Towards an Automated Debugging Approach for Fault Identification in Quantum Circuits

Anton Maidl*†, Abhoy Kole‡, Kamalika Datta†‡, Jannis Stoppe*, Rolf Drechsler†‡
* Institute for the Protection of Maritime Infrastructure, German Aerospace Center (DLR e.V.), Bremerhaven, Germany
Email: anton.maidl@dlr.de
† Institute of Computer Science, University of Bremen, Bremen, Germany
‡ Cyber-Physical Systems, DFKI GmbH, Bremen, Germany

***Abstract*—In this paper, we propose a novel method for locating and diagnosing bugs in quantum circuits. Debugging in the quantum domain is especially challenging due to the inherent inability of assessing the quantum state of a program. Moreover, explaining the root cause behind unexpected outcomes is hard due to the limited information gain provided by measurements. Our approach aims to address both of these issue: Firstly, the bug site is identified using a standard circuit slicing technique combined with an associated measurement strategy. Secondly, we provide information about the nature of the bug, generated through repeated measurements. To minimize the number of measurements, we introduce a notion of equivalence classes based on unitary operations. This allows us to partition the gate library into classes that produce indistinguishable results under certain measurements. Finally, We assess the effectiveness and measurement complexity of our method by applying it to relevant primitive gate components and well-known quantum algorithms. Our empirical results shows that in 95.79% of all cases, our approach reveals the correct location of the bug along with a valid set of fault candidates. Furthermore, we demonstrate that the required number of circuit executions scales logarithmically with the circuit depth or linearly with the number of qubits.**

***Index Terms*—Quantum Circuits, Quantum Debugging, Quantum Measurement**

I. Introduction

Quantum computing promises a fundamental shift in the way we solve problems computationally. Computations of this kind are expected to have great potential in a variety of fields, such as physics [1], chemistry [2] and finance [3]. More specifically, it may trigger a paradigm shift in conventional computing, such as in cryptography due to the threat posed by Shor's factoring algorithm [4], database search using Grover's algorithm [5], and finding solutions to optimization problems [6].

Currently, quantum programs are still severely limited by the available hardware. However, the number of available quantum hardware resources is constantly increasing, making it possible to execute ever larger quantum algorithms. With the growth of quantum algorithms, the complexity of the associated debugging processes also increases and specialized tools become crucial. The problems are mainly caused by the limited observability of the quantum state, as measurements only provide partial information about the internal quantum state. In addition, the results of simulating quantum algorithms are non-deterministic, i.e. either a statistical analysis is necessary or the circuit must be adapted. This often makes it more complicated for a developer to isolate and eliminate bugs.

There has been considerable work in the direction of quantum debugging in the past, including statistical analysis of measurement outcomes [7] and runtime assertions based on projective measurements [8]. While these approaches can detect faulty segments, they don't necessarily resolve the issue. A developer must still locate and eliminate the bug causing the error. This paper proposes a debugging framework that identifies the bug location using circuit slicing and explains the specific error type. We also introduce equivalence classes based on unitary operations to group gates into identical classes for a given measurement. Experiments were conducted on a set of quantum algorithms to evaluate the measurement complexity and effectiveness of our proposed method. Results reveal that our approach achieves an average accuracy of 95.79% for the considered benchmark.

The rest of the paper is organized as follows: In Section II, the necessary context about quantum computing and related works on debugging quantum circuits are presented. In Section III, the proposed measurement-based debugging approach is illustrated in detail. This includes the localization of bugs and the process of generating corresponding explanations. In Section IV, the implemented debugging technique is demonstrated on a set of well-known quantum algorithms and the obtained results are discussed. Finally, Section V concludes the paper outlining future research in this domain.

II. Background

In this section we briefly introduce the preliminaries necessary to make the paper self-contained.

A. Quantum Computing

In quantum computing, the basic unit of information is known as a quantum bit or qubit. A qubit can exist in the basis states 0 or 1, which are represented in Dirac notation as $|0\rangle$ or $|1\rangle$. In general, the state of a qubit is written as the superposition of the basis states: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Here, α and β are complex coefficients, or amplitudes, that state how the qubit is related to the individual basis states. These coefficients must satisfy the normalization criterion $|\alpha|^2 + |\beta|^2 = 1$.

For a gate-based computing model, such qubit's state is often represented as a vector in a 2-dimensional Hilbert space, e.g.

This research has been partly supported by the Federal Ministry of Education and Research (BMBF) within the project EASEPROFIT under grant no. 16KIS2127.

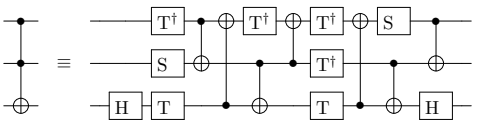

Fig. 1. An example quantum circuit realizing 3-qubit Toffoli operation using Clifford+T gates.

$|\phi\rangle = [\alpha, \beta]^\top$. Computation is carried out by manipulating states, applying quantum gates which are also referred to as unitary operations. Such gates can be represented as unitary matrices, i.e. an n-qubit unitary operator is designated by a $2^n \times 2^n$ unitary matrix. Some of the well known quantum gates and their corresponding unitary matrices are shown below:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & i \\ -i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$
$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Quantum circuits are designed as a cascade of quantum gates. For circuit description, gates from specific functionally complete sets are often used, i.e. universal gate libraries. One such gate library is Clifford+T [9], which consists of H, T, $T^\dagger$ and *CNOT* gates. The use of this library in practice is motivated due to its fault-tolerant features. Fig. 1 shows the Clifford+T realization of a Toffoli gate.

B. Related Works

Prior to this work, there has already been considerable effort in the development of quantum debugging methods and tools. Most notably, different kinds of assertions have been considered, based on statistical methods [7] and projective measurements [8].

For the statistical method, classical observations obtained from the quantum program are analyzed using a χ^2 test. This approach can detect whether specific properties hold at predefined breakpoints, such as whether an intermediate state is in superposition, represents classical values, or is entangled. While such assertions can provide some insight into the faulty behavior of the system, they only offer partial information about how the bug actually occurs and do not help to isolate the bug's exact location. Additionally, a large number of measurements are necessary, which can impact the efficiency of the debugging process and prevents debugging at runtime.

AUTOQ [10] is a formal system which promises to detect bugs in quantum programs. At the core of this system is a compact algebraic representation of quantum states in the form of tree automata. This kind of representation exploits redundancies in the complex values representing state vectors [11]. A quantum program is then implemented by a set of transformers which operate on such tree representations. AUTOQ can then be used to check for bugs by performing an equivalence check on the buggy circuit and the specified output. As such, the method can reliably detect when an error is present but does not provide information about its location or root cause.

There also exist several other works which aim to simplify the handling of quantum circuits during the debugging process. Most notably, the Cirquo tool [12] provides simple routines to extract both vertical and horizontal slices from quantum circuits. This enables the fast debugging of subroutines of a quantum algorithm. Another work [13] provides an algorithm which is able to quickly locate the position of a bug, based on an enhanced binary search on such circuit slices. Here, a cost-aware approach is selected to determine the next circuit cut, together with relaxed requirements for the statistical accuracy of intermediate outcomes. These approaches provide a systematic way to locate the faulty segment, but they lack a strategy to interpret the faulty classical results and do not explain the underlying cause of the bug.

While some of the reported approaches can successfully detect and locate bugs, others, like AUTOQ, only indicate the presence of an error without specifying its location. However, none of these methods provide detailed insights into the underlying cause of the bug. In contrast, our approach achieves all three objectives by not only detecting bugs but also locating them and providing insights into their root cause. In doing so, we perform the following:

1) Divide the gate library into equivalence classes based on the expected outcome of projective measurements.
2) Utilize a combination of statistical analysis and binary search to detect and locate bugs.
3) Use the equivalence classes to generate insight about the root cause of the bug by repeated projective measurement.
4) Demonstrated the debugging ability on various benchmarks constructed using the considered extended Clifford+T gate library.

III. PROPOSED DEBUGGING APPROACH

In this section, we introduce our novel debugging approach starting by defining the assumptions considered for the corresponding debugging environment. The approach mainly consists of two stages. In the first stage, the circuit is repeatedly sliced vertically to find the segment which introduces a fault into the circuit. This stage is inspired by the method in [12], but we utilize multiple different measurements to gain confidence in the correctness of intermediate outcomes. Secondly, and at the core of our approach, we detail how possible bug types can be deduced efficiently from repeated measurements.

A. Basic Assumptions

Throughout the debugging flow, we assume the existence of a `eq(`m_1`, `m_2`)` method. Given two (possibly noisy) sets of measurements m_1 and m_2, the `eq` method determines whether the underlying probability distribution is identical. As shown in [7], this can be realized, for example, by statistical tests. Our debugging model assumes the presence of exactly one bug within the circuit. However, it should be possible to use our approach to diagnose multiple bugs by not terminating the binary search procedure after a bug is found in the circuit. In addition, we assume that the developer is able to arbitrarily add and remove (possibly unknown) gates from the circuit to observe intermediate measurement outcomes.

We assume that a bug manifests as an additional gate within the circuit. This additional gate can be positioned arbitrarily

TABLE I
EQUIVALENCE CLASSES FOR GATE LIBRARY G AND MEASUREMENTS ALONG THE THE z-, y- AND x-AXIS.

z	$[I, Z, S, S^\dagger, T, T^\dagger]$, $[X]$, $[H]$
y	$[I]$, $[H, X, Z]$, $[S]$, $[S^\dagger]$, $[T]$, $[T^\dagger]$
x	$[I, X]$, $[Z]$, $[S]$, $[S^\dagger]$, $[T]$, $[T^\dagger]$, $[H]$

in the circuit such that it does not behave equivalently to the identity operator. Gate deletions are not modeled explicitly as our library is constructed in a way such that deletions can be thought of as the addition of the corresponding inverse gate. With these assumptions, we primarily target bugs which are introduced either by human developer or by faulty methods during the transpilation of a circuit. As such, adaptions are necessary to apply our approach when considering bugs that occur due to noise on a device level. Throughout the following definitions and the experimental evaluation, we consider the gate library

$$G = \{H, X, Z, S, S^\dagger, T, T^\dagger, CZ\}. \tag{1}$$

Nonetheless, to the best of our abilities, we provide general definitions for the utilized concepts such that they can be used for any arbitrary universal gate library. The library represents a slightly adapted version of the universal Clifford+T set. The significant difference lies in the design choice to employ the controlled-Z operator instead of the conventional controlled-X. However, the CZ operator possesses some properties that promise to simplify the debugging process. Mainly, we selected CZ due to its inherent symmetry and decide to decompose any CX operator using CZ together with the Hadamard transformation.

B. Equivalence Classes

To reduce the amount of necessary tests for each gate candidate, we first introduce the concept of gate equivalency. Given two unitary operators U_1, U_2 and a projective measurement operator M, we want to decide whether it is possible to distinguish U_1 and U_2 when M is applied right after their application to the circuit. Two unitary operators are not distinguishable by a projective measurement operator if and only if the probability of observing the associated eigenstate of M is equal for any given state, i.e.:

$$\begin{aligned}\forall |\varphi\rangle \in \mathcal{H}\colon\ & \langle\varphi| U_1^\dagger M U_1 |\varphi\rangle - \langle\varphi| U_2^\dagger M U_2 |\varphi\rangle \\ &= \langle\varphi| \left(U_1^\dagger M U_1 - U_2^\dagger M U_2\right) |\varphi\rangle \overset{!}{=} 0. \end{aligned} \tag{2}$$

If $U \in \mathcal{C}^{n\times n}$ is a skew-hermitian matrix, i.e. $U^\dagger = -U$, it is a well-known fact [14] that

$$U \in \mathcal{C}^{n\times n} \text{ is skew-hermitian } \iff \forall x \in \mathcal{C}^n\colon x^\dagger U x = 0.$$

Using this fact, we define the equivalence $\equiv_M$ in relation to a projective measurement operator M as

$$U_1 \equiv_M U_2 \iff U_1^\dagger M U_1 - U_2^\dagger M U_2 \text{ is skew-hermitian.}$$

For our gate library G and measurements along the z-, y- and x-axis, we use $\equiv_M$ to partition G into equivalence classes as shown in Table I. By utilizing this table, the number of required measurements can be reduced to identify a buggy gate. This is achievable because, for instance, repeated probing with I, Z, S, $S^\dagger$, T, and $T^\dagger$ gates does not yield additional information through measurement in the z-basis and can thus be omitted.

This concept can be extended for arbitrary multi-qubit operators by comparing the higher dimensional matrix representations of these operators. For our experiments, we selected CZ as our only multi-qubit operator because no separation between control and target is present due to its symmetry. Assuming that a bug is present in the form of CZ, either 2, 1 or no qubit will have undesired outputs. If both the qubits are buggy, we know that CZ must have been the culprit as it is the only multi-qubit gate. In case all of the outcomes are as expected then the CZ does not count as a bug. When only a single qubit is buggy, we consider a simplified case in which the effect of CZ on that single qubit is not distinguishable from a single qubit Z operation.

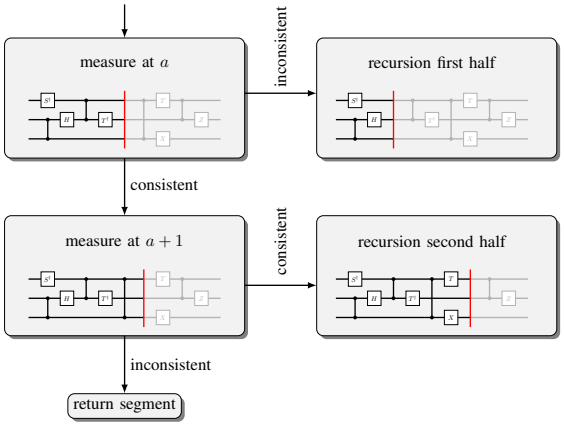

Fig. 2. Overview over the binary search strategy for a quantum circuit, starting by measuring some index a. The red line indicates the current cut-off point, meaning that the greyed out circuit elements are excluded from execution.

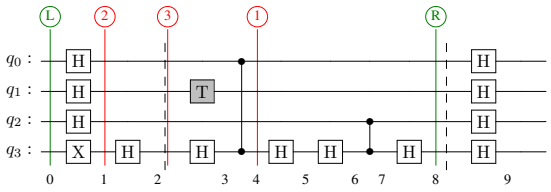

Fig. 3. Bernstein-Vazirani algorithm [15] with the secret string $s = 101$. An additional T gate is present, shaded in gray. Values below circuit indicate the slice index. Vertical red lines indicate the order of slicing using the binary search strategy. The standard strategy is used to determine the next cut: Two initial limits L and R are determined (here: $L = 0$, $R = 8$) and the cut is performed at $a = \lfloor \frac{L+R}{2} \rfloor$. Depending on the result, $L = a+1$ or $R = a-1$ is then set.

C. Segmentation

After the equivalence classes have been identified, the next step is to find the bug location through circuit slicing. To identify the vertical position of the bug in the quantum circuit, a standard binary search algorithm is employed. This means that the quantum circuit is repeatedly cut in half to locate the bug site. Given a circuit which is cut at index a, we then verify if the bug occurred at that position. In order to do so, it is first verified whether the measurement outcomes at index a

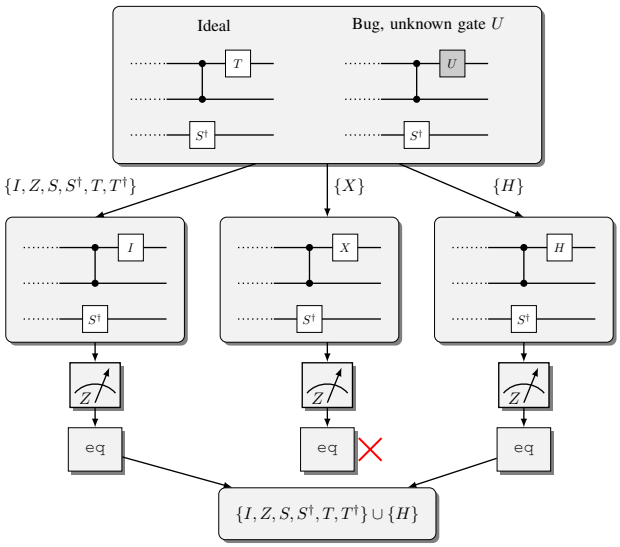

Fig. 4. Application of the `getCandidates` procedure using only the z-basis. Starting from the ideal circuit, a new circuit is constructed for each of the equivalence classes of basis z to deduce the unknown gate U. After executing and comparing the measurement results, the union of all equivalence classes is taken for which the measurements did match. For the specific example, we assume that the circuit representing $\{X\}$ did not match expectation.

are consistent with the belief about the correct values or not. If the measurement is already inconsistent at slice a, the bug must have occurred before and the binary search routine is recursively called on the partition before index a.

If the outcomes are consistent with our expectation at a, we can conclude that the bug must occur in the second half of the circuit. However, before applying the search recursively to the posterior partition, we first check if a itself is the bug site. To verify if the bug occurred at slice a, slice $a+1$ is also measured. Assuming that slice a is still consistent with the expectation, but slice $a+1$ is no longer, the bug must occur exactly in the layer between a and $a+1$. If $a+1$ is also consistent, the routine is applied to the second part of the circuit partition. A graphical overview of this strategy is given in Fig. 2.

In order to check a slice for consistency, the quantum state is measured with a set of three different measurement bases. From the Bloch sphere representation of each individual qubit, it is clear that three orthogonal measurements suffice to distinguish different points on the sphere. For our purposes, we utilize measurements in the x- and y-basis, together with the computational basis. An example for the application of this method can be seen in Fig. 3.

D. Generation of Candidate Sets

In the next step, we introduce the `getCandidates` procedure. This method is the central component of our approach, as it is used to recognize possible causes of errors and inform the developer about them. Given is an ideal circuit and a faulty circuit which are both segmented up to the fault location. For some qubit index k, the `getCandidates` procedure now aims to generate a set of gates which could be present at the fault location at qubit k. These types of sets are made available to the developer to detect possible errors. Furthermore, the `getCandidates` method is used to determine the faulty qubit at the fault-location.

A naive approach of generating such sets would be to repeatedly insert each gate in the given library into the fault location and check if the measurement results are consistent with the observed faulty results. However, as seen in subsection III-B, this is not necessary since gates can be grouped in equivalence classes which are indistinguishable under a given measurement. Instead, for each available basis, a circuit is prepared for each equivalence class.

The partial flow of the procedure is exemplified for the z-basis in Fig. 4 using the classes from Table I. Given the equivalence classes for some fixed basis, a set of circuits is prepared by inserting an arbitrary representative from each class into the ideal circuit at index k. Afterwards, each of the circuits is measured and the resulting probability distribution is compared to the observed faulty distribution using the `eq` procedure. The basis-specific candidate set is then constructed as the union of all equivalence classes whose measurement result matched those of the buggy circuit.

For the `getCandidate` procedure, this principle is repeated for each of the available bases with their respective equivalence classes, in our case x, y and z. The unknown gate at qubit k must be consistent with all base-specific sets, meaning that for each base it is part of the corresponding set. Accordingly, the `getCandidates` procedure returns the intersection of all basis-specific candidates.

E. Algorithm

The overall workflow of our debugging scheme is the composition of the binary search algorithm and the repeated application of the `getCandidate` procedure. In the first stage, the binary search strategy is employed to find the segment in which the bug occurs. Afterwards, the `getCandidate` procedure is applied to find the faulty qubit and to generate a candidate set which would explain the observed behaviour.

In order to find this faulty qubit, the procedure is applied to each qubit individually. After generating the candidate set for each qubit, the faulty qubit can be identified by comparing its candidate set with the expected gate in the ideal circuit. If the expected gate is in the set of candidate gates, the qubit's operation is indistinguishable from the intended behaviour, ruling it out as faulty. On the other hand, if the desired gate is not part of the candidate set, it can be concluded that the operation of the qubit yields a result that does not correspond to the expectation and that the bug is located at this qubit.

The runtime of the complete algorithm is composed from the binary search and the application of `getCandidates` to each qubit. The runtime for the binary search grows logarithmically with the depth d of the quantum circuit. The number of circuit executions required to execute the `getCandidates` method depends on the number of overall equivalence classes, which in turn is bounded by the size of the gate library. Since we assume that the library has a constant size for different circuits, the method itself can be executed in constant time. By applying `getCandidates` to each of the n qubits, an overall runtime of $\mathcal{O}(n + \log d)$ is given.

IV. Experimental Results

We considered three algorithms that include Bernstein-Vazirani's, Grover's and Shor's algorithm for evaluation. For Bernstein-Vazirani's algorithm, random secret words of different lengths were generated. In the case of Grover's algorithm, an oracle was generated for randomly selected SAT instances. Shor's algorithm was considered with the input combinations $(15, 2)$, $(21, 4)$ and $(35, 4)$. Further, individual components of these algorithms, i.e. Grover's oracle and diffusion operator, Shor's modular exponentiation operator and 2-qubit QFT were also considered.

To assess the efficiency and effectiveness of our approach, a total of 25 faulty versions were generated for each combination of gate and circuit, yielding $|G| \cdot 25 = 200$ experiments per circuit. For each experiment, the current gate is injected at a randomly selected segment and qubit. We did not allow for the replacement or deletion of existing gates. However, a new gate can be placed at any location in the circuit, including the addition of gates between two previously neighbouring gates. For each execution, the number of measurements was reported, in addition to the output of the method itself. All experiments were conducted on a quad core 1.80 GHz Intel Core i7 processor with 8GB RAM.

A. Effectiveness

Table II was constructed from the observed values during our experimentation. Overall, our debugging scheme is able to find the correct location and a correct set of candidate gates in 95.79% of all considered buggy circuits. On closer examination of the values, however, it becomes clear that the current treatment of the CZ gate does not work in all cases, prompting the need for further refinement of the strategy for CZ. Nevertheless, overall we assume a high accuracy of our approach, especially given the almost 100% success rate of the single qubit gates.

Another advantage of our approach compared to AUTOQ is also demonstrated in Table II. Although AUTOQ was able to indicate in 100% of the experiments when an error was present, the procedure is unable to determine the specific location or nature of the bug, only indicating that an issue existed somewhere in the circuit. In most cases, we are not only able to decide the presence of a bug, but also construct a set of gates that indicate possible erroneous additions of gates from the library, aiming to provide the developer information about possible sources of the faulty behaviour.

B. Efficiency

We plot the number of required circuit executions for each circuit in our benchmark in Fig. 5. Overall, the number of measurements appear to follow a logarithmic trend. As the runtime scales with $\mathcal{O}(n + \log d)$, this matches our expectation, since $n \ll d$ for most of our benchmarks. Additionally, for all the individual components of our algorithms, we give a comparison to the execution time of AUTOQ in Table III. For each of the primitives, we injected a randomly selected bug for each gate into the circuit and applied the AUTOQ implementation to it. We report the execution time in milliseconds and provide the average number of circuit executions for our approach, generated by the previous experiment.

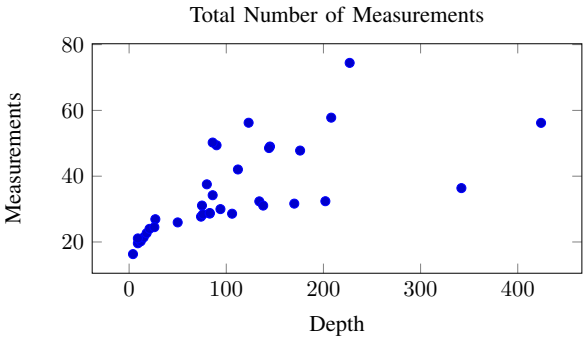

Fig. 5. Total number of measurements required versus circuit depth for our benchmarks.

Although AUTOQ is able to check comparatively large benchmarks, we observe that the scalability for actual full-scale quantum algorithms is unclear. In addition to that, we observed that the runtime of AUTOQ appears to increase exponentially for some specific benchmarks, such as Grover's diffusion operator. While a direct comparison in runtime is hard, the values for our approach do not seem to indicate an exponential increase in the number of required circuit execution. This is because we try to utilise quantum resources to perform non-trivial tasks, while AUTOQ relies solely on the utilisation of classical resources.

V. Conclusion

In this paper we propose a debugging approach to locate and diagnose bugs in quantum circuits. In order to do so, we utilize a binary search approach and we introduce the notion of equivalence classes for indistinguishable gates. Our experiments reveal that our method performs well in terms of both accuracy and efficiency. In terms of accuracy, we were able to detect the location of a bug together with a valid set of possible bugs in 95.79% of all cases. In terms of effectiveness, we showed that our tool is not only able to find the presence of a bug, but also the exact location and type in most of the cases. For efficiency, we provide evidence that our tool may be able to be applied to particularly large circuits, given a quantum computer on which to run them. In future work, we plan to further refine the binary search strategy. While this strategy may be used to locate faulty segments in a number of circuit executions that scale logarithmically with circuit depth, it depends on reliably determining whether the measurement results match expectation, which is challenging in noisy circuits. We plan to explore methods, such as statistical tests, to achieve this task given a specific gate library and noise model. Furthermore, we also want to investigate the possibility of using the circuit slicing approach to handle multiple bugs.

References

[1] R. P. Feynman, "Simulating physics with computers," in *Feynman and computation*. CRC Press, 2018, pp. 133–153.

[2] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon, "Simulated quantum computation of molecular energies," *Science*, vol. 309, no. 5741, pp. 1704–1707, 2005.

TABLE II

ACCURACY FOR THE CONSIDERED BENCHMARKS. THE NUMBER OF QUBITS, CIRCUIT DEPTH AND NUMBER OF GATES IS GIVEN BY n, d AND g RESPECTIVELY. FOR EACH GATE IN OUR LIBRARY WE REPORT THE ACCURACY OF DETECTION DURING THE EXPERIMENT, WHERE A DETECTION INCLUDES THE CORRECT LOCATION AND A VALID SET OF CANDIDATE GATES. THE LAST COLUMN INDICATES THE AVERAGE SUCCESS RATE FOR ALL RANDOMLY INJECTED BUGS.

Benchmark	n	d	g	H	S	$S^\dagger$	T	$T^\dagger$	X	Z	CZ	Ø
BV5	5	9	17	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
BV7	7	12	24	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
BV10	10	15	33	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
BV12	12	18	40	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
GA5	5	86	118	1.000	1.000	1.000	1.000	1.000	0.960	1.000	0.640	0.950
GA8	8	145	214	1.000	0.960	0.960	1.000	1.000	1.000	0.960	0.480	0.920
GA9	9	123	188	1.000	0.960	0.960	1.000	1.000	1.000	0.960	0.200	0.885
GA12	12	424	676	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.480	0.935
Shor_15_2	6	90	119	1.000	1.000	1.000	0.960	0.960	1.000	0.920	0.280	0.890
Shor_21_4	7	86	113	1.000	1.000	0.920	1.000	0.960	0.960	0.960	0.240	0.880
Shor_35_4	8	227	326	1.000	0.920	0.960	1.000	1.000	0.960	0.840	0.320	0.875
qft2	2	9	11	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
GA5_ORACLE	5	75	99	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.880	0.985
GA8_ORACLE	8	134	195	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.920	0.990
GA9_ORACLE	9	94	139	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.920	0.990
GA12_ORACLE	12	342	537	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
DIFF3	3	27	41	1.000	0.960	1.000	1.000	1.000	0.880	0.960	0.560	0.920
DIFF4	6	80	129	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.600	0.950
DIFF5	8	112	187	1.000	0.960	1.000	1.000	1.000	0.960	1.000	0.760	0.960
DIFF6	10	144	245	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.720	0.965
DIFF7	12	176	303	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.640	0.955
DIFF8	14	208	361	1.000	1.000	1.000	1.000	1.000	0.960	0.960	0.800	0.965
MCT3	3	21	27	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.360	0.920
MCT4	6	74	111	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.720	0.965
MCT5	8	106	165	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.800	0.975
MCT6	10	138	219	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.760	0.970
MCT7	12	170	273	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.760	0.970
MCT8	14	202	327	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.800	0.975
CU_k15_a2_0	3	4	6	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
CU_k15_a2_1	5	76	99	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.640	0.955
CU_k21_a4_0	3	26	33	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.480	0.935
CU_k21_a4_1	4	50	66	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.840	0.980
CU_k35_a4_0	5	83	117	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.720	0.965
CU_k35_a4_1	5	83	117	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.560	0.945

TABLE III

COMPARISON OF THE EFFICIENCY OF AUTOQ AND OUR METHOD. FOR AUTOQ WE INDICATE THE REPORTED RUNTIME IN MILLISECONDS. FOR OUR APPROACH, WE INDICATE THE AVERAGE NUMBER OF REPORTED CIRCUIT EXECUTIONS.

Benchmark	AUTOQ [10] [ms]	This Paper [#meas.]
qft2	19	21.085
GA5_ORACLE	104	31.075
GA8_ORACLE	549	32.350
GA9_ORACLE	281	29.990
GA12_ORACLE	1074	36.375
DIFF3	45	26.920
DIFF4	214	37.525
DIFF5	693	42.040
DIFF6	1197	48.550
DIFF7	2801	47.790
DIFF8	5456	57.795
MCT3	38	23.990
MCT4	109	27.700
MCT5	338	28.595
MCT6	423	31.065
MCT7	600	31.660
MCT8	758	32.390
CU_k15_a2_0	118	16.310
CU_k15_a2_1	75	28.425
CU_k21_a4_0	41	24.505
CU_k21_a4_1	73	25.950
CU_k35_a4_0	143	28.625
CU_k35_a4_1	255	28.805

[3] R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: Overview and prospects," *Reviews in Physics*, vol. 4, p. 100028, 2019.

[4] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 1994, pp. 124–134.

[5] L. Grover, "A fast quantum mechanical algorithm for database search," in *ACM Symp. Theory Comput.*, Jul 1996, pp. 212–219.

[6] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn *et al.*, "Quantum optimization using variational algorithms on near-term quantum devices," *Quantum Science and Technology*, vol. 3, no. 3, p. 030503, 2018.

[7] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 541–553.

[8] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020.

[9] M. Nielsen and I. Chuang, *Quantum computation and quantum information*. Cambridge Univ. Press, Oct 2000.

[10] 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," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1218–1243, 2023.

[11] A. Zulehner and R. Wille, "Advanced simulation of quantum computations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 848–859, 2018.

[12] S. A. Metwalli and R. V. Meter, "Cirquo: A suite for testing and debugging quantum programs," 2023.

[13] N. Sato and R. Katsube, "Locating buggy segments in quantum program debugging," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 26–31.

[14] R. A. Horn and C. R. Johnson, "Matrix Analysis," 1985.

[15] E. Bernstein and U. Vazirani, "Quantum complexity theory," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1411–1473, 1997.