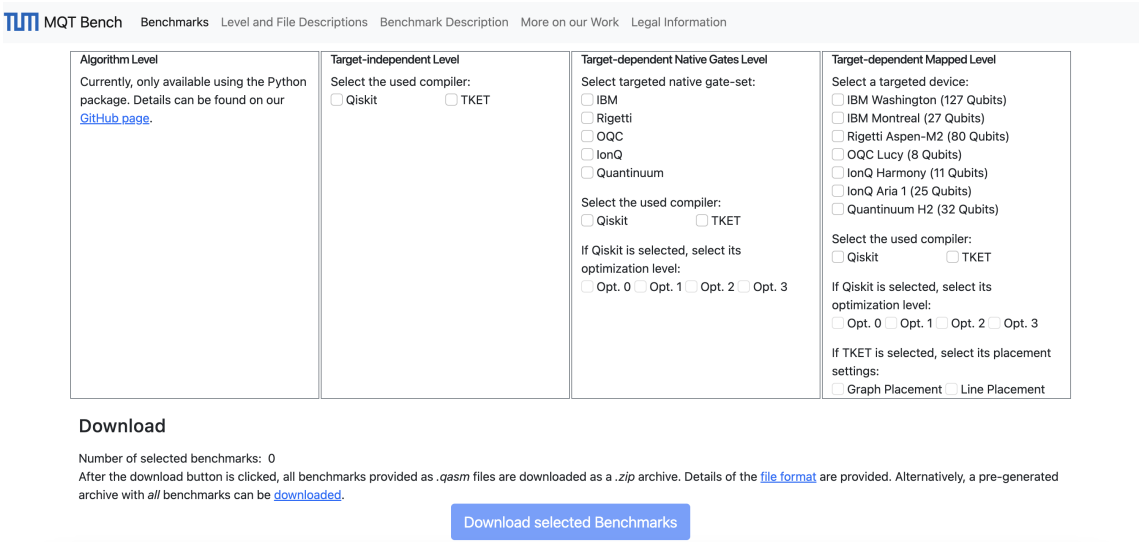


Figure 3: User interface of the MQT Bench website.

4 MQT Bench

In this section, we describe in more detail how each of the four core traits mentioned above is implemented.

4.1 Cross-Level Benchmarking

As exemplarily illustrated in Section 2, software development in quantum computing takes place on various levels. While there arguably are numerous possible levels, MQT Bench focuses on four as a balanced measure between specificity and generizability—inspired by the structure proposed in the openQASM 3.0 specification [58]. Providing these benchmarks on all of the four abstraction levels is the main contribution of this work. More precisely, the following levels are covered:

1. *Algorithmic level*: In this format, all kinds of quantum gates may be used and subsumed into high-level building blocks. Loops are possible and the circuit structure might be described by constants, variables, and mathematical expressions. This level provides benchmarks on the most generic level and specific values are assigned to its variables, e.g., the number of iterations of a loop. The number of qubits can be specified.

2. *Target-independent level*: Here, loop unrolling, constant folding and propagation are conducted. Also, naive gate simplification is enforced. Again, the number of qubits can be adapted through the website and the used compiler can be selected.

3. *Target-dependent native gates level*: On this level, a particular native gate-set is chosen and the circuit is transpiled and optimized accordingly. Here, the native gate-set, the compiler used and its settings can be specified.

4. *Target-dependent mapped level*: Finally, a dedicated architecture layout is chosen and the circuit is mapped to the targeted device such that it satisfies all its connectivity constraints and becomes executable on the device. Similar to the previous level, the used compiler and its settings are selectable—in addition to the target device.

All relevant information of a generated benchmark is denoted in short in the file name and in detail within the file itself.

4.2 Accessibility

MQT Bench strives to meet the needs of many. To accomplish this goal, it has to be as user-friendly as possible. On the one hand, this involves the benchmark library itself. All of the high-level, algorithmic benchmark scripts and routines to generate representations for the individual levels are publicly available on GitHub (https://github.com/cda-tum/mqt-bench). This allows for complete transparency on how the respective circuits have been generated. Additionally, a Python package is provided such that each benchmark can be generated on-demand.

On the other hand, and most importantly, this involves the way in which users interface with the benchmark suite. While most benchmark suites are provided as an accumulation of benchmark files—making it hard to extract the desired set of benchmarks, MQT Bench includes an easy-to-use, no-coding-required web interface (https://www.cda.cit.tum.de/mqtbench) that provides the means to filter the large number of pre-generated benchmarks according to the specific needs of the user. This web interface and its underlying server software is also part of our Python package, such that every user can utilize the same interface locally without the need to access our publicly available server. A screenshot of the website's user interface and its configuration options is shown in Fig. 3.

4.3 Algorithm Selection

To provide a broad spectrum of different benchmarks, MQT Bench comprises most of today's de-facto standard quantum algorithms. This includes building blocks, e.g., Quantum Fourier Transform (QFT) and the Greenberger-Horne-Zeilinger state preparation (GHZ), up to higher-level algorithms, e.g., Grover's [67] and Shor's [66] algorithm. At the time of writing, MQT Bench comprises the following benchmarks:

- Amplitude Estimation (AE)
- Deutsch-Jozsa (DJ) algorithm
- Graph state preparation
- GHZ state preparation
- Grover's algorithm
- Quantum Approximation Optimization Algorithm (QAOA)
- Quantum Fourier Transformation (QFT)
- Quantum Phase Estimation (QPE)
- Quantum Walk
- Random circuit
- Shor's algorithm
- Variational Quantum Eigensolver (VQE)
- Three VQA ansatz functions with randomly initialized parameters: Two Local, Real Amplitudes and Efficient SU2
- W-State preparation

Additionally, MQT Bench provides several application benchmarks specifically targeting variational quantum algorithms, since those algorithms are especially promising in the NISQ-era [68]. To this end, we use the classification provided by IBM Qiskit's application levels: Optimization, Machine Learning, Finance, and Nature:

- *Optimization*: Travelling salesman problem and vehicle routing
- *Machine Learning*: Quantum Neural Network (QNN)
- *Finance*: Portfolio optimization and option pricing
- *Nature*: Ground state estimation

4.4 Extendability

Quantum computing is a rapidly evolving area of research and, especially in recent years, numerous promising algorithms and applications have emerged. Any benchmark suite has to be extendable and must continuously evolve in order to remain relevant. MQT Bench is designed to be easily extendable in a multitude of ways:

- *Algorithms*: New applications and algorithms can be easily integrated into MQT Bench by providing the corresponding algorithmic level description. The generation of all other levels and options (such as the native gate-sets, architectures, and compilers with their settings) is automatically handled by the proposed library.
- *Native gate-sets*: So far, native gate-sets of superconducting quantum computers from IBM, Rigetti, and Oxford Quantum Circuits are provided—in addition to the gate-sets of IonQ's and Quantinuum's ion trap-based quantum computers. In the future, additional gate-sets (such as those of Google's and or AQT's quantum computers) can be realized by adopting the necessary compilation routines in the proposed library.
- *Hardware architectures*: All major players currently rely on a modular platform for the architecture of their devices in order to further scale their quantum computers. It is straightforward to extend MQT Bench's list of available architectures to accommodate future architectures.

Although the four levels MQT Bench is based on cover today's main use cases, the number of abstraction levels most likely is going to increase in the future. While the mapped level has been considered the last one before a circuit is sent to the quantum computer for execution, pulse-level programming is envisioned to give even more control to software developers as proposed in [69]. On the other end of the spectrum, higher-level abstractions and programming languages will be required to foster the adoption of quantum technology. Currently, a programmatic description of the benchmarks is needed on the algorithmic level. An even higher level where this programmatic description will be automatically derived from is also envisioned. *QUARK* [70], a framework for quantum computing application benchmarking consisting of four levels where the lowest level considers the whole compilation flow discussed in Section 2.1, constitutes one step towards this direction.

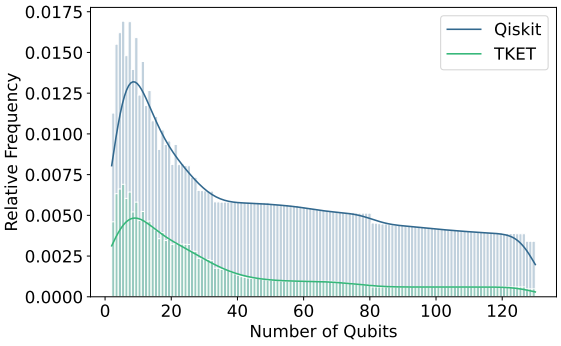
5 Evaluation

At the time of writing, MQT Bench (*v1.0.0*) considers two compilers, five native gate-sets, and seven devices ranging from 8 to 127 qubits. In this section, we provide an overview of several characteristics and statistics of the resulting pre-generated benchmarks. To this end, the following metrics are illustrated in Fig. 4:

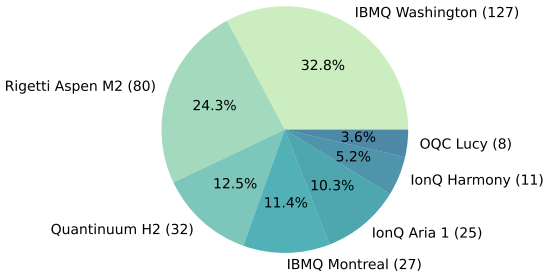
Number of qubits: A first broad overview is given in Fig. 4a, which shows the relative frequency of the generated benchmarks per number of qubits and per compiler. The relative frequency of benchmarks is decreasing with an increasing number of qubits due to fewer devices being available for higher qubit numbers and the maximal generation time for a benchmark file being exceeded more frequently. Furthermore, the benchmark generation using the *TKET* compiler generally took more time than *Qiskit*—leading to a smaller number of generated benchmarks, especially on the target-dependent mapped level.

Distribution of target-dependent mapped level benchmarks: The distribution of the target device for the mapped level is shown in Fig. 4b, with the number of qubits denoted in brackets. As expected, the larger the device, the more benchmarks are created for it. On the target-dependent native gates level, there is an equal distribution with respect to the number of benchmarks created for each of the five native gate-sets.

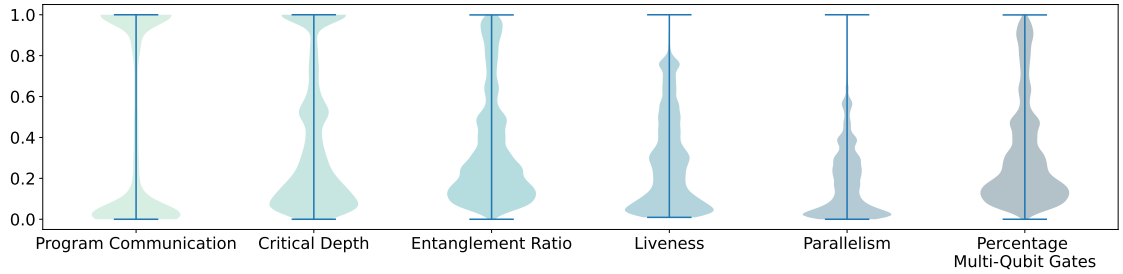
Distribution of benchmark characteristics: In Fig. 4c, six different characteristics with values between 0 and 1 are evaluated for all pre-generated benchmarks. Five of those characteristics (Program Communication, Critical Depth, Entanglement Ratio, Liveness, and Parallelism) have been proposed in [55]. Furthermore, the percentage of multi-qubit gates is evaluated.



(a) Relative distribution of benchmarks per number of qubits and used compiler.



(b) Device distribution of target-dependent mapped level benchmarks.



(c) Distributions of quantum circuit characteristics (with all but the last one taken from [55]).

Figure 4: Insights into the provided benchmarks and their characteristics.

6 Conclusion

In this work, we proposed *MQT Bench*—a quantum circuit benchmark suite (as part of the *Munich Quantum Toolkit*, MQT) comprising different algorithms, compilers, native gate-sets, and target devices—resulting in more than 70,000 benchmark circuits ranging from 2 to 130 qubits on four abstraction levels. To keep this large number of benchmarks manageable, we provide an easy-to-use web interface (https://www.cda.cit.tum.de/mqtbench) allowing users to filter the benchmarks according to their needs. *MQT Bench* is also provided as a Python package (including the server software to start the web interface locally), such that each of the benchmarks can be easily generated on-demand. Furthermore, we give access to the open-source repository on GitHub (https://github.com/cda-tum/mqt-bench). By this, *MQT Bench* presents a first step towards serving a single benchmark suite for the whole quantum software stack—facilitating empirical evaluations of quantum software tools that are comparable, reproducible, and transparent.

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

References

[1] A. Zulehner and R. Wille, "Advanced simulation of quantum computations," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019. DOI: 10.1109/TCAD.2018.2834427.

[2] D. M. Miller, M. A. Thornton, and D. Goodman, "A decision diagram package for reversible and quantum circuit simulation," in *Int'l Conf. on Evolutionary Computation*, 2006. DOI: 10.1109/CEC.2006.1688610.

[3] S. Hillmich, A. Zulehner, R. Kueng, I. L. Markov, and R. Wille, "Approximating decision diagrams for quantum circuit simulation," *ACM Transactions on Quantum Computing*, 2022. DOI: 10.1145/3530776.

[4] S. Hillmich, A. Zulehner, and R. Wille, "Concurrency in DD-based quantum circuit simulation," in *Asia and South Pacific Design Automation Conf.*, 2020. DOI: 10.1109/ASP-DAC47756.2020.9045711.

[5] L. Burgholzer, H. Bauer, and R. Wille, "Hybrid Schrödinger-Feynman simulation of quantum circuits with decision diagrams," in *Int'l Conf. on Quantum Computing and Engineering*, 2021. DOI: 10.1109/QCE52317.2021.00037.

[6] L. Burgholzer, A. Ploier, and R. Wille, "Simulation paths for quantum circuit simulation with decision diagrams: What to learn from tensor networks, and what not," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2022. DOI: 10.1109/TCAD.2022.3197969.

[7] A. Kissinger, J. van de Wetering, and R. Vilmart, "Classical simulation of quantum circuits with partial and graphical stabiliser decompositions," 2022. DOI: 10.48550/arXiv.2202.09202. arXiv: 2202.09202.

[8] J. Brennan *et al.*, "Tensor network circuit simulation at exascale," 2021. DOI: 10.48550/arXiv.2110.09894. arXiv: 2110.09894.

[9] T. Vincent *et al.*, "Jet: Fast quantum circuit simulations with parallel task-based tensor-network contraction," 2021. DOI: 10.48550/arXiv.2107.09793. arXiv: 2107.09793.

[10] J. Im and S. Kang, "Graph Partitioning Approach for Fast Quantum Circuit Simulation," in *Asia and South Pacific Design Automation Conf.*, 2023. DOI: 10.1145/3566097.3567928.

[11] D. Lykov, R. Schutski, A. Galda, V. Vinokur, and Y. Alexeev, "Tensor Network Quantum Simulator With Step-Dependent Parallelization," 2020. DOI: 10.48550/arXiv.2012.02430. arXiv: 2012.02430.

[12] H. De Raedt *et al.*, "Massively parallel quantum computer simulator, eleven years later," *Computer Physics Communications*, 2019. DOI: 10.1016/j.cpc.2018.11.005.

[13] S. Bravyi and D. Gosset, "Improved classical simulation of quantum circuits dominated by Clifford gates," *Physical Review Letters*, 2016. DOI: 10.1103/PhysRevLett.116.250501.

[14] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *Quantum Science and Technology*, 2018. DOI: 10.1088/2058-9565/aaa5cc.

[15] M. Amy and V. Gheorghiu, "Staq—a full-stack quantum processing toolkit," *Quantum Science and Technology*, 2020. DOI: 10.1088/2058-9565/ab9359.

[16] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," *ACM SIGPLAN Not.*, 2013. DOI: 10.1145/2499370.2462177.

[17] N. Quetschlich, L. Burgholzer, and R. Wille, "Predicting Good Quantum Circuit Compilation Options," in *Int'l Conf. on Quantum Software*, 2023. DOI: 10.48550/arXiv.2210.08027.

[18] N. Quetschlich, L. Burgholzer, and R. Wille, "Compiler Optimization for Quantum Computing Using Reinforcement Learning," in *Design Automation Conf.*, 2023. DOI: 10.48550/arXiv.2212.04508.

[19] T. Peham, N. Brandl, R. Kueng, R. Wille, and L. Burgholzer, "Depth-optimal synthesis of Clifford circuits with SAT solvers," 2023. DOI: 10.48550/arXiv.2305.01674. arXiv: 2305.01674.

[20] L. Burgholzer, S. Schneider, and R. Wille, "Limiting the search space in optimal quantum circuit mapping," in *Asia and South Pacific Design Automation Conf.*, 2022. DOI: 10.1109/ASP-DAC52403.2022.9712555.

[21] 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. DOI: 10.1145/3316781.3317859.

[22] S. Hillmich, A. Zulehner, and R. Wille, "Exploiting Quantum Teleportation in Quantum Circuit Mapping," in *Asia and South Pacific Design Automation Conf.*, 2021. DOI: 10.1145/3394885.3431604.

[23] 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. DOI: 10.1109/TCAD.2018.2846658.

[24] A. Zulehner and R. Wille, "Compiling SU(4) quantum circuits to IBM QX architectures," in *Asia and South Pacific Design Automation Conf.*, 2019. DOI: 10.1145/3287624.3287704.

[25] I. Shaik and J. van de Pol, "Optimal layout synthesis for quantum circuits as classical planning," 2023. DOI: 10.48550/arXiv.2304.12014. arXiv: 2304.12014.

[26] J. Liu, E. Younis, M. Weiden, P. Hovland, J. Kubiatowicz, and C. Iancu, "Tackling the Qubit Mapping Problem with Permutation-Aware Synthesis," 2023. DOI: 10.48550/arXiv.2305.02939. arXiv: 2305.02939.

[27] R. Wille and L. Burgholzer, "MQT QMAP: Efficient quantum circuit mapping," in *Int'l Symp. on Physical Design*, 2023. DOI: 10.1145/3569052.3578928.

[28] C. Zhang, A. B. Hayes, L. Qiu, Y. Jin, Y. Chen, and E. Z. Zhang, "Time-optimal qubit mapping," in *Int'l Conf. On Architectural Support for Programming Languages and Operating Systems*, 2021. DOI: 10.1145/3445814.3446706.

[29] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," in *Int'l Conf. On Architectural Support for Programming Languages and Operating Systems*, 2019. DOI: 10.1145/3297858.3304075.

[30] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the qubit routing problem," in *Theory of quantum computation, communication and cryptography*, 2019. DOI: 10.4230/LIPIcs.TQC.2019.5.

[31] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "T|ket⟩: A retargetable compiler for NISQ devices," *Quantum Science and Technology*, 2021. DOI: 10.1088/2058-9565/ab8e92.

[32] B. Tan and J. Cong, "Optimal layout synthesis for quantum computing," in *Int'l Conf. on CAD*, 2020. DOI: 10.1145/3400302.3415620.

[33] 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. DOI: 10.1145/3297858.3304023.

[34] K. N. Smith and M. A. Thornton, "A quantum computational compiler and design tool for technology-specific targets," in *Int'l Symp. on Computer Architecture*, 2019. DOI: 10.1145/3307650.3322262.

[35] S. Yamashita and I. L. Markov, "Fast equivalence-checking for quantum circuits," in *Int'l Symp. on Nanoscale Architectures*, 2010. DOI: 10.1109/NANOARCH.2010.5510932.

[36] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton, and R. Drechsler, "QMDDs: Efficient quantum function representation and manipulation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2016. DOI: 10.1109/TCAD.2015.2459034.

[37] R. Duncan, A. Kissinger, S. Perdrix, and J. van de Wetering, "Graph-theoretic simplification of quantum circuits with the ZX-calculus," *Quantum*, 2020. DOI: 10.22331/q-2020-06-04-279.

[38] L. Burgholzer, R. Raymond, and R. Wille, "Verifying results of the IBM Qiskit quantum circuit compilation flow," in *Int'l Conf. on Quantum Computing and Engineering*, 2020. DOI: 10.1109/QCE49297.2020.00051.

[39] T. Peham, L. Burgholzer, and R. Wille, "Equivalence checking of quantum circuits with the ZX-Calculus," *Journal of Emerging and Selected Topics in Circuits and Systems*, 2022. DOI: 10.1109/JETCAS.2022.3202204.

[40] T. Peham, L. Burgholzer, and R. Wille, "Equivalence checking of parameterized quantum circuits: Verifying the compilation of variational quantum algorithms," in *Asia and South Pacific Design Automation Conf.*, 2023. DOI: 10.1145/3566097.3567932.

[41] L. Burgholzer and R. Wille, "Advanced equivalence checking for quantum circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2021. DOI: 10.1109/TCAD.2020.3032630.

[42] L. Burgholzer, R. Kueng, and R. Wille, "Random stimuli generation for the verification of quantum circuits," in *Asia and South Pacific Design Automation Conf.*, 2021. DOI: 10.1145/3394885.3431590.

[43] L. Burgholzer and R. Wille, "Handling non-unitaries in quantum circuit equivalence checking," in *Design Automation Conf.*, 2022. DOI: 10.1145/3489517.3530482.

[44] W. Chun-Yu, T. Yuan-Hung, J. Chaio-Shan, and J. Jie-Hong, "Accurate BDD-based Unitary Manipulation for Scalable and Robust Quantum Circuit Verification," in *Design Automation Conf.*, 2022. DOI: 10.1145/3489517.3530481.

[45] R. Tao *et al.*, "Giallar: Push-button verification for the Qiskit quantum compiler," in *Int'l Conf. on Programming Language Design and Implementation*, 2022. DOI: 10.1145/3519939.3523431.

[46] R. Wille and L. Burgholzer, "Verification of Quantum Circuits," in *Handbook of Computer Architecture*, 2022. DOI: 10.1007/978-981-15-6401-7_43-1.

[47] S.-A. Wang, C.-Y. Lu, I.-M. Tsai, and S.-Y. Kuo, "An XQDD-based verification method for quantum circuits," in *IEICE Trans. Fundamentals*, 2008. DOI: 10.1093/ietfec/e91-a.2.584.

[48] X. Hong, M. Ying, Y. Feng, X. Zhou, and S. Li, "Approximate Equivalence Checking of Noisy Quantum Circuits," in *Design Automation Conf.*, 2021. DOI: 10.1109/DAC18074.2021.9586214.

[49] Y.-F. Chen, K.-M. Chung, O. Lengál, J.-A. Lin, W.-L. Tsai, and D.-D. Yen, "An automata-based framework for verification and bug hunting in quantum circuits," *Programming Languages*, 2023. DOI: 10.1145/3591270.

[50] H.-L. Liu, Y.-T. Li, Y.-C. Chen, and C.-Y. Wang, "A Robust Approach to Detecting Non-Equivalent Quantum Circuits Using Specially Designed Stimuli," in *Asia and South Pacific Design Automation Conf.*, 2023. DOI: 10.1145/3566097.3567935.

[51] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023. DOI: 10.5281/zenodo.2573505.

[52] Cirq Developers, "Cirq," version v0.12.0, See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors, 2021. DOI: 10.5281/zenodo.5182845.

[53] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016. DOI: 10.48550/arXiv.1608.03355. arXiv: 1608.03355.

[54] T. Lubinski *et al.*, "Application-oriented performance benchmarks for quantum computing," *IEEE Transactions on Quantum Engineering*, 2023. DOI: 10.1109/TQE.2023.3253761.

[55] T. Tomesh *et al.*, "Supermarq: A scalable quantum benchmark suite," in *IEEE Int'l Symp. on High-Performance Computer Architecture*, 2022. DOI: 10.1109/HPCA53966.2022.00050.

[56] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang, "QASMBench: A low-level quantum benchmark suite for NISQ evaluation and simulation," *ACM Transactions on Quantum Computing*, 2022. DOI: 10.1145/3550488.

[57] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, RevLib is available at http://www.revlib.org, 2008. DOI: 10.1109/ISMVL.2008.43.

[58] A. Cross *et al.*, "Openqasm 3: A broader and deeper quantum assembly language," *ACM Transactions on Quantum Computing*, 2022. DOI: 10.1145/3505636.

[59] T. Grurl, R. Kueng, J. Fuß, and R. Wille, "Stochastic quantum circuit simulation using decision diagrams," in *Design, Automation and Test in Europe*, 2021. DOI: 10.23919/DATE51398.2021.9474135.

[60] B. Villalonga *et al.*, "A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware," *npj Quantum Information*, 2019. DOI: 10.1038/s41534-019-0196-1.

[61] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "QuEST and High Performance Simulation of Quantum Computers," *Scientific Reports*, 2019. DOI: 10.1038/s41598-019-47174-9.

[62] T. Grurl, J. Fuß, and R. Wille, "Noise-aware quantum circuit simulation with decision diagrams," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2023. DOI: 10.1109/TCAD.2022.3182628.

[63] 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. DOI: 10.1145/3168822.

[64] D. Janzing, P. Wocjan, and T. Beth, ""Non-identity check" is QMA-complete," *Int. J. Quantum Inform.*, 2005. DOI: 10.1142/S0219749905001067.

[65] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open Quantum Assembly Language," 2017. DOI: 10.48550/arXiv.1707.03429. arXiv: 1707.03429.

[66] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, 1999. DOI: 10.1137/S0097539795293172.

[67] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Symp. on Theory of Computing*, 1996. DOI: 10.1145/237814.237866.

[68] M. Cerezo *et al.*, "Variational Quantum Algorithms," *Nature Reviews Physics*, 2021. DOI: 10.1038/s42254-021-00348-9.

[69] B. Li *et al.*, "Pulse-level noisy quantum circuits with QuTiP," *Quantum*, 2022. DOI: 10.22331/q-2022-01-24-630.

[70] J. R. Finžgar, P. Ross, L. Hölscher, J. Klepsch, and A. Luckow, "Quark: A framework for quantum computing application benchmarking," in *Int'l Conf. on Quantum Computing and Engineering*, 2022. DOI: 10.1109/QCE53715.2022.00042.