Scaling Quantum Computations via Gate Virtualization

Nathaniel Tornow
nathaniel.tornow@tum.de
TU Munich
and Leibniz Supercomputing Centre
Germany

Emmanouil Giortamis
emmanouil.giortamis@tum.de
TU Munich
Germany

Pramod Bhatotia
pramod.bhatotia@tum.de
TU Munich
Germany

Abstract

We present the Quantum Virtual Machine (QVM), an end-to-end generic system for scalable execution of large quantum circuits with high fidelity on noisy and small quantum processors (QPUs) by leveraging gate virtualization. QVM exposes a virtual circuit intermediate representation (IR) that extends the notion of quantum circuits to incorporate gate virtualization. Based on the virtual circuit as our IR, we propose the QVM compiler—an extensible compiler infrastructure to transpile a virtual circuit through a series of modular optimization passes to produce a set of optimized circuit fragments. Lastly, these transpiled circuit fragments are executed on QPUs using our QVM runtime—a scalable and distributed infrastructure to virtualize and execute circuit fragments on a set of distributed QPUs.

We evaluate QVM on IBM's 7- and 27-qubit QPUs. Our evaluation shows that using our system, we can scale the circuit sizes executable on QPUs up to double the size of the QPU while improving fidelity by 4.7× on average compared to larger QPUs and that we can effectively reduce circuit depths to only 40% of the original circuit depths.

1 Introduction

Quantum computers promise to solve otherwise intractable problems in optimization [24], factorization [72], or quantum simulation [34, 61]. However, the reliable operation of quantum processing units (QPUs) is extremely challenging, as the same properties that could lead to computational benefits are also the main reason for uncontrollable noise and state-decoherence during a quantum computation on a QPU[63]. This still severely limits the number of qubits and operations we can run within the same quantum program.

Gate virtualization (GV) has recently been proposed to scale the size of quantum programs running with high fidelity on small and noisy QPUs [44]. This technique virtualizes binary qubit gates by sampling the gate with single qubit operations instead. Theoretical work shows that GV allows quantum circuits to be optimized to scale and improve fidelity in two different dimensions: First, quantum circuits can be decomposed into multiple smaller circuit fragments to run on small QPUs [44, 60, 62], and second, circuit depth can be reduced to increase overall fidelity [10, 89].

However, the effectiveness of gate virtualization is severely hampered by the lack of general and extensible procedures for automatically applying and executing gate virtualization. Previous studies have primarily concentrated on utilizing gate virtualization through ad-hoc methods or on an individual application level [9, 10, 89]. Moreover, the applicability of gate virtualization suffers greatly from the high computational cost, since virtualizing k two-qubit gates comes with a quantum circuit and classical post-processing overhead of $O(6^k)$ [44].

To this end, we target the following research question: *How can we design a generic and extensible system that fully utilizes the full potential of GV to scale the size of circuits that can be executed with high fidelity on current QPUs, despite the computational overhead?*

To address this research question, in this work, we present the Quantum Virtual Machine (QVM), a system for scalable and reliable execution of quantum circuits on small and noisy QPUs by fully leveraging GV. We make the following key contributions:

- To enable the general and programmable application of GV, and as the basis of our work, we present the **virtual circuit IR** (VC-IR). The VC-IR extends the quantum circuit abstraction, manages virtual gates, decompositions into several smaller circuit fragments, and data structures to efficiently analyze and apply GV (§ 4.2).
- To enable automatic and efficient GV that optimizes a circuit with as little post-processing as possible, we introduce the **QVM Compiler**, an extensible pipeline for converting large quantum circuits into optimized virtual circuits (§ 4). The compiler enables multiple optimization passes that apply GVs based on the VC-IR. We present three generic optimization passes for efficient gate virtualization on arbitrary quantum circuits. (1) The *circuit cutter* (§ 4.3) decomposes a circuit into several smaller fragments to run on smaller QPUs, (2) the *dependency reducer* (§ 4.4) reduces the dependencies within a circuit to reduce error propagation and the number of SWAP gates, and (3) the *qubit reuser* (§ 4.5) applies a qubit reuse technique to reduce the width of circuit fragments, enabling a trade-off between GV's overhead and circuit depth.
- To enable as many GVs as possible to benefit from its opportunities, we present the **QVM Runtime**, a scalable system that can run a virtual circuit (VC) on a set of QPUs and classical nodes (§ 5). The runtime uses the core component of a *virtualizer* (§ 5.2) that instantiates fragments of the VC and computes the result of the VC

using highly parallel post-processing. The quantum circuits are executed in a scalable manner on a distributed set of QPUs using QVM's *QPU manager* (§ 5.3).

We implement QVM in Python by building on the Qiskit framework [64]. For our compiler, in addition to heuristic algorithms, we implement optimal passes using Answer Set Programming following our optimization models [26].

We evaluate QVM on IBM's 7- and 27-qubit QPUs and simulators using various circuits used in popular quantum algorithms [37, 65, 82]. Based on our analysis, we can execute circuits with up to 2× the number of qubits of the QPU used while improving fidelity by an average of 4.7× and up to 33.6× (§ 6.1). Our intra-circuit dependency reduction techniques reduce the depth of transpiled circuits on average to 64% of the original circuit and increase fidelity by an average of 1.4× and up to 5.2× (§ 6.2). Our dependency reducer also enables the reuse of more qubits to reduce the width of circuits with less virtualization overhead (§ 6.3). Our QVM runtime scales and can be used to run large virtual circuits with low memory requirements (§ 6.4-6.5).

2 Background and Motivation

2.1 Computational Model

We define a quantum computation by adopting the computational model of [60]. A quantum computation is defined by a quantum circuit with m qubits and one- and two-qubit gates, initialized with the $|0\rangle^{\otimes m}$ state (Fig. 1, (a)). All output qubits are measured in the computational basis to obtain a measurement bitstring $x \in \{0,1\}^m$. We then apply a classical post-processing function $f : x \to [-1, 1]$ to the bitstrings. We aim to accurately approximate the expectation value $\mathbb{E}_x\, f(x)$ of the function across the bitstrings.

In our work, we focus on measuring expectation values of m-qubit projection operators $O = P_y$, i.e., we measure the expectation value $\langle P_y \rangle$ of the observable P_y. In our case, f is therefore defined as $f(x) = 1$ if $x = y$, otherwise $f(x) = 0$. This is the same as calculating the probability of obtaining a certain output string y, i.e., $\langle P_y \rangle = p(y)$. $\langle P_y \rangle$ can, therefore, be determined by sampling the quantum circuit many times and calculating the ratio of the shots that returned y to the total number of shots.

2.2 Foundations of Gate Virtualization

Gate virtualization (GV) allows us to decompose two-qubit gates into a combination of single-qubit gates [44]. We show GV schematically in Fig. 1 (b). Instead of executing the original circuit's binary gate, we can calculate a weighted sum of six circuit instances to estimate $\langle O \rangle$. In each circuit instance i, instead of the original two-qubit gate, we insert A_i and B_i, which are either one-qubit unitary gates or projective measurements. This allows us to decompose each instance i into two smaller, completely independent sub-circuits, which can be sampled independently. We reconstruct the result with

$$\langle O \rangle = \sum_{i=1}^{6} c_i \langle O \rangle_i = \sum_{i=1}^{6} c_i \langle O_1 \rangle_i \langle O_2 \rangle_i, \tag{1}$$

where $\langle O \rangle_i$ is the result of each instance i and $O = O_1 \otimes O_2$. The exact proofs and formulas of GV for standard gates such as CNOT or R_{ZZ} can be found in [44].

Virtualizing Mulitple Gates. We now generalize GV to be applied on k gates in a circuit. We can think of adding a GV of another gate in a circuit as performing an additional GV for each instance of the original virtual gate.

To formalize the QPD of multiple gates in a circuit, let G_v be the set of all virtual gates in a quantum circuit. We then define a coefficient vector for each virtual gate $g \in G_v$ as $\mathbf{c}_g = (c_1, ..., c_6)$. We define the global coefficient vector as

$$\mathbf{C} = \bigotimes_{g \in G_v} \mathbf{c}_g, \tag{2}$$

i.e., the tensor product of all individual coefficient vectors c_g. Therefore, $\mathbf{C}$ is a vector with $|\mathbf{C}| = 6^k$ entries. To reconstruct the final results, we calculate

$$\langle O \rangle = \sum_{c_i \in \mathbf{C}} c_i \prod_{j=1}^{f} \langle O_j \rangle_i, \tag{3}$$

which is the generalization of Eq. (1). Here f is the number of subcircuits, and $\langle O_j \rangle_i$ is the result of the jth subcircuit in the ith global instance. We must, therefore, calculate a sum over $|\mathbf{C}| = 6^k$ elements. Since in general $|\mathbf{C}| \gg f > |G_v|$, we need $O(6^k)$ operations to calculate Eq. (3).

GV thus causes an exponential post-processing overhead of $O(6^k)$ [44]. This severely limits the number of gates that can be virtualized within a circuit, meaning that we need to find a good compromise between the additional runtime and the benefits of GV, as described in the next section.

2.3 Opportunities of Gate Virtualization

State-of-the-art circuit transpilers [5] mainly focus on minimizing the circuit's post-transpilation depth and number of CNOTs. GV can be used effectively to reduce the width and qubit dependencies in a circuit, leading to improved execution fidelity for larger circuits on small QPUs, mostly at the cost of computational overheads, as shown in previous ad-hoc and theoretical work [44, 60, 70, 89]. In total, GV gives us opportunities in the following two dimensions:

Cutting Quantum Circuits. By virtualizing binary gates, a circuit can be divided into smaller subcircuits, each with a lower number of qubits which can be run independently on small and noisy QPUs [44, 60]. Circuits with lower widths exhibit less qubit mapping and routing constraints, therefore less post-transpilation CNOT operations and lower depth [49]. Moreover, as recent work shows, the wire-cutting technique [60, 77] can also be modeled using gate virtualization

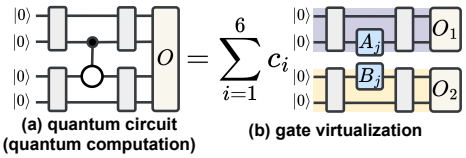

Figure 1. Computational model and gate virtualization (§ 2.2). *(a) A quantum computation to estimate the expectation value of an observable O. (b) Virtualizing a two-qubit gate by computing a weighted sum over circuit instances with single-qubit gates inserted ($O = O_1 \otimes O_2$)*

[16]. Therefore, a system that is able to virtualize gates is a complete solution for circuit cutting and knitting.

Reducing Qubit Dependencies. Gate virtualization can be used to virtualize binary gates that cause qubit dependencies. Virtualizing them and therefore reducing qubit dependencies in a circuit has three-fold advantages: Firstly, by reducing the propagation of errors through gates and qubits, fidelity can be improved. Secondly, reducing intra-circuit dependencies facilitates optimized qubit mapping and routing on QPUs during the transpilation process, which leads to lower depth and number of CNOTs. Lastly, fewer qubit dependencies enable the application of qubit-reuse [33], which could enable a computationally efficient way to reduce circuit width.

To summarize, GV is a promising technique to improve the execution accuracy and scaling of quantum circuits, as shown in small ad-hoc examples in previous work. However, to fully exploit the benefits of GV on arbitrary circuits despite the exponential post-processing overhead, we need an automatic and efficient application as well as a scalable execution of GV.

3 Overview

We aim to design a system that can scale the sizes of circuits that can practically be executed on QPUs with high fidelity. To realize this goal, gate virtualization (GV) is a promising technique that decomposes circuits into smaller circuit fragments or reduces the intra-dependencies in the circuit. However, dealing with the programming and exponential computational complexity of GV is challenging. We next discuss these challenges and present our key ideas to address them.

3.1 Design Challenges and Key Ideas

Challenge #1: Programmability and Generality. The promising technique of GV is a new and rather complex concept. It is not trivial how to virtualize two-qubit gates using single-qubit gates or how to keep track of the created circuit fragments. Therefore, we must develop general abstractions that implement the new virtualization techniques and allow simple, automatic application to quantum circuits while allowing straightforward integration into existing transpilation and optimization infrastructures.

Virtual Circuit IR (VC-IR) as an Intermediate Representation: We introduce the *virtual circuit IR* (VC-IR) to enable a unified optimization and execution process of large circuits using gate virtualization. The VC-IR is an intermediate step between any high-level circuit representation and smaller optimized circuit fragments.

Challenge #2: Fidelity. The noisy circuit executions on QPUs hinder the practicality of current quantum algorithms. Every operation applied on a qubit incurs noise to the final result, which propagates and amplifies throughout the circuit. To ensure higher fidelity in quantum computations, it is essential to employ procedures that optimize the circuit's structure using the promising technique of gate virtualization. This involves decomposing the circuit into smaller fragments, reducing the circuit's depth, and minimizing the number of non-local operations or qubit dependencies while minimizing the overhead of virtualization.

A Compiler for Optimal Gate Virtualization: We introduce the *QVM Compiler*, a modular architecture designed to compile circuits utilizing gate virtualization. The compiler converts a quantum circuit into a VC, applies a customizable series of optimization passes on the VC to take advantage of gate virtualization opportunities, and prepares the VC for execution on a distributed set of QPUs.

Challenge #3: Scalability. Gate virtualization incurs an exponential overhead of $O(6^k)$ for k virtual gates, both in quantum computation and in classical postprocessing (Fig. 1). This overhead appears since we need to execute the fragments in $O(6^k)$ instantiations, and then we need to post-process all instantiation results to compute the final result. To maximize the possible gate virtualizations, it is crucial to implement highly parallel computation on multiple QPUs and classical processors.

A Distributed Scalable Runtime: We present *QVM Runtime*, a scalable system for executing virtual circuits. The runtime efficiently instantiates the high amount of fragments, distributes them between available QPUs for parallel quantum processing, and uses a highly scalable parallel process to post-process the fragment results.

3.2 The QVM Framework

Based on the aforementioned key ideas, we propose the design of our Quantum Virtual Machine (QVM) framework, an end-to-end system that exploits the full potential of gate virtualization to achieve scalable execution of large circuit with high fidelity (see Fig. 2). The QVM system builds on the abstraction of a *virtual circuit* to utilize gate virtualization. It consists of two main components: the QVM Compiler (§ 4) and the QVM Runtime (§ 5).

QVM Virtual Circuit IR (VC-IR). The virtual circuit (VC) abstraction extends the traditional quantum circuit abstraction (§ 4.2). For this, it incorporates the abstraction of *virtual gates* and views the circuit as a collection of circuit *fragments*,

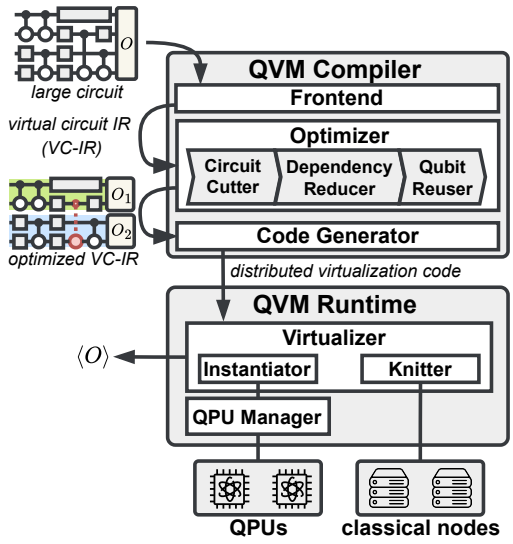

Figure 2. Overview of the QVM framework (§ 3.2). *The Quantum Virtual Machine (QVM) consists of two main components: QVM Compiler and QVM Runtime.*

where each fragment is a circuit acting on a subset of qubits of the original circuit.

QVM Compiler. The QVM compiler (Fig. 2, top) is responsible for compiling a quantum circuit efficiently to a set of smaller circuit fragments by using gate virtualization. The compiler operates in three stages: (1) the *frontend* converts the circuit into the VC-IR, (2) the *virtual circuit optimizer* applies gate virtualization to reduce circuit depth, width, and/or intra-circuit dependencies, and (3) the *distributed transpiler* prepares the circuit fragments for execution on a set of QPUs. For the virtual circuit optimizer, we describe the implementation of three optimization passes of the *circuit cutter*, the *dependency reducer*, and the *qubit reuser*, which are designed to optimize arbitrary virtual circuits. Additionally, users can easily plug in their own optimization passes, which modify a virtual circuit, e.g., to efficiently optimize specific circuits of a known structure.

QVM Runtime. The QVM runtime (Fig. 2, bottom) is the system responsible for the scalable execution of virtual circuits. The runtime consists of the two components *virtualizer* and the *scheduler*. The virtualizer is responsible for implementing the gate virtualizations according to Fig. 1, using fragment instantiation and parallel post-processing. The QPU manager is responsible for the parallel execution of the fragments on a set of distributed QPUs.

4 The QVM Compiler

We now describe the design and implementation of our QVM compiler. The QVM compiler is an extensible pipeline for the efficient virtualization of gates and to prepare a large circuit for executing a set of small QPUs.

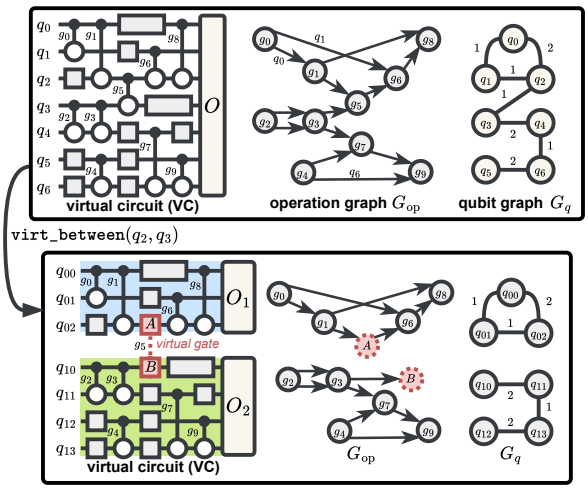

Figure 3. Virtual Circuit IR (VC-IR) (§ 4.2). *A virtual circuit (VC) extends a quantum circuit by incorporating virtual gates and managing the qubits in sets of fragments. The VC-IR manages the operation graph G_{op} and the qubit graph G_q for efficient analysis and manipulation with the VC-IR API to apply GV.*

4.1 Workflow of the QVM Compiler

Fig. 2 (top) shows the workflow of the QVM compiler. First, the **frontend** of our compiler takes a (large) quantum circuit and converts the circuit into the virtual circuit IR (VC-IR) (Fig. 3).

Then, the VC-IR is optimized using the **optimizer**. Each compiler optimization pass receives two inputs: the maximum fragment size s, which specifies the maximum width each fragment must have, and a virtualization budget b, which constrains the number of allowed gate virtualizations to limit the maximum virtualization overhead. Typically, we choose s as the size of the largest available QPU to ensure every fragment is executable by at least one QPU. For our optimizer, we design a pipeline of the following three generic optimization passes:

#1: Circuit Cutter (CC): First, the circuit cutter (CC) pass (§ 4.3) aims to decompose the VC into fragments smaller than s, using $v \leq b$ virtual gates, and sets the budget to $b = b - v$. If CC fails to decompose the circuit within the given budget, no gate is virtualized and we forward the VC to the next stage.

#2: Dependency Reducer (DR): In the case where the budget b is not yet exhausted by the circuit cutter pass, the dependency reducer (DR) (§ 4.4) applies up to b gate virtualizations to reduce the dependencies between qubits and operations within the VC to reduce noise propagation and circuit depth.

#3: Qubit Reuser (QR): Lastly, the qubit reuser (§ 4.5) reuses qubits within individual fragments to further reduce circuit width if the CC fails to reduce the fragment sizes sufficiently. If the qubit reuser fails to reduce the width of any fragment to s, the optimization pipeline fails.

After the optimization phase, the **code generator** (CG) acts as the backend of our compiler (§4.6) by extracting the fragments as parameterized circuits, optimizing the circuits and generating the inputs for the instantiation of each fragment.

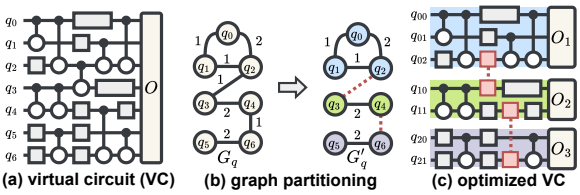

Figure 4. Circuit Cutter (CC) (§ 4.3). *The CC receives a large virtual circuit (VC) with $n_q = 6$ qubits and performs a graph partitioning on the qubit graph G_q to dissect the VC into fragments of size $s = 3$ by inserting virtual gates between the partitions.*

Next, we describe the QVM compiler stages in detail.

4.2 Virtual Circuit IR and Frontend

To enable easy integration and a simple workflow for gate virtualization during compilation and runtime, QVM provides the virtual circuit IR (VC-IR) (Fig. 3). In total, the VC-IR provides three main data structures: (1) A *virtual circuit* (VC), which can contain *virtual gates* and consists of several *fragments*, (2) an operation graph G_{op} and (3) a qubit graph G_q, as described below:

Virtual Circuit and Virtual Gates. A VC extends the traditional abstraction of a quantum circuit by additionally incorporating the functionality of consisting of a set of fragments and allowing *virtual gates* to be part of its instructions.

A *fragment* describes a subset of qubits that are not connected to other qubits in the VC via a real two-qubit gate. We implement fragments by using a separate qubit register for each fragment.

A *virtual gate* expresses the notion of the virtualization of a binary quantum gate (§ 2.2). A virtual gate is a binary quantum gate that does not require a real connection between its two qubits. Therefore, a conventional transpiler or circuit optimizer would treat a virtual gate as two one-qubit gates. Hence, a virtual gate has no influence on, e.g., the assignment and routing of qubits. A virtual gate can be split into two parameterized one-qubit gates, whose instantiations are inserted during execution (§ 5.2).

Operation Graph. The operation graph G_{op} expresses the gate dependencies of the VC as a directed acyclic graph (DAG). G_{op} is a graph in which the vertices are the two-qubit gates of the circuits, and the edges represent the direct dependencies between the respective operations via a qubit wire. Therefore, each edge contains the respective qubit as an attribute.

Qubit Graph. To efficiently represent the connections between qubits of a VC, we utilize the representation of a qubit graph G_q, where the qubits are the vertices. An edge exists between two qubits when the qubits are connected with at least one two-qubit gate. So, the connected subgraphs of G_q directly correspond to the VC's fragments. Each edge holds a weight with the number of two-qubit gates between the two qubits.

Gate Virtualization API. To efficiently virtualize gates, the VC-IR exposes two main functions:

- `virt_gate`(g_x): Virtualizes the gate g_x, removes g_x from G_{op} and adds single-qubit gates instead. Decrements the weight on the edge (q_i, q_j) in G_q, where g_x acts on q_i and q_j.
- `virt_between`(q_i, q_j): Virtualizes every gate which acts on the qubits q_i and q_j. Removes the edge (q_i, q_j) from G_q, and updates G_{op} accordingly.

Fig. 3 shows an example of calling `virt_between`(q_i, q_j).

Frontend: Virtual Circuit Generation. The frontend of the QVM compiler generates the VC-IR from an input circuit. The VC is initially a copy of the original circuit, i.e. a VC that consists of one fragment and no virtual gates. We generate G_{op} and G_q by traversing the operations of the original circuit.

4.3 Circuit Cutter (CC)

The aim of the Circuit Cutter (CC) optimization pass is to split the VC into several fragments so that each fragment has s or fewer qubits, while using as minimal virtual gates as possible to minimize the computational overhead (Fig. 4). For this purpose, the CC performs a graph partitioning on the qubit-graph G_q as follows:

Circuit Cutter Model. We assign the vertices $q_x \in V_q$ of the qubit graph $G_q = (V_q, E_q)$ into at least $f = \lceil n_q/s \rceil$ subsets F_j. According to this mapping, $E_{cut} = \{(q_x,q_y) : q_x \in F_j, q_y \in F_i, F_j \neq F_i\}$ is the set of all edges that need to be removed to decompose the G_q into independent subgraphs. In our cutting model, we find a solution that minimizes $\sum_{(q_x,q_y)\in E_{cut}} w(q_x,q_y)$, where $w(q_x,q_y)$ is the weight of the respective edge $(q_x,q_y) \in E_{cut}$. Amongst all possible optimal solutions that amount to the weight, we choose a solution that minimizes $\sum_j |F_j|^2$, such that we favor the solution that distributes the number of qubits evenly across the fragments. The subsets F_j correspond to fragments of the resulting optimized VC.

We implement the model with Answer Set Programming (ASP) using the Clingo solver to find an optimal solution [1, 26]. For each $(q_x,q_y) \in E_{cut}$ we call `virt_between`(q_x, q_y) to update the VC-IR according to the solution of the model.

Greedy Circuit Cutter. In addition to the procedure that finds an optimal solution for our model, we also implement a CC that uses an efficient heuristic approach based on the greedy Kernigan-Lin bisection algorithm [35] to enable shorter compilation times for large circuits. To decompose a VC into multiple fragments of size s or less, we iteratively apply the Kernighan-Lin bisection to the currently largest connected subgraph of G_q. The bisection determines two distinct sets of vertices V_1 and V_2 such that $|V_1| \approx |V_2|$, and the sum of weights of the edges between the two sets of vertices is as minimal as possible. Then we call `virt_between`(q_x, q_y) for each (q_x, q_y), where $q_x \in V_1$ and $q_y \in V_2$. We apply this iteration until each fragment of the VC has less than s qubits.

Note that in the partitioning algorithms for gate virtualization, the search space scales only one-dimensionally with the number of qubits in the circuit and not also with the number of

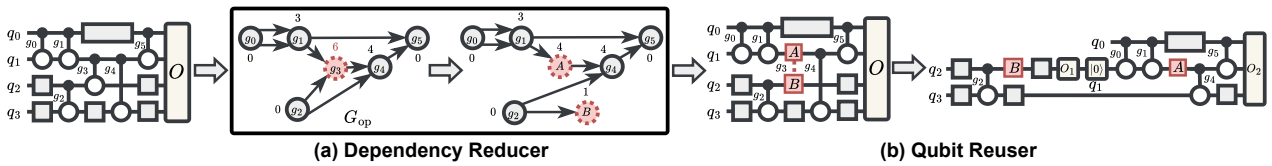

Figure 5. Dependency Reducer (DR) and Qubit Reuser (QR) (§ 4.4-4.5). *(a) The greedy DR iteratively virtualizes gates to reduce the number of qubit-dependencies in a circuit. (b) Because of reduced qubit dependencies, the QR can reuse qubits to reduce the circuit width.*

binary gates in the circuit, as is the case with wire-cutting [77]. Since the number of binary gates in a circuit is typically much larger than the number of qubits, our gate-cutting techniques are generally much more efficient than their wire-cutting counterparts. This makes optimal graph partitioning for gate virtualization a suitable option for current quantum algorithms with hundreds of qubits and few partitions, where the circuit cutting time is negligible compared to execution time.

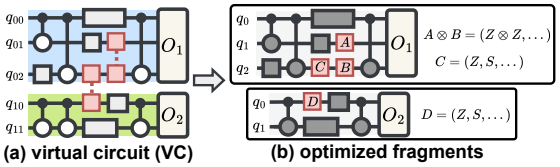

Figure 6. Code Generator (CG) (§ 4.6). *The CG generates distributed virtualization code in form of parameterized fragments.*

4.4 Dependency Reducer (DR)

The Dependency Reducer (DR) reduces the number of circuit intra-dependencies circuit using as few virtual gates as possible

The dependencies between qubits and operations are best illustrated with the VC-IR's operation graph G_{op}. An example of a G_{op} in the context of qubit dependencies is shown in Fig. 5 (a). In this example, every qubit q_i is dependent on every other qubit q_j, since some gate g_x acting on q_i depends on a gate g_y acting on q_j [33]. This means that noise occurring on one qubit could also propagate to all other qubits in the circuit, amplifying overall errors.

As shown in Fig. 5, the DR can reduce the intra-dependency of the circuit by inserting virtual gates into the circuit while being constrained by the budget b of the maximum gate virtualizations. To reduce the qubit-dependencies as efficiently as possible, we adhere to the following model:

Dependency Reducer Model. We aim to minimize the number of qubit-dependencies by virtualizing gates in the VC-IR. A qubit q_i is dependent on another qubit q_j if there exists a path in G_{op} from a gate g_x acting on q_j to a gate g_y acting on q_i. Let $D_q = \{(q_i, q_j) : q_i \text{ depends on } q_j\}$ be the set of all qubit-dependencies. We need to find a set G_{virt} of gates that, when removed from G_{op}, minimize the number of qubit dependencies $|D_q|$ the most. If multiple optimal solutions exist, we choose a solution that minimizes $|G_{virt}|$. We implement this model using Answer Set Programming (ASP) and use the Clingo solver to solve for an optimal solution [1, 26].

Greedy Dependency Reducer. For circuits with a large number of gates, we additionally design an efficient greedy DR (G-DR) algorithm (Fig. 5). The algorithm works as follows:

First, we determine the most critical binary gate in the circuit, i.e., the binary gate that, when virtualized, can reduce the most intra-circuit dependencies. For this, we label every binary gate g_x in the circuit with cost d_i. This cost is defined as $d_i = anc(g_x) \cdot desc(g_x)$, where $anc(g_x)$ is the number of ancestors, and $desc(g_x)$ is the number of descendants of g_x. Therefore, the gate with the highest cost depends on most other gates in the circuit and is, therefore, most likely responsible for a large amount of qubit and gate dependencies. Then, we call `virt_gate`(g_x) on the most-critical gate g_x to virtualize the gate in the VC and remove the two-qubit gate from G_{op}. If two or more gates have the same cost, we choose a gate randomly. We decrement the budget b, and repeat the process until $b = 0$.

In the example from Fig. 5 (a), the G-DR would virtualize g_3 first, since it has the highest single cost of $d_3 = anc(g_3) \cdot desc(g_3) = 3 \cdot 2 = 6$. In this single iteration of G-DR, we can reduce the number of qubit dependencies from 12 to 11 since now q_2 does not depend on q_1 anymore. This means that errors of q_1 cannot propagate to q_2. It also reduces the cost of all the gates in the circuit, meaning that the gates depend on significantly fewer other gates and are, therefore, less likely to amplify overall noise.

Note that our G-DR computes the number of ancestors for each node in a single traversal of G_g in topological order. Similarly, the number of descendants of each node is computed in a single traversal in reversed order. Therefore, the time complexity of G-DR is $O(2 \cdot n_v \cdot |V_g|)$, where $|V_g|$ is the set of nodes in G_g. Thus, the algorithm has linear time complexity in the number of gates in the circuit.

4.5 Qubit Reuser (QR)

In the final pass of the optimizer, we apply the qubit reuser on individual fragments to reduce their width further, in case their width still exceeds the maximal size s. To this end, the qubit reuser first checks whether each fragment in the VC has a width of s or less. For each fragment with a width greater than s, the qubit reuser applies a qubit reuse procedure to reduce the width to s to ensure that each fragment can execute on the available QPUs. We can reuse a qubit q_i for another qubit q_j if q_i does not depend on q_j by inserting a mid-circuit

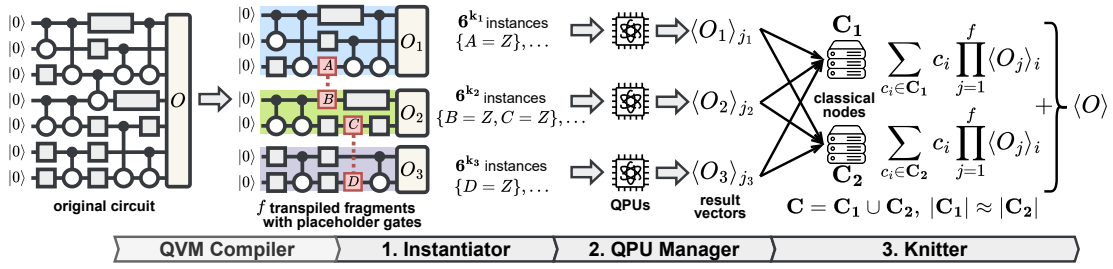

Figure 7. Workflow of the QVM Runtime (§ 5). *(1) The instantiator generates the instantiations inserted into the placeholder gates of the compiled fragments. (2) The QPU manager runs the instances on distributed QPUs in parallel. (3) The knitter reconstructs the probability distribution of the original circuit by merging and then knitting the instances in highly parallelizable steps.*

measurement and resetting the qubit [33]. Fig. 5 (b) shows this qubit reuse pass, where we can reuse q_2 for q_1 since q_2 does not depend on q_1, with $O = O_1 \otimes O_2$

The appropriate level of qubit reuse may only be possible through the preceding DR pass, which reduces the number of dependent qubit pairs as shown by the example of Fig. 5. Similar reduction in width by circuit cutting would have required two virtual gates (or two wire cuts [60, 77]). Therefore, in our example, reducing dependencies and reusing qubits is the most efficient solution in terms of virtualization overhead: we reduce the width of the circuit to $s = 3$ with a virtualization budget of only $b = 1$. Note that the QR does not affect the execution of the VC on a distributed set of QPUs, since the QR only considers the reuse of qubits within the same fragment.

4.6 Code Generator (CG)

The final step of the QVM compiler is generating the code in the form of circuits, which can be executed by the QVM runtime (Fig. 6). To do so, we first extract each fragment as an individual circuit from the VC by collecting all operations on the respective qubit register. In these extracted circuits, we insert placeholder gates at the qubits of the virtual gates. The placeholder gates are parameterized gates, which can be instantiated with the actual gates that we need to insert to reconstruct the result (§ 2.2). For the instantiation, the CG creates a parameter vector for each placeholder gate, which describes the gates to be inserted for the respective instance of the virtual gate. Finally, the code-generator runs a set of standard circuit optimization passes to optimize the individual circuits. This means the heavy optimization must be executed only once, reducing the just-in-time transpilation time before instance execution.

5 The QVM Runtime

We next describe the QVM runtime, a system for the scalable execution of virtualized circuits.

5.1 Workflow of the QVM Runtime

Fig. 7 shows the workflow of the QVM runtime. As the first step, we pass optimized fragment circuits generated by the QVM compiler to the *virtualizer*. Here, the *instantiator* generates the instances of each circuit and passes them together with the fragments to the QPU manager. The *QPU manager* then executes the fragments with each given instantiation on the distributed set of QPUs. The results are returned to the *knitter* component of the virtualizer, where the final result is reconstructed through parallel classical post-processing.

5.2 Virtualizer

The virtualizer implements the logic for executing virtual gates. For this purpose, the virtualizer consists of two components, the *instantiator* (Fig. 7, Step 1) and the *knitter* (Fig. 7, Step 3).

Instantiator. The instantiator is responsible for creating instances of gates that must be inserted into the fragment. For this purpose, the instantiator creates 6^{k_j} instances for each fragment F_j, where k_j is the number of virtual gates that act on the qubits of F_j. The instances are described as assignments to the parameterized gates and include every possible combination of the total 6^{k_j} combinations of each fragment. These assignments are essentially the tensor-product of the parameter vectors of the generated code for each fragment (Fig. 6).

Knitter. The knitter takes the results of the probability distribution of all fragment instances and calculates the final result of the original circuit by applying the formulas for gate virtualization with highly parallel processing. (Section 2.2). The results are given as vectors for each fragment F_j with entries $\langle O_j \rangle_i$ with $i = 1,\ldots,6^{k_j}$. To knit the results, the knitter distributes the result vectors of each fragment to the available classical nodes, where each node is given the task of computing a part of the global 6^k instances. We determine this part by assigning an equal part of the global coefficient vector $\mathbf{C}$ to each node (Eq. 2). In the example of Fig. 7, we divide the coefficient vector $\mathbf{C}$ into two parts $\mathbf{C_1}$ and $\mathbf{C_2}$ and calculate the partial sum at each node over the instances corresponding to each coefficient. Finally, we calculate the sum of the two partial results to obtain the final result $\langle O \rangle$. In this way, we are able to linearly scale the post-processing of the circuit virtualization with respect to the number of cores used.

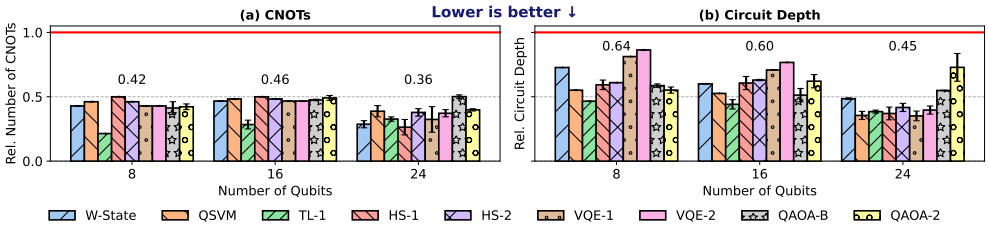

Figure 8. Circuit Cutter (§ 6.1). *Impact of QVM's optimal circuit cutter compiler on the number of CNOTs and circuit depth.*

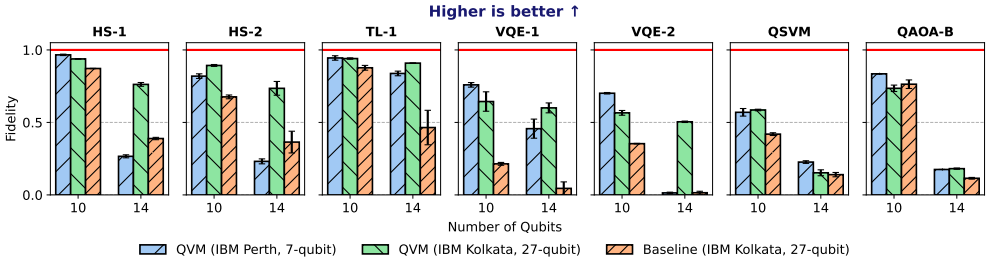

Figure 9. Circuit Cutter (§ 6.1). *Fidelity of running QVM with the circuit cutter compiler on IBM Perth and IBM Kolkata.*

Extensibilty. We implement a virtualizer for gate virtualization as presented in [44]. However, the design of our virtualizer also allows us to implement other divide-and-conquer techniques effectively [8, 77]. Such techniques all follow the same workflow of our virtualizer and could, therefore, be easily integrated into the QVM runtime.

5.3 QPU Manager

The QPU manager is responsible for a scalable execution of the 6^{k_j} instances of each circuit fragment F_j on a set of distributed QPUs, returning the result-vector for each fragment (Fig. 5, Step 2). For this, the QPU manager receives an optimized circuit fragment (§ 4.6) and all instance combinations generated by the instantiator. To execute a fragment, the QPU-manager does the following steps:

Step 1: For each QPU QPU_i with enough qubits to run the circuit, we transpile the circuit to, including mapping and routing on the physical qubits of QPU_i. Note that this has to be done only once for each We then compute the estimated probability of success $esp(QPU_i)$ of executing the circuit on that QPU. This is done by computing the cost of the errors induced by the gates and measurements on the assigned physical qubits, as described in [49].

Step 2: We normalize the current job queue sizes of the QPUs by dividing the length of each job queue by the length of the maximum job queue. This yields a relative waiting time as $w(QPU_i) \in [0,1]$, where a higher value means a longer waiting time for the job.

Step 3: We compute the score s_i for each QPU, where $c_i = \alpha \cdot (1 - w(QPU_i)) + \beta \cdot esp(QPU_i)$, and choose the QPU with the highest score to execute the corresponding fragment. The user can choose α and β to provide either fast runtime or less noisy results.

Step 4: Finally, for each instance combination, we insert the instantiation into the transpiled fragment for the selected QPU, resulting in a total of 6^{k_j} circuits when k_j gates act in the respective fragment F_j. These circuits are then sent to the QPU as a job for execution, and the results are returned to the virtualizer.

Our strategy of incorporating queue times and estimated probabilities of success into the QPU manager can be easily applied to the current cloud-centric quantum infrastructure, where our QPU manager would be a client for some quantum resources offered by cloud providers [67]. Our solution is currently the most efficient, as there is little control over the cloud providers' internal queues.

6 Evaluation

Experimental Setup. We conduct three types of experiments: (1) circuit transpilation with and without QVM's compiler to measure the circuit's properties post-compilation, (2) runs on real QPUs for measuring the circuit's fidelity, and (3) classical simulation of large circuits cut into fragments of different sizes. For (2) we conduct our experiments on Falcon r5.11H QPUs, namely the 7-qubit IBM Perth and the 27-qubit IBMQ Kolkata. For (1) and (3) we use the Qiskit Transpiler and Qiskit Aer [4, 5], respectively, and run on our local classical machines. For classical tasks, i.e., transpilation, post-processing (knitting), and simulation, we use a server with a 64-core AMD EPYC 7713P processor and 512 GB ECC memory.

Framework and Configuration. We use the *Qiskit* [64] Python SDK version 0.41.0 for quantum circuits and simulations. We transpile any quantum circuit we run with the highest optimization level O3 and run with 20,000 shots. To get a meaningful measurement of the fidelity or circuit properties on real QPUs, we run QVM only on a single QPU. When we benchmark the performance of the QVM runtime with simulators, we utilize every system core.

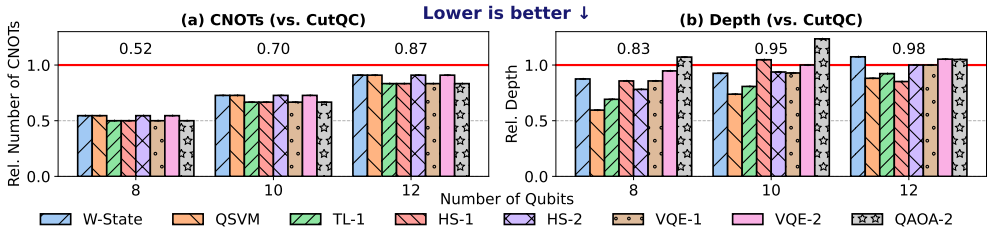

Figure 10. Circuit Cutter vs. CutQC (§ 6.1). *Relative number of CNOT gates and fragment depth compared to CutQC on IBM Kolkata.*

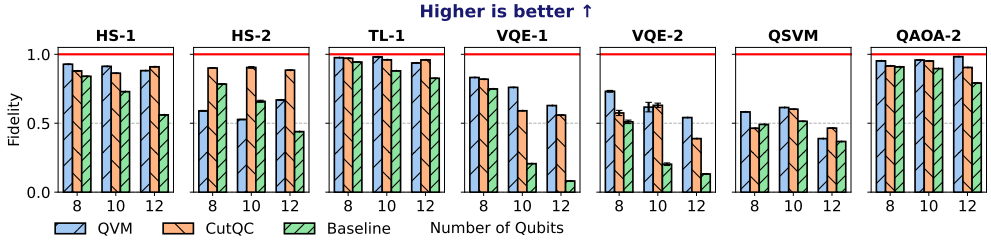

Figure 11. Circuit Cutter vs. CutQC (§ 6.1). *Fidelities of running QVM vs. CutQC vs. Qiskit baseline on IBM Kolkata.*

Benchmarks. We study QVM on a set of circuits used in the state-of-the-art benchmark suits Supermarq [82], MQT-Bench [65], and QASM-Bench [37]. These circuits can be scaled both in the number of qubits and depth. Specifically, we study: W-State, Bernstein Vazirani (BV) , Quantum Support Vector Machine (QSVM), Hamiltonian Simulation (HS-t), Two Local Ansatz (TL-n) with circular entanglement, Variational Quantum Eigensolver (VQE-n) with a Real-Amplitudes ansatz of linear entanglement, Approximate Optimization Algorithm (QAOA-d) with regular graphs of degree $d \in \{2,3,4\}$ and barbell graphs (QAOA-B). HS, VQE, and TL are scalable in their circuit layers t or n, respectively.

Metrics. We evaluate the following metrics.

- **Fidelity**: We use the *Hellinger fidelity* to measure how close a noisy result is to the desired ground truth of a quantum circuit. The Hellinger fidelity is calculated as $\left(1-H\left(P_{ideal},P_{noisy}\right)^2\right)^2 \mapsto [0,1]$, where H is the Hellinger distance between two probability distributions [32].
- **Circuit Properties**: Number of *CNOT* gates, *depth* and the number of qubit *dependencies*. When a VC contains more than one fragment, we use the fragment with the *worst* property (i.e., maximal depth, dependecies, number of CNOTs)
- **Execution Time**: The execution time of a VC in seconds.
- **Estimated Success Probability**: We use the estimated success probability (ESP) metric to measure the estimated fidelity on larger quantum systems. We define the ESP as $\prod_i (1-e_i)$, where e_i is the error of the i-th operation in the circuit [48]. If a VC has multiple fragments, we report the minimum ESP.

Baseline. We use the Qiskit transpiler [5] with O3 and CutQC as our baselines for circuit compilation and runtime evaluation [10, 77].

6.1 Circuit Cutter

RQ1: *How well does QVM's circuit cutter allow scaling of circuits that can run on noisy QPUs with acceptable fidelity?* We evaluate the impact of the circuit cutter on the CNOT count and depth of transpiled circuits and the fidelity of running virtual circuits using our optimal graph partitioning model.

Impact on Number of CNOTs and Circuit Depth. In Fig. 8, we study the maximum number of CNOTs and circuit depths of the fragments after compilation with our circuit cutter with a maximum of three virtual gates. Each virtual circuit is decomposed into fragments of a maximum of 13 qubits, and the fragments are transpiled for the 27-qubit IBMQ Kolkata QPU. The results in Fig. 8 (a) show that the number of CNOTs decreases by 41% on average. Fig. 8 (b) shows that the circuit depth decreases by 56% on average. This shows that it is possible to almost double the size of the circuits running with high fidelity on the given QPU since the number of CNOTs and circuit depth is approximately halved.

Impact on Fidelity. The impact of using QVM on the fidelity of the execution is shown in Fig. 9. Here, the circuit cutter decomposes the circuits into fragments of maximally 7 qubits in order to theoretically fit the small 7-qubit IBM QPUs. The fragments are run on both the 7-qubit IBM Perth and the 27-qubit IBM Kolkata QPUs, and compared to the baseline fidelity of running the circuits on IBM Kolkata. We run the experiment for various benchmarks with sizes of 10 and 14 qubits. We observe that the fidelity of running the circuit on IBM Kolkata improves the fidelity by 4.7× on average and up to 33.6×. E.g. for the VQE-2 benchmark, the fidelity of the benchmark goes to zero, while QVM can still create higher fidelities. Compared to the baseline, running QVM on the IBM Perth improves the fidelity by 2.1× on average and up to 10.6×. Therefore we show that QVM can reliably simulate a larger QPU using smaller noisy QPUs while producing higher fidelity. This is

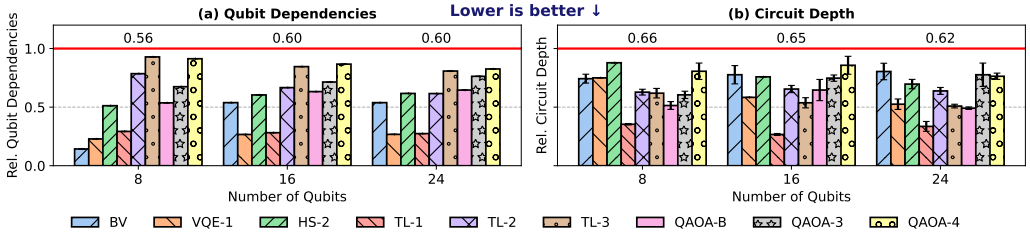

Figure 12. Dependency Reducer (§ 6.2). *Impact of the greedy qubit dependency reducer on (a) the number of qubit dependencies and (b) on the circuit depth of the transpiled circuit for IBM Kolkata. We use at most three virtual gates to compile the circuit.*

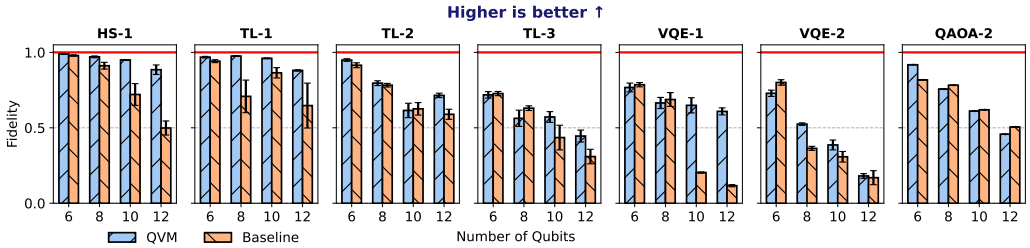

Figure 13. Dependency Reducer (§ 6.2). *Fidelity of the greedy dependency reducer using one virtual gate on IBM Kolkata.*

despite IBM Perth having a median of 2.3× higher readout and 1.2× higher CNOT error during our experiments.

Comparsion to CutQC. In Fig. 10 and 11 we compare the circuit cutter of QVM with CutQC. We run the QVM and CutQC circuit cutters with the same configuration to generate circuit fragments of up to 7 qubits and compile and run the fragments on the IBM Kolkata QPU. We use several benchmarks of sizes of 8-12 qubits. We find that, compared to CutQC, QVM only produces 70% of the CNOTs on average, since gate virtualization allows a reduction of the qubit connectivity significantly compared to CutQC (Fig. 10 (a)). QVM achieves similar circuit depth reduction as CutQC as both can cut the circuits into significantly smaller fragments (Fig. 10 (b)).

A look at the fidelity benchmark (Fig. 11) shows that CutQC and QVM achieve similar fidelity and significantly outperform the Qiskit baseline, with QVM achieving on average 1% higher fidelity than CutQC. We suspect the relatively small improvement despite the promising results in circuit properties is due to the noisy mid-circuit measurements with an error of $\geq 10^{-2}$, which we need to perform to virtualize gates. With less measurement noise, QVM will likely perform similar or better to CutQC.

We conclude that both QVM and CutQC, with their different techniques, are efficient in-circuit cutting and should ideally be used together in future work to take advantage of both methods with their respective benefits [13], especially since we will likely be able to mitigate the mid-circuit measurement errors [30].

RQ1 takeaway: With the circuit cutter, we reliably scale the size of circuits that can be run on noisy QPUs, up to 2×, improving the overall fidelity 4.7× on average and up to 33.6× due to significant depth and CNOT gate reduction.

6.2 Dependency Reducer

RQ2: *By how much does the dependency reducer (DR) decrease the number of dependencies within the circuit, improving the fidelity of running the circuit on noisy QPUs?* For this experiment, we evaluate DR with a maximum of three virtual gates on several benchmarks with different circuit sizes on IBM Kolkata.

Impact on Qubit Dependencies and Circuit Depth. Fig. 12 (a) shows the effect of DR on the number of qubit dependencies in the logical circuit, compared to the baseline of the circuit without DR. On average, the number of qubit dependencies decreases by 58%. This shows that the DR can effectively resolve the dependencies between qubits, reducing noise propagation through the circuit. As Fig. 12 (b) shows, the depth of the circuits transpiled for IBM Kolkata decreases significantly by 64% on average. This is due to the transpiler having fewer constraints on circuit mapping and routing after applying DR, resulting in a transpiled circuit with less depth.

Impact on Fidelity. We analyze the fidelity of our baseline and compared it to the DR in Fig. 13, utilizing only one virtual gate. Our results indicate an average increase in fidelity of 36% and up to 5.2×. However, the noisy mid-circuit measurements needed for gate virtualization could limit the improvement in fidelity. These measurements typically induce significant noise, which affects the overall fidelity of virtual circuit execution [73, 89].

RQ2 takeaway: The DR decreases the dependencies between qubits by 58% and circuit depth by 64% using at most three virtual gates. This also leads to an average increase in fidelity by 36% and up to 5.2×, using only one virtual gate.

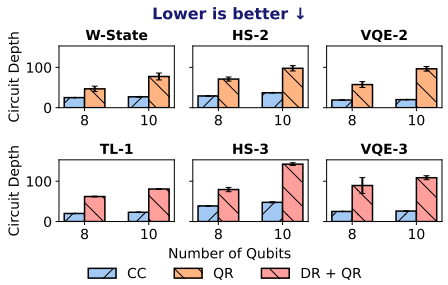

Figure 14. Qubit Reuser (§ 6.3). *Depths of compiled circuits with a maximal fragment size of 5 transpiled for IBM Perth. (a) Comparison of the circuit cutter vs. qubit reuser. (b) Comparison of the circuit cutter vs. dependency reducer and qubit reuser.*

6.3 Tradeoffs with the Qubit Reuser

RQ3: *What is the effect of using the qubit reuser (QR) to reduce the width of the circuit fragments further?* We show the tradeoffs of using the CC alone against the DR and QR to reduce the width of circuits to run on small QPUs. To show this tradeoff, we compile circuits with different optimizer configurations, such that each fragment's width is maximally five qubits.

Circuit-Cutter vs. Qubit-Reuse. In Fig. 14 (top), we compare the effects of using either the CC or the QR to reduce the width of a virtual circuit on the circuit depth of the transpiled fragments. Our results show that the CC compiles the circuits to only 37% compared to QR on average. This is because the CC can break down the circuit into smaller fragments with reduced width while only incurring a maximum of two virtual gates. The QR, however, increases the depth of the circuit substantially while reusing qubits, which in turn will reduce overall fidelity. So, there is a tradeoff between using gate virtualization to reduce the depth against using qubit reuse without overhead but with more depth.

Combining Dependency Reducer and Qubit-Reuse. In Fig. 14 (bottom), we show how the CC pass compares to the DR and QR passes to reduce circuit width. For this, we choose benchmarks where, without our DR, qubit-reuse would be impossible since every qubit depends on every other qubit in the circuit. We apply the QR on the reduced-dependency circuit produced by the DR. Like before, we aim to reduce the circuit width to five qubits. The CC uses at most three, and the DR uses one virtual gate, with a 36× lower virtualization overhead. Although using DR & QR incurs a low overhead, it also leads to a significantly higher depth than CC. This means that the virtual circuit using DR & QR has 3.2× more depth than the virtual circuit of CC, which could negatively impact the fidelity.

RQ3 takeaway: We find a trade-off between overhead and noise when using the CC or DR & QR to reduce the width of quantum circuits. While the CC produces circuits with smaller depths, combining DR and QR allows lower virtualization overhead.

6.4 QVM End-to-end Runtime Analysis

RQ4: *How scalable is QVM's runtime and how does QVM compare to classical simulations without cutting & knitting and CutQC?* We study the HS-1 benchmark and use the circuit cutter (CC) to compile a VC for a QPU of up to s qubits.

Fig. 15 (a) illustrates the end-to-end runtime needed to simulate *HS-1* after cutting the circuit into fragments that fit QPUs of sizes $s \in \{15,20,25\}$. As the full circuit size increases, the runtime also increases, but the growth rate varies among fragment sizes. The smallest fragment size is the fastest, as the simulation overhead outweighs the knitting overhead, even if the circuit has 100 qubits and is cut with five virtual gates. This is evident in Fig. 15 (b) as well, which shows the runtime breakdown for simulating the 70-qubit *HS-1*. As s increases, there is a shift in the runtime from knitting to simulation time. The compilation time remains relatively constant.

Fig. 15 (c) shows the scalability of the knitter (§ 5.2) with its parallelism. We generate knit workloads for 1-4 virtual gates for the 70-qubit *HS-1* benchmark and scale the number from 1 to 32 threads. We observe near-linear scalability with an increasing number of threads, allowing a speedup of up to 25.6× for 32 threads.

We show the memory required to simulate *HS-1* with a chosen QPU size of 20 qubits in Fig. 15 (d). While the baselines, Qiskit Aer statevector, and CutQC with full definition query [77], exhibit exponentially growing memory for linearly increasing circuit sizes, QVM maintains a slightly increasing memory requirement by utilizing sparse quasi-probability distributions. In contrast, CutQC and simulations operate on tensors that need to cover the entire sample space.

Finally, in Fig. 15 (e), we compare the runtimes of QVM and CutQC. We are limited to comparing on small examples, due to CutQC's memory limitations. In particular, we perform 20-qubit circuits for *HS-1* with simulated QPUs of 8-12 qubits. We observe similar runtimes for the QPU size of 8 qubits, as QVM spends more time to simulate a larger number of circuits due to the higher circuit cost [60, 77]. However, with a QPU size of 10 and 12 qubits, QVM clearly outperforms CutQC, as it achieves a significant acceleration in knitting due to its more efficient memory utilization.

RQ4 takeaway: QVM enables simulating large circuits on classical simulators. It can handle circuit sizes of up to 100 qubits or five virtualized gates while maintaining acceptable runtime (~ 1.5 hours) and relatively very low memory consumption. QVM's knitter allows it to scale linearly through its high parallelization..

6.5 QVM at Practical Scale

RQ5: *How does QVM behave on a practical scale with circuits of hundreds of qubits?* We would need hundreds to thousands of high-fidelity qubits to demonstrate quantum advantage. However, current QPUs with hundreds of qubits cannot reliably execute circuits with tens of qubits and higher depth. To

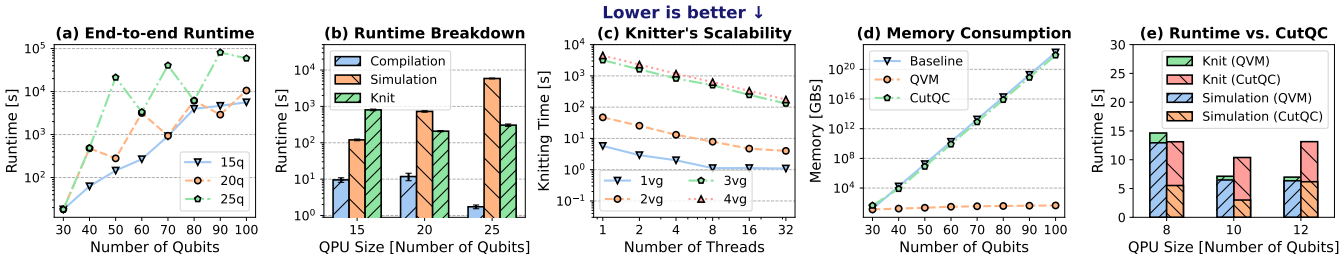

Figure 15. QVM end-to-end runtime analysis (§ 6.4). *(a) End-to-end runtime of 30-100 qubits with different QPU sizes. (b) Runtime breakdown for 70 qubits with different QPU sizes. (c) Knit-time dependent on the number of parallel threads for different numbers of virtual gates (vg). (d) Memory consumption for 30-100 qubits compared to the Baseline (statevector simulator) and Full Definition Query CutQC [77] with 20 qubits QPU size. (e) Runtime Comparison against CutQC for 20-qubit circuits.*

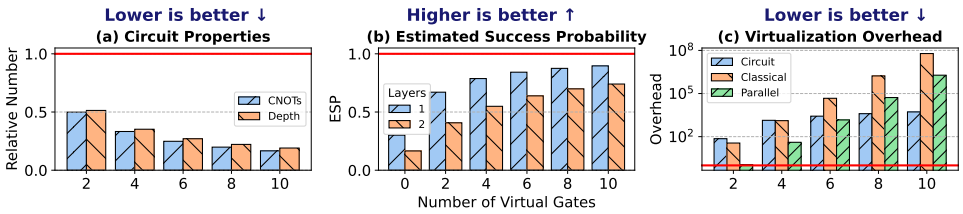

Figure 16. QVM at practical scale with 500 qubit *VQE* circuits (§ 6.5). *(a) Relative number of CNOTs and circuit depth of the compiled VQE-2 benchmark. (b) Estimated success probability (ESP) with VQE-1 and VQE-2. (c) The overheads of circuit instances and classical postprocessing with and without parallel processing on 32 cores (linear speedup).*

investigate how QVM would behave on a practical scale, we evaluate the impact of QVM on 500-qubit *VQE* circuits on a heavy-hex lattice QPU with 883 physical qubits, which is the typical chip layout for current IBM QPUs [3].

Impact on Number of CNOTs and Circuit Depth. In Fig. 16 (a) we show the effects of the number of CNOTs and the circuit depth of the *VQE-2* benchmark. We see that using a budget of two virtual gates reduces the number of CNOTs and the circuit depth by 2×, and using up to 10 virtual gates reduces the numbers by 6×. We see a diminishing improving impact on higher virtualization budgets.

Impact on Estimated Success Probability. Fig. 16 (b) shows the estimated success probability (ESP) of the benchmarks *VQE-1* and *VQE-2*. We find that the baseline without virtual gates achieves an ESP of only 30% and 16%, respectively, which leads to unusable results. When using only two virtual gates, the ESP more than doubles and shows improvements, reaching 90% and 74% with 10 virtual gates. This shows that with QVM, we only need a handful of virtual gates to significantly improve the ESP, which could lead to usable results of quantum computation.

Impact on Processing Overhead. The virtualization costs incurred using virtual gates to improve circuit fidelity are shown in Fig. 16 (c). The number of circuits that need to be instantiated and executed increases exponentially with a small number of virtual gates but then only starts to grow linearly with the number of fragments since we only instantiate as many circuits as correspond to the number of gates in the respective fragment (§ 5.2). The classical post-processing overhead grows exponentially with $O(6^k)$, meaning that adding two more virtual gates in the same configuration results in a runtime increase of 36×. Since the QVM runtime provides an almost linear speedup (§ 6.4), we can distribute the knitting across dozens of cores, which significantly mitigates this overhead for a small number of 4-6 virtual gates. This is shown in Fig. 16 (c) as an example of (perfect) linear scaling in classical post-processing with 32 cores.

RQ5 takeaway: For large-scale algorithms, QVM achieves high estimated success probability (ESP) while using only a handful of virtual gates for which our runtime can achieve significant speedups through parallelization. We therefore find a trade-off between fidelity and quantum-classical co-processing resources.

7 Related Work

Quantum Transpilers and Error Mitigation. We can categorize quantum circuit transpilation techniques as (1) qubit mapping and routing [38, 45, 46, 55, 58, 74, 79, 81, 87, 90, 92], (2) instruction/pulse scheduling [19, 29, 47, 71, 75, 83, 91] and (3) gate optimization/decomposition [18, 40, 55, 59, 71, 88]. Finally, there is work on post-execution processing, readout improvement, and error correction [14, 17, 20, 42, 43, 56, 57, 78, 80]. These proposals are orthogonal to our work and can be integrated into QVM. This is especially the case for measurement error mitigation, which can help to improve the fidelity of the mid-circuit measurements during execution [73, 89].

Circuit Cutting and Knitting. Circuit cutting & knitting is the process of breaking down a large quantum circuit into

smaller sub-circuits that can be executed separately, then synthesizing the results to obtain the result of the original circuit. Circuit cutting can be divided into gate virtualization (§ 2.2) and *wire cutting* [15, 16, 60, 77, 85]. While wire cutting optimizes circuits for small QPUs with reduced noise, it is limited in reducing qubit dependencies. Our work proposes a generic architecture for gate virtualization. Furthermore, wire cutting can be simulated using gate virtualization [10, 16].

Qubit Reuse. Qubit reuse can be classified into two categories, namely ancilla reuse using *uncomputation* [11], and reuse through *dynamic circuits* [2, 6]. Work such as [12, 23, 54] utilize uncomputation to reclaim ancilla qubits. In contrast, work such as [22, 33, 53, 69] exploit the newly supported dynamic circuits with mid-circuit measurements and mid-circuit reset operations to reuse qubits. However, applying these techniques on densely connected circuits can be impractical due to the large number of qubit dependencies [33, 82]. By first applying QVM's DR pass (§ 4.4), qubit reuse can be practically applied with enhanced efficiency.

Application-specific Optimizations. Application-specific circuit optimizations go beyond generic strategies and target the unique characteristics of a particular algorithm or circuit structure in order to improve fidelity [7, 8, 27, 28, 31, 36, 39, 68, 76, 84]. Our work tries to build a generic and extensible framework to incorporate different application-specific optimizations in the context of gate virtualization.

Quantum Cloud Computing. This area addresses quantum circuit multi-programming [21, 41, 51, 52], quantum resource management/scheduling [66, 67, 86], and quantum serverless [25, 50]. Our work is complimentary to these proposals, QVM proposes a scalable infrastructure for supporting gate virtualization optimizations, which can be incorporated by quantum cloud environments.

8 Conclusion

We introduce the Quantum Virtual Machine (QVM), a generic system for scalable, high-fidelity execution of large circuits on noisy and small QPUs by leveraging gate virtualization. QVM extends the quantum circuit abstraction with the *virtual circuit IR*, which forms the foundation for the QVM Compiler—a modular compiler infrastructure for implementing a series of optimization passes to generate smaller, optimized fragments. These fragments are virtualized and executed using our QVM Runtime—a distributed and scalable system to execute and post-process the instantiated circuit fragments in a highly parallel manner on a distributed set of QPUs. Our evaluation on IBM's 7- and 27-qubit QPUs of QVM demonstrates practical scaling of circuits with sizes up to double the QPU capacity while significantly improving fidelity.

9 Acknowledgments

We thank Karl Jansen and Stefan Kühn from the Center for Quantum Technology and Applications (CQTA)- Zeuthen for supporting this work by providing access to IBM quantum resources. We thank Martin Ruefenacht for his valuable contributions during his employment at the Leibniz Supercomputing Center (LRZ). We also thank Ahmed Darwish for his contributions to this work.

We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team. Funded by the Bavarian State Ministry of Science and the Arts as part of the Munich Quantum Valley (MQV).

References

[1] Clingo: A grounder and solver for logic programs. https://github.com/potassco/clingo. Accessed: 2023-07-17.

[2] Getting started with dynamic circuits. https://quantum-computing.ibm.com/services/programs/docs/runtime/manage/systems/dynamic-circuits/Getting-started-with-Dynamic-Circuits. Accessed: 2022-06-02.

[3] IBM Quantum — quantum-computing.ibm.com. https://quantum-computing.ibm.com/. [Accessed 28-07-2023].

[4] Qiskit AerSimulator. https://qiskit.org/ecosystem/aer/stubs/qiskit_aer.AerSimulator.html#qiskit_aer.AerSimulator. Accessed: 2022-07-07.

[5] Qiskit Transpiler. https://qiskit.org/documentation/apidoc/transpiler.html. Accessed: 2022-06-09.

[6] The IBM Quantum Development Roadmap. https://www.ibm.com/quantum/roadmap. Accessed: 2022-06-02.

[7] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. Circuit compilation methodologies for quantum approximate optimization algorithm. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 215–228, 2020.

[8] Ramin Ayanzadeh, Narges Alavisamani, Poulami Das, and Moinuddin Qureshi. Frozenqubits: Boosting fidelity of qaoa by skipping hotspot nodes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 311–324, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Marvin Bechtold, Johanna Barzen, Frank Leymann, Alexander Mandl, Julian Obst, Felix Truger, and Benjamin Weder. Investigating the effect of circuit cutting in qaoa for the maxcut problem on nisq devices. *arXiv preprint arXiv:2302.01792*, 2023.

[10] Luciano Bello, Agata M. Brańczyk, Sergey Bravyi, Almudena Carrera Vazquez, Andrew Eddins, Daniel J. Egger, Bryce Fuller, Julien Gacon, James R. Garrison, Jennifer R. Glick, Tanvi P. Gujarati, Ikko Hamamura, Areeq I. Hasan, Takashi Imamichi, Caleb Johnson, Ieva Liepuoniute, Owen Lockwood, Mario Motta, C. D. Pemmaraju, Pedro Rivero, Max Rossmannek, Travis L. Scholten, Seetharami Seelam, Iskandar Sitdikov, Dharmashankar Subramanian, Wei Tang, and Stefan Woerner. Circuit Knitting Toolbox. https://github.com/Qiskit-Extensions/circuit-knitting-toolbox, 2023.

[11] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, Nov 1973.

[12] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, 2020.

[13] Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal partitioning of quantum circuits using gate cuts and wire cuts. *arXiv preprint arXiv:2308.09567*, 2023.

[14] Sergey Bravyi, Sarah Sheldon, Abhinav Kandala, David C. Mckay, and Jay M. Gambetta. Mitigating measurement errors in multiqubit experiments. *Phys. Rev. A*, 103:042605, Apr 2021.

[15] Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading classical and quantum computational resources. *Phys. Rev. X*, 6:021043, Jun 2016.

[16] Lukas Brenner, Christophe Piveteau, and David Sutter. Optimal wire cutting with classical communication. *arXiv preprint arXiv:2302.03366*, 2023.

[17] Siddharth Dangwal, Gokul Subramanian Ravi, Poulami Das, Kaitlin N. Smith, Jonathan M. Baker, and Frederic T. Chong. Varsaw: Application-tailored measurement error mitigation for variational quantum algorithms, 2023.

[18] Poulami Das, Eric Kessler, and Yunong Shi. The imitation game: Leveraging copycats for robust native gate selection in nisq programs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[19] Poulami Das, Swamit Tannu, Siddharth Dangwal, and Moinuddin Qureshi. Adapt: Mitigating idling errors in qubits via adaptive dynamical decoupling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 950–962, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Poulami Das, Swamit Tannu, and Moinuddin Qureshi. Jigsaw: Boosting fidelity of nisq programs via measurement subsetting. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 937–949, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. A case for multi-programming quantum computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 291–303, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Matthew DeCross, Eli Chertkov, Megan Kohagen, and Michael Foss-Feig. Qubit-reuse compilation with mid-circuit measurement and reset. *arXiv preprint arXiv:2210.08039*, 2022.

[23] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 570–583, May 2020.

[24] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[25] Jose Garcia-Alonso, Javier Rojo, David Valencia, Enrique Moguel, Javier Berrocal, and Juan Manuel Murillo. Quantum software as a service through a quantum api gateway. *IEEE Internet Computing*, 26(1):34–41, Jan 2022.

[26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[27] Pranav Gokhale, Olivia Angiuli, Yongshan Ding, Kaiwen Gui, Teague Tomesh, Martin Suchara, Margaret Martonosi, and Frederic T Chong. Minimizing state preparations in variational quantum eigensolver by partitioning into commuting families. *arXiv preprint arXiv:1907.13623*, 2019.

[28] Pranav Gokhale, Yongshan Ding, Thomas Propson, Christopher Winkler, Nelson Leung, Yunong Shi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. Partial compilation of variational algorithms for noisy intermediate-scale quantum machines. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 266–278, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Pranav Gokhale, Ali Javadi-Abhari, Nathan Earnest, Yunong Shi, and Frederic T. Chong. Optimized quantum compilation for near-term algorithms with openpulse. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 186–200, Oct 2020.

[30] Riddhi Swaroop Gupta, Ewout van den Berg, Maika Takita, Kristan Temme, and Abhinav Kandala. Probabilistic error cancellation for dynamic quantum circuits. *Bulletin of the American Physical Society*, 2024.

[31] Tianyi Hao, Kun Liu, and Swamit Tannu. Enabling high performance debugging for variational quantum algorithms using compressed sensing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Ernst Hellinger. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik*, 1909(136):210–271, 1909.

[33] Fei Hua, Yuwei Jin, Yanhao Chen, Suhas Vittal, Kevin Krsulich, Lev S Bishop, John Lapeyre, Ali Javadi-Abhari, and Eddy Z Zhang. Caqr: A compiler-assisted approach for qubit reuse through dynamic circuit. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 59–71, 2023.

[34] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *nature*, 549(7671):242–246, 2017.

[35] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[36] Lingling Lao and Dan E. Browne. 2qan: A quantum compiler for 2-local qubit hamiltonian simulation algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 351–365, New York, NY, USA, 2022. Association for Computing Machinery.

[37] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2), feb 2023.

[38] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1001–1014, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. Paulihedral: A generalized block-wise compiler optimization framework for quantum simulation kernels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 554–569, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Andrew Litteken, Lennart Maximilian Seifert, Jason D. Chadwick, Natalia Nottingham, Tanay Roy, Ziqian Li, David Schuster, Frederic T. Chong, and Jonathan M. Baker. Dancing the quantum waltz: Compiling three-qubit gates on four level architectures. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[41] Lei Liu and Xinglei Dou. Qucloud: A new qubit mapping mechanism for multi-programming quantum computing in cloud environment. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 167–178, Feb 2021.

[42] Filip B Maciejewski, Zoltán Zimborás, and Michał Oszmaniec. Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography. *Quantum*, 4:257, 2020.

[43] Satvik Maurya, Chaithanya Naik Mude, William D. Oliver, Benjamin Lienhard, and Swamit Tannu. Scaling qubit readout with hardware efficient machine learning architectures. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Kosuke Mitarai and Keisuke Fujii. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New Journal of Physics*, 23(2):023021, 2021.

[45] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Qubit mapping and routing via maxsat. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1078–1091, Oct 2022.

[46] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1015–1029, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1001–1016, 2020.

[48] Paul Nation, Matthew Treinish, and Clemens Possel. mapomatic. https://github.com/Qiskit-Partners/mapomatic, 2022.

[49] Paul D Nation and Matthew Treinish. Suppressing quantum circuit errors due to system variability. *PRX Quantum*, 4(1):010327, 2023.

[50] Hoa T Nguyen, Muhammad Usman, and Rajkumar Buyya. Qfaas: A serverless function-as-a-service framework for quantum computing. *arXiv preprint arXiv:2205.14845*, 2022.

[51] Siyuan Niu and Aida Todri-Sanial. Enabling multi-programming mechanism for quantum computing in the NISQ era. *Quantum*, 7:925, feb 2023.

[52] Yasuhiro Ohkura, Takahiko Satoh, and Rodney Van Meter. Simultaneous execution of quantum circuits on current and near-future nisq systems. *IEEE Transactions on Quantum Engineering*, 3:1–10, 2022.

[53] Alexandru Paler, Robert Wille, and Simon J. Devitt. Wire recycling for quantum circuit optimization. *Phys. Rev. A*, 94:042337, Oct 2016.

[54] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. Unqomp: Synthesizing uncomputation in quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 222–236, New York, NY, USA, 2021. Association for Computing Machinery.

[55] Tirthak Patel, Daniel Silver, and Devesh Tiwari. Geyser: A compilation framework for quantum computing with neutral atoms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 383–395, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Tirthak Patel and Devesh Tiwari. Disq: A novel quantum output state classification method on ibm quantum computers using openpulse. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Tirthak Patel and Devesh Tiwari. Veritas: Accurately estimating the correct output on noisy intermediate-scale quantum computers. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, Nov 2020.

[58] Tirthak Patel and Devesh Tiwari. Qraft: Reverse your quantum circuit and know the correct program output. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 443–455, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Tirthak Patel, Ed Younis, Costin Iancu, Wibe de Jong, and Devesh Tiwari. Quest: Systematically approximating quantum circuits for higher output fidelity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 514–528, New York, NY, USA, 2022. Association for Computing Machinery.

[60] Tianyi Peng, Aram W. Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Phys. Rev. Lett.*, 125:150504, Oct 2020.

[61] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5(1):4213, 2014.

[62] Christophe Piveteau and David Sutter. Circuit knitting with classical communication. *arXiv preprint arXiv:2205.00016*, 2022.

[63] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

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

[65] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. Mqt bench: Benchmarking software and design automation tools for quantum computing. *arXiv preprint arXiv:2204.13719*, 2022.

[66] Gokul Subramanian Ravi, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong. Quantum computing in the cloud: Analyzing job and machine characteristics. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 39–50, Nov 2021.

[67] Gokul Subramanian Ravi, Kaitlin N. Smith, Prakash Murali, and Frederic T. Chong. Adaptive job and resource management for the growing quantum cloud. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 301–312, Oct 2021.

[68] Salonik Resch, Anthony Gutierrez, Joon Suk Huh, Srikant Bharadwaj, Yasuko Eckert, Gabriel Loh, Mark Oskin, and Swamit Tannu. Accelerating variational quantum algorithms using circuit concurrency. *arXiv preprint arXiv:2109.01714*, 2021.

[69] Movahhed Sadeghi, Soheil Khadirsharbiyani, and Mahmut Taylan Kandemir. Quantum circuit resizing. *arXiv preprint arXiv:2301.00720*, 2022.

[70] Zain H Saleem, Teague Tomesh, Michael A Perlin, Pranav Gokhale, and Martin Suchara. Quantum divide and conquer for combinatorial optimization and distributed computing. *arXiv preprint arXiv:2107.07532*, 10, 2021.

[71] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[72] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[73] Akhil Pratap Singh, Kosuke Mitarai, Yasunari Suzuki, Kentaro Heya, Yutaka Tabuchi, Keisuke Fujii, and Yasunobu Nakamura. Experimental demonstration of a high-fidelity virtual two-qubit gate. *arXiv preprint arXiv:2307.03232*, 2023.

[74] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 113–125, New York, NY, USA, 2018. Association for Computing Machinery.

[75] Kaitlin N. Smith, Gokul Subramanian Ravi, Prakash Murali, Jonathan M. Baker, Nathan Earnest, Ali Javadi-Cabhari, and Frederic T. Chong. Timestitch: Exploiting slack to mitigate decoherence in quantum circuits. *ACM Transactions on Quantum Computing*, 4(1), oct 2022.

[76] Samuel Stein, Nathan Wiebe, Yufei Ding, Peng Bo, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. Eqc: Ensembled quantum computing for variational quantum algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 59–71, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: Using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 473–486, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Swamit Tannu, Poulami Das, Ramin Ayanzadeh, and Moinuddin Qureshi. Hammer: Boosting fidelity of noisy quantum circuits by exploiting hamming behavior of erroneous outcomes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 529–540, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Swamit S. Tannu and Moinuddin Qureshi. Ensemble of diverse mappings: Improving reliability of quantum computers by orchestrating dissimilar mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 253–265, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Swamit S. Tannu and Moinuddin K. Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 279–290, New York, NY, USA, 2019. Association for Computing Machinery.

[81] Swamit S. Tannu and Moinuddin K. Qureshi. Not all qubits are created equal: A case for variability-aware policies for nisq-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 987–999, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R Martonosi, and Frederic T Chong. Supermarq: A scalable quantum benchmark suite. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 587–603. IEEE, 2022.

[83] Vinay Tripathi, Huo Chen, Mostafa Khezri, Ka-Wa Yip, E.M. Levenson-Falk, and Daniel A. Lidar. Suppression of crosstalk in superconducting qubits using dynamical decoupling. *Phys. Rev. Appl.*, 18:024068, Aug 2022.

[84] Cenk Tüysüz, Giuseppe Clemente, Arianna Crippa, Tobias Hartung, Stefan Kühn, and Karl Jansen. Classical splitting of parametrized quantum circuits. *Quantum Machine Intelligence*, 5(2):34, 2023.

[85] Christian Ufrecht, Maniraman Periyasamy, Sebastian Rietsch, Daniel D Scherer, Axel Plinge, and Christopher Mutschler. Cutting multi-control quantum gates with zx calculus. *arXiv preprint arXiv:2302.00387*, 2023.

[86] Benjamin Weder, Johanna Barzen, Frank Leymann, and Marie Salm. Automated quantum hardware selection for quantum workflows. *Electronics*, 10(8), 2021.

[87] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to ibm qx architectures using the minimal number of swap and h operations. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Synthesizing quantum-circuit optimizers. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[89] Takahiro Yamamoto and Ryutaro Ohira. Error suppression by a virtual two-qubit gate. *arXiv preprint arXiv:2212.05493*, 2022.

[90] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. Time-optimal qubit mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 360–374, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Alexander Zlokapa and Alexandru Gheorghiu. A deep learning model for noise prediction on near-term quantum devices. *arXiv preprint arXiv:2005.10811*, 2020.

[92] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, July 2019.