QDB: From Quantum Algorithms Towards Correct Quantum Programs

Yipeng Huang[1]
Princeton University, USA
yipeng@cs.princeton.edu
https://orcid.org/0000-0003-3171-6901

Margaret Martonosi
Princeton University, USA
mrm@princeton.edu

Abstract

With the advent of small-scale prototype quantum computers, researchers can now code and run quantum algorithms that were previously proposed but not fully implemented. In support of this growing interest in quantum computing experimentation, programmers need new tools and techniques to write and debug QC code. In this work, we implement a range of QC algorithms and programs in order to discover what types of bugs occur and what defenses against those bugs are possible in QC programs. We conduct our study by running small-sized QC programs in QC simulators in order to replicate published results in QC implementations. Where possible, we cross-validate results from programs written in different QC languages for the same problems and inputs. Drawing on this experience, we provide a taxonomy for QC bugs, and we propose QC language features that would aid in writing correct code.

2012 ACM Subject Classification Computer systems organization → Quantum computing

Keywords and phrases correctness, debugging

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.4

1 Introduction

Quantum computing is reaching an inflection point. After years of work on both QC algorithms and low-level QC devices, small but viable QC prototypes are now available to run programs. These QC prototypes are increasing in size, with much research attention being placed on improving their reliability and increasing the counts of qubits (quantum bits), the fundamental building block for QC [11, 19, 31].

With small-scale machines available to run real code, a natural challenge lies in creating correct and useful programs to run on them [3, 12]. Until recently, QC algorithms were rarely programmed for actual execution, and therefore relatively little QC debugging has ever occurred. Furthermore, QC debugging faces challenges beyond that of classical computing. In particular, typical debugging approaches based on printing out variable values during program execution do not easily apply to QC programs, because program states in QC "collapse" to classical values when observed. Second, QC's "no cloning rule" precludes us from making a spare copy of variables to observe them elsewhere. Third, while we have more freedom to observe states in QC simulations on classical computers, the massive state spaces of QC executions limits this approach to small programs. Finally, even when limited simulations are tractable, it can be difficult to interpret the simulation results.

[1] This work is funded in part by EPiQC, an NSF Expedition in Computing, under grant 1730082

This paper surveys a range of QC algorithms and programs and offers a set of empirical and experiential insights on today's state-of-the-art in QC debugging. For three benchmarks representing different application areas, we perform detailed debugging based on small-scale simulations. For each, we give case studies of the types of bugs we found. Most importantly, we use these experiences to assemble a set of "design patterns for QC programming" and related best practices in QC debugging.

In particular, the contributions of this paper are as follows:

- We specifically explore three major areas: quantum chemistry, integer factorization, and database search. This is a broad spectrum of QC algorithms across not just application domains, but also problem size and algorithm strategies. This allows us to point out particular domain-specific challenges or opportunities.
- Where available, we study the same algorithm implemented in different languages or infrastructures. From this, we draw comparative insights regarding how programming language or environment support can be useful in QC programming and debugging.
- From these insights and experiences, we lay out a plan for debugging support in QC programming environments to aid users in creating quantum code. These include assertions, unit testing, code reuse, polymorphism, and QC-specific language types and syntax.

Overall, while QC programming has received significant prior attention and QC debugging has received some as well, our work offers steps forward in its detailed and comparative assessment across problem types and languages. We see our work offering useful insights for QC programmers themselves, as well as language and system designers interested in building next-generation compilers and debuggers.

2 Background on QC programming

First, we review the principles of quantum computing [14, 22, 23, 26], in order to understand how writing correct quantum programs is different from classical programming.

2.1 Qubits, superpositions, and entanglement

The basic unit of information in QC is the qubit, which can take on values of $|0\rangle$ and $|1\rangle$ like bits in classical computing, but can also be viewed as a probabilistic "superposition" between the two values. Quantum computers can also "measure" the value of a qubit, forcing it to collapse out of superposition into a classical value such as '0' or '1'. Measurement disturbs the values of variables in a quantum computer, so we cannot easily pause execution and observe the values of qubits as a quantum program runs.

The state of individual qubits can be "entangled" together. For this reason, as more qubits come into play in a quantum computer, the number of states that data can be in grows exponentially. For example, a two-qubit system can take on the values $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, along with superpositions among these values; furthermore, the two qubits can even be in a state of entanglement where the two cannot be treated as independent pieces of information. A three qubit system has potential superpositions of eight states, and so on. This exponential growth of possible values underlies the power of QC.

As a result of this large number of possible states, running a quantum program in simulation on a classical computer is costly. Naive simulation of a 20-qubit quantum computer, for example, needs 2^{20} or roughly one million floating point numbers just to store the program state at any instant. For this reason, testing and debugging quantum programs in simulation is only possible for toy-sized programs.

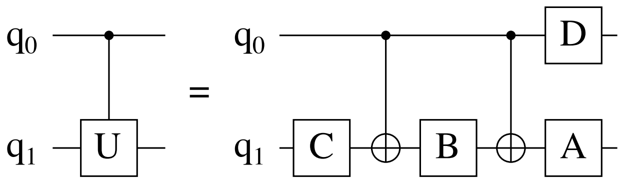

Figure 1 Decomposition of a simple QC program. Time flows left to right, showing sequences of operations applied to qubits q_0 and q_1. The left program is a "controlled" arbitrary operation U, which means whether the operation U works on q_1 is dependent on the value of q_0. The left sequence decomposes into the equivalent right sequence of more basic operations. The basic operations include single-qubit "rotations" A through D that alter the probability distribution of qubit values. The operations also include two two-qubit "CNOT" operations that flip a qubit (denoted $\oplus$) contingent on the value of another qubit (denoted $\bullet$) [26].

2.2 Quantum computer operations, programs, and a taxonomy for bugs

The process of quantum computing involves applying operations on qubits. We use diagrams such as Figure 1 to represent sequences of quantum operations. Looking at Figure 1 we see that quantum programs consist of three conceptual parts [8]:

1. **Inputs** to quantum algorithms include *classical* input parameters such as coefficients for rotations A through D, and *quantum* initial values for qubits such as q_0 and q_1.
2. **Operations**, such as the specification of how a complex operation such as controlled arbitrary operation U (Figure 1, left) decomposes into basic operations A through D and CNOTs (Figure 1, right). Additionally, both basic and complex operations can be further composed according to patterns such as iteration, recursion, and mirroring.
3. **Outputs** of quantum algorithms are the final classical measurement values of qubits such as q_0 and q_1. Furthermore, any temporary variables used in the course of a program have to be safely disentangled from the rest of the quantum state and discarded.

Bugs in quantum programs can crop up due to mistakes made in any of these three parts of a QC program. We will give examples of each kind of bug along with how to prevent them, using detailed case studies in the rest of this paper.

2.3 QC algorithm primitives, benchmarks, and open source frameworks

Given the rapid growth of QC infrastructure, we now have a chance to test a variety of quantum algorithms written in many languages [18]. Many different quantum algorithms rely on a handful of QC algorithm primitives to get speedups relative to classical algorithms [4, 24, 25]. Table 1 classifies canonical quantum algorithms according to their algorithm primitives, and cites example implementations in different QC languages and tool chains.

This paper specifically focuses on program bugs and defenses in three areas: a quantum chemistry problem that uses quantum phase estimation, integer factorization using Shor's order finding algorithm, and Grover's database search algorithm.

Using programs written in the Scaffold language as a starting point [13], we compile Scaffold code to OpenQASM, a QC assembly language [5]. Then, we simulate the programs operation-by-operation in the QX simulator [15], in order to see their intermediate states and outputs. We cross reference the programs' results against implementations in other languages, such as LIQUi|> [32], ProjectQ [10, 36] and Q# [37]. From this debugging experience we identify possible bugs and defenses. Furthermore, we review the codes across languages to understand the relative merits of different QC language features.

Table 1 Quantum algorithm primitives and open source benchmarks in open source tool chains.

Primitives	Quantum algorithms	Benchmark implementations
Entanglement protocols	superdense coding / quantum teleportation	Q# teleportation [37] pyQuil teleportation [35]
Quantum (random) walks	tree traversal	Scaffold / Quipper binary welded tree [6, 13, 39]
	graph traversal	Scaffold / Quipper triangle finding problem [6, 13, 39]
	satisfiability	Scaffold / Quipper Boolean formula [6, 13, 39]
Adiabatic	Ising spin model	Scaffold / Q# adiabatic Ising model [13, 37]
	quantum approximate optimization algorithm	QISKit Aqua QAOA pyQuil QAOA ansatz [35]
Variational Quantum Eigensolver	Hamiltonian simulation	QISKit Aqua quantum chemistry Q# H_2 simulation [37] Rigetti Grove VQE [35]
Quantum Fourier Transform (QFT)	phase estimation	Scaffold / Quipper ground state estimation [6, 13, 39]
	period finding	Scaffold class number [13]
	order finding	Scaffold / ProjectQ / Q# Shor's factoring [13, 36, 37]
	hidden subgroup problem	Quipper unique shortest vector [6, 39]
	linear algebra	Quipper quantum linear systems [6, 39]
Amplitude amplification	database search	Scaffold square root [13] ProjectQ / Q# Grover's database search [36, 37]

3 Case study: Quantum chemistry

First, we discuss our experience building up and debugging a simple quantum chemistry program. Quantum chemistry problems entail finding properties of molecules from theoretical first principles [20, 27]. Researchers anticipate these will be the first applications for QC due to the relatively few number of qubits they need to surpass classical computer algorithms. Debugging these problems is distinctively challenging, due to the importance of getting a large number of classical input parameters all correct, and because of the dearth of physically meaningful intermediate states we can check in the course of algorithm execution.

3.1 Bug type 1: Incorrect classical input parameters

A key part of quantum chemistry programs is in correctly building up a "Hamiltonian" subroutine that simulates inter-electron forces. The procedure for doing this was laid out in detail by Whitfield [41]. We followed this procedure to create a subroutine for simulating the hydrogen molecule, but we needed additional validation from several other sources to get a bug-free subroutine [40]. These resources include raw chemistry data found in open source repositories for the LIQUi|> framework[2]. The final parameters for actual operations on qubits were validated against a follow-up paper [33] and an implementation in the QISKit framework[3]. Because the procedure for preparing these quantum chemistry models involves many steps and needs domain expertise, software packages such as OpenFermion now automate this process [21]. Nonetheless, there is room for improvement in standardizing input data formats to eliminate bugs in this process.

[2] `https://github.com/StationQ/Liquid/blob/master/Samples/h2_sto3g_4.dat`
[3] `https://github.com/Qiskit/aqua/blob/master/test/H2-0.735.json`

Table 2 QC calculated energy for H_2 (bond length = 73.48 pm) for different electron assignments.

	Electron assignments				**QC calculated energy (relative)**
	Bonding		Antibonding		
	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	
3rd excited state (E3)	0	0	1	1	-0.164
2nd excited state (E2)	0	1	1	0	-0.217
	1	0	0	1	
1st excited state (E1)	0	1	0	1	-0.244
	1	0	1	0	
Ground state (G)	1	1	0	0	-0.295

Once the Hamiltonian subroutine is built, we can use the model in a variety of quantum algorithms spanning different primitives in Table 1. These include phase estimation (an application of quantum Fourier transforms) [28], variational quantum eigensolvers [30], and adiabatic algorithms [1]. In this paper, we use iterative phase estimation to find the ground state energy of our H_2 model, validating results published by Lanyon [17].

3.2 Bug type 2: Incorrect quantum initial values

The correct preparation of qubit initial values is important. Incorrect initial values would cause the program to find solutions to different problems altogether. In this quantum chemistry problem, the initial values control the locations of the two electrons in H_2. As shown in Table 2, we need the qubit assignment for finding the ground energy of H_2, while other assignments lead to results for other energy levels.

The symmetry of H_2 allows us to perform a sanity check, to make sure the Hamiltonian and the iterative phase estimation subroutines are working correctly. Though there are six ways to assign two electrons to four locations, there are in fact only four distinct energy levels, as shown in the experimental data. Checking that the two different ways to obtain E1 (and E2) give the same energy levels validates that the model correctly preserves symmetry.

3.3 Defense type 1: Assertions on algorithm preconditions

Given how important correct initial values are for all quantum algorithms, it is worthwhile to explicitly check for these algorithm preconditions before continuing with execution or simulation. What the preconditions should be depends on the type of algorithm. For example, the phase estimation subroutine in this case study (along with other algorithms relying on quantum Fourier transforms), expect inputs that are maximally in superposition among all possible values. Likewise, "ancillary qubits" such as the inputs to the Hamiltonian subroutine take on completely classical (integer) initial values. Lastly, quantum protocols often need to start with entangled states. These required input states are among the few places in quantum algorithms where we can check states for specific values. We can check these preconditions by running or simulating programs up to the entry point of subroutines, and performing a premature measurement to check for these anticipated states, finally restarting the program knowing that execution is correct up to that point. Thus far, the Q# framework has the most extensive support for precondition checking [37].

Table 3 Shor's factorization algorithm subroutines [23, p. 25].

Program subroutine code	Shared library code
Shor's routine for factoring 15; calculating powers of a number - controlled modular multiplication - controlled modular addition - controlled addition	- quantum Fourier transform - controlled controlled rotation - controlled rotation - controlled swap - swap

3.4 Defense type 2: Assertions on algorithm progress

Unlike the other two case studies later in this paper, the debugging process for the quantum chemistry benchmark is coarse-grained. That is because the Hamiltonian subroutine is a monolithic block of code whose components do not have obvious expected outputs—its components represent pair-wise electron interactions, and do not have inherent physical meaning. So how do we debug this program? The preconditions in the last section make sure the inputs to the algorithm are correct; the other observable state we have for debugging is to check the behavior of the algorithm as a whole.

In this quantum chemistry program, we can check for two types of overall algorithm behavior. One is the solution should converge to a steady value as finer Trotter time steps (a kind of numerical approximation) are chosen; a lack of this type of convergence indicates a bug in the Hamiltonian subroutine. The other algorithm behavior is when we vary the precision of the phase estimation algorithm, the most significant bits of the measurement output sequences should be the same—in other words, rounding the output of a high-precision experiment should yield the same output as a lower-precision experiment. a lack of this convergence indicates a bug in the iterative phase estimation subroutine. These checks for expected algorithm progress also apply to other algorithms.

4 Case study: Shor's algorithm for integer factorization

While our debugging strategy for quantum chemistry had to be coarse-grained, the debugging process for Shor's algorithm in this section allows us to look inside the program one subroutine at a time, where we can compare the intermediate results against known expected values.

Shor's factorization algorithm uses a quantum computer to factor a composite number in polynomial time complexity, providing exponential speedup relative to the best known classical algorithms [34]. We follow an example for an implementation that minimizes the qubit cost [2], and replicate results for factoring 15, the simplest example [16] [26, p. 235].

4.1 Bug type 3: Incorrect operations and transformations

In order to correctly implement Shor's algorithm we first have to build up the quantum subroutines shown in Table 3. These basic subroutines can be tricky to get right. Take the controlled rotation in Figure 1 as an example: Table 4 shows multiple ways to code the decomposition of the controlled rotation, and small mistakes can lead to incorrect behavior.

Table 4 Correct and incorrect code for rotation decomposition. Using the Scaffold language [13] as an example, we code out the controlled operation U in Figure 1 where U is a rotation in just one axis. Because only one axis is needed, we can drop either operation A or C, paying attention to the sign on the angles. Reordering the lines of code or signs results in a rotation in the wrong direction.

Correct, operation A unneeded	Correct, operation C unneeded	Incorrect, angles flipped
`Rz(q1,+angle/2); // C`	`CNOT(q0,q1);`	`Rz(q1,-angle/2);`
`CNOT(q0,q1);`	`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`
`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`	`Rz(q1,+angle/2);`
`CNOT(q0,q1);`	`Rz(q1,+angle/2); // A`	`CNOT(q0,q1);`
`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`

Listing 1 Controlled adder subroutine using Fourier transform in the Scaffold language [13].

```
1  // outputs a + b, where a is a ‘width’ bit constant integer
2  // b is an integer encoded on ‘width’ qubits in Fourier space
3  module cADD (
4    const unsigned int c_width, // number of control qubits
5    qbit ctrl0, qbit ctrl1, // control qubits
6    const unsigned int width, const unsigned int a, qbit b[]
7  ) {
8    for (int b_indx=width-1; b_indx>=0; b_indx--) {
9      for (int a_indx=b_indx; a_indx>=0; a_indx--) {
10       if ((a >> a_indx) & 1) { // shift out bits in constant a
11         double angle = M_PI/pow(2,b_indx-a_indx); // rotation angle
12         switch (c_width) {
13           case 0: Rz ( b[b_indx], angle ); break;
14           case 1: cRz ( ctrl0, b[b_indx], angle ); break;
15           case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
16 }}}}}
```

4.2 Defense type 3: Language support for subroutines / unit tests

An obvious defense against coding mistakes in basic subroutines is to use a library of shared code. Doing so helps ensure program correctness by allowing programmers to exhaustively validate small subroutines, in order to bootstrap larger subroutines. Unit testing is especially important in QC as running or simulating large quantum programs is impossible for now.

An additional benefit is logically structured code allows compilers to select the best concrete implementation for the abstract functionality the programmer needs, based on hardware constraints and input parameters [8]. For example, the most cost-efficient implementation for modular exponentiation in Shor’s factorization algorithm depends on how many qubits are available: the compiler can choose from minimum-qubit [2, 9, 38] or minimum-operation [29] implementations for the arithmetic subroutines.

4.3 Bug type 4: Incorrect composition of operations using iteration

Once we have built our basic subroutines, a common pattern in quantum programs is to use iterations to compose subroutines. Listing 1 shows the iteration code for a constant-value adder, showing tricky places in lines 8 through 11 for bugs to crop up, including indexing errors, bit shifting errors, endian confusion, and mistakes in rotation angles. In general this type of iteration code is commonplace in programs that rely on quantum Fourier transforms.

Table 5 Correct classical input a and a^{-1} to Shor's algorithm for factoring 15, using 7 as a guess.

k, the algorithm iteration	0	1	2	3	...
$a = 7^{2^k} \mod 15$	7	4	1	1	...
a^{-1}; $a \times a^{-1} \equiv 1 \mod 15$	13	4	1	1	...

4.4 Defense type 4: Language support for numerical data types

One way to defend against bugs in iteration code is to introduce QC data types for numbers, providing greater abstraction than working with raw qubits. For example, ProjectQ has quantum integer data types [36], while Q# [37] and Quipper [6, 39] offer both big endian and little endian versions of subroutines involving iterations. These QC data types permit useful operators (e.g., checking for equality) that help with debugging and writing assertions.

4.5 Bug type 5: Incorrect deallocation of qubits

Variable scoping is an important language feature in classical computing that ensures proper data encapsulation. In QC, scoping is similarly important for temporary variables known as "ancillary qubits." Anything that happens to a subroutine's ancillary qubits—such as measurement, reinitialization, or lapsing into decoherence—may have unintended effects on the subroutine's outputs[4]. Because improper ancillary qubit deallocation can lead to wrong results, it is important for subroutines to reverse their operations on their ancillary qubits, so that they properly undo any entanglement between the ancillary and output qubits.

We can demonstrate a bug involving incorrect qubit deallocation, by deliberately making a mistake while reversing operations in a subroutine. For example, Shor's algorithm relies on correct pairs modular inverse numbers as input parameters, such as those in Table 5. By feeding an incorrect pair of inputs (e.g., replacing 13 with a 12), the algorithm proceeds to possibly give us wrong output values, as shown in Table 6. At the same time, the mistake prevents the modular multiplication operation from being properly reversed, which has the effect of preventing the ancillary qubits from properly disentangling with other qubits, so they fail to return to their initial values at the end of the algorithm.

4.6 Defense type 5: Assertions on algorithm postconditions

We can use postconditions at the end of algorithms to detect bugs that lead to incorrect deallocation of ancillary qubits. Continuing with our example in Table 6, we see that the cases where ancillary qubits collapse to anything other than zero correspond to cases where the outputs are wrong. That is because the ancillary qubits remain improperly entangled with the output qubits at the end of the algorithm. We can detect these buggy outputs by asserting that ancillary qubits should always return to their initial values. The significance of these observations is that when algorithms work correctly, we typically do not care to measure the value of ancillary qubits as they do not contain information. But in buggy QC algorithm implementations, they are useful side channels for debugging.

[4] As an analogy in classical computing, it is as if accessing an out-of-scope variable can still affect program state; while such behavior is unintuitive, it is a result of how entanglement works in QC.

Table 6 Probability of measuring values of outputs and ancillary qubits of Shor's algorithm, with incorrect inputs ($a^{-1} = 12$ instead of 13 on first iteration). If the ancillary qubits collapse to zero on measurement, the algorithm still succeeds, returning correct outputs of 0, 2, 4, 6 [26, p. 235]. However, the possibility of measuring non-zero for the ancillary qubits indicates a bug.

Probability		**Output measurement**							
		0	1	2	3	4	5	6	7
	0	1/8	0	1/8	0	1/8	0	1/8	0
Ancillary	2	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
qubit	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
measurement	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

Table 7 Grover's amplitude amplification subroutine in two languages, showcasing QC-specific language syntax for reversible computation (rows 2 & 6) and controlled operations (rows 3 & 5).

	Scaffold (C syntax) [13]	**ProjectQ (Python syntax) [36]**

1	`int j;`
	`qbit ancilla[n-1]; // scratch register`
	`for(j=0; j<n-1; j++) PrepZ(ancilla[j],0);`
		`# reflection across`
		`# uniform superposition`

2	`// Hadamard on q`
	`for(j=0; j<n; j++) H(q[j]);`
	`// Phase flip on q = 0...0 so invert q`
	`for(j=0; j<n; j++) X(q[j]);`
		`with Compute(eng):`
		`    All(H) | q`
		`    All(X) | q`

3	`// Compute x[n-2] = q[0] and ...  and q[n-1]`
	`CCNOT(q[1], q[0], ancilla[0]);`
	`for(j=1; j<n-1; j++)`
	`    CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
		`with Control(eng, q[0:-1]):`

4	`// Phase flip Z if q=00...0`
	`cZ(ancilla[n-2], q[n-1]);`
		`    Z | q[-1]`

5	`// Undo the local registers`
	`for(j=n-2; j>0; j-)`
	`    CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
	`CCNOT(q[1], q[0], ancilla[0]);`
		`# ProjectQ automatically`
		`# uncomputes control`

6	`// Restore q`
	`for(j=0; j<n; j++) X(q[j]);`
	`for(j=0; j<n; j++) H(q[j]);`
		`Uncompute(eng)`

5 Case study: Grover's algorithm for database search

So far, we have presented defenses against bugs following two general strategies. One is to use assertions to detect when and where the program has a bug. The other is to use quantum-specific programming language features to prevent bugs altogether: these features include support for subroutines and numerical types for quantum data. Here in this section, we use the Grover's benchmark to showcase two more language features for common QC program patterns: reversible computation and controlled operations.

Grover's search algorithm finds an entry that matches search criteria, among an input data set of size N, with a time cost on the order of $\sqrt{N}$. That represents a polynomial speedup relative to the linear time cost in a classical computer [7].

The Grover's algorithm comprises three parts. First, the input qubits representing the indices of the matching entries are put in a state of superposition, akin to querying all entries at once. Second, the queries are put through a subroutine that checks for the search criteria.

In this case study, our criteria is to find the square root of a number in a Galois field of two elements, a simple abstract algebra setting. Finally in the critical step, the amplitude amplification algorithm primitive amplifies the index that matches the criteria while damping out those that do not. The operations in this final step are prime examples of two QC program patterns, reversible computation and controlled operations. We show in Table 7 how these code patterns are written in two languages, Scaffold [13] and ProjectQ [36].

5.1 Bug type 6: Incorrect composition of operations using mirroring

Section 4.5 discussed how bugs in deallocating ancillary qubits can happen due to bad parameters. Here we see how bugs in deallocating ancillary qubits can happen due to incorrect composition of operations following a mirroring pattern. For example, in Table 7, the operations in rows 2 and 3 are respectively mirrored and undone in rows 6 and 5. These lines of code need careful reversal of every loop and every operation.

5.2 Defense type 6: Language support for reversible computation

Syntax support for reversible computation, such as that in ProjectQ [36], automatically mirrors and inverts sequences of operations, shortening code and reducing mistakes.

5.3 Bug type 7: Incorrect composition of operations using recursion

A common pattern in quantum programs involves performing operations (e.g., add), contingent on a set of qubits known as control qubits. Without language support, this pattern needs many lines of code and manual allocation of ancillary qubits. In the Scaffold code example in Table 7, rows 3 and 5 are just computing the intersection of qubits `q`, with the help of ancillary qubits initialized in row 1, in order to realize the controlled rotation operation in row 4. Furthermore, quantum algorithms often need varying numbers of control qubits in different parts of the algorithm, leading to replicated code from multiple versions of the same subroutine differing only by the number of control qubits[5].

5.4 Defense type 7: Language support for controlled operations

Language support for controlled operations (e.g, ProjectQ) shortens code, preventing mistakes.

6 Conclusion

For the first time, we have access to comprehensive and representative program benchmarks for all major areas of quantum algorithms, implemented in multiple languages, along with input datasets and outputs that are detailed enough to permit cross-validation. Using our experience running and debugging these programs, we presented in this paper defense strategies that facilitate writing bug-free QC code, summarized in Table 8. Successful transplantation of these ideas from classical languages to QC languages can pave the way towards correct and useful quantum programs.

[5] An example appeared in the Shor's case study Listing 1. The addition operation was contingent on control qubits taken as parameters in lines 4 and 5. Depending on how many control qubits were needed, the switch statement in lines 12 through 15 applied the correct operation.

Table 8 Applicability of defense strategies (down) against location of QC program bugs (across).

		input		operations				output
		classical params. §3.1	qubit alloc. §3.2	basic §4.1	iterate §4.3	mirror §5.1	recurse §5.3	qubit dealloc. §4.5
QC specific lang. features	unit testing §4.2		✓	✓	✓	✓	✓	✓
	data types §4.4				✓			
	reverse comp. §5.2				✓	✓		✓
	controlled ops. §5.4				✓	✓	✓	
Assertion checks	preconditions §3.3		✓					
	algo progress §3.4	✓	✓	✓	✓	✓	✓	✓
	postconds. §4.6	✓	✓	✓	✓	✓	✓	✓

References

1 R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. Las Heras, R. Babbush, A. G. Fowler, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, D. Sank, A. Vainsencher, J. Wenner, T. C. White, E. Solano, H. Neven, and John M. Martinis. Digitized adiabatic quantum computing with a superconducting circuit. *Nature*, 534:222 EP–, June 2016. doi:10.1038/nature17658.

2 Stephane Beauregard. Circuit for Shor's Algorithm Using 2N+3 Qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003. URL: http://dl.acm.org/citation.cfm?id=2011517.2011525.

3 Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549:180 EP–, September 2017. doi:10.1038/nature23459.

4 Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr M. Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Y. Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan M. Mniszewski, Balu Nadiga, Dan O'Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Philip Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard J. Zamora, and Wei Zhu. Quantum Algorithm Implementations for Beginners. *CoRR*, abs/1804.03719, 2018. arXiv:1804.03719.

5 A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open Quantum Assembly Language. *ArXiv e-prints*, July 2017. arXiv:1707.03429.

6 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462177.

7 Lov K Grover. From Schrödinger's equation to the quantum search algorithm. *Pramana*, 56(2-3):333–348, 2001.

8 T. Häner, T. Hoefler, and M. Troyer. Using Hoare logic for quantum circuit optimization. *ArXiv e-prints*, September 2018. arXiv:1810.00375.

9 Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring Using 2N + 2 Qubits with Toffoli Based Modular Multiplication. *Quantum Info. Comput.*, 17(7-8):673–684, June 2017. URL: http://dl.acm.org/citation.cfm?id=3179553.3179560.

10 Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. High Performance Emulation of Quantum Circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 74:1–74:9, Piscataway, NJ, USA, 2016. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=3014904.3015003`.

11 Aram Harrow. Why Now is the Right Time to Study Quantum Computing. *XRDS*, 18(3):32–37, March 2012. `doi:10.1145/2090276.2090288`.

12 Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018. URL: `http://stacks.iop.org/2058-9565/3/i=2/a=020501`.

13 Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 1:1–1:10, New York, NY, USA, 2014. ACM. `doi:10.1145/2597917.2597939`.

14 Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, Inc., New York, NY, USA, 2007.

15 N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels. QX: A high-performance quantum computer simulation platform. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, pages 464–469, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association. URL: `http://dl.acm.org/citation.cfm?id=3130379.3130487`.

16 B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. V. James, A. Gilchrist, and A. G. White. Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement. *Phys. Rev. Lett.*, 99:250505, December 2007. `doi:10.1103/PhysRevLett.99.250505`.

17 B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, A. Aspuru-Guzik, and A. G. White. Towards quantum chemistry on a quantum computer. *Nature Chemistry*, 2:106 EP–, January 2010. `doi:10.1038/nchem.483`.

18 R. LaRose. Overview and Comparison of Gate Level Quantum Software Platforms. *ArXiv e-prints*, July 2018. `arXiv:1807.02500`.

19 Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017. `doi:10.1073/pnas.1618020114`.

20 S. McArdle, S. Endo, A. Aspuru-Guzik, S. Benjamin, and X. Yuan. Quantum computational chemistry. *ArXiv e-prints*, August 2018. `arXiv:1808.10402`.

21 J. R. McClean, I. D. Kivlichan, K. J. Sung, D. S. Steiger, Y. Cao, C. Dai, E. Schuyler Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlíček, C. Huang, J. Izaac, Z. Jiang, X. Liu, M. Neeley, T. O'Brien, I. Ozfidan, M. D. Radin, J. Romero, N. Rubin, N. P. D. Sawaya, K. Setia, S. Sim, M. Steudtner, Q. Sun, W. Sun, F. Zhang, and R. Babbush. OpenFermion: The Electronic Structure Package for Quantum Computers. *ArXiv e-prints*, October 2017. `arXiv:1710.07629`.

22 N.D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007.

23 Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong. Quantum Computing for Computer Architects, Second Edition. *Synthesis Lectures on Computer Architecture*, 6(1):1–203, 2011. `doi:10.2200/S00331ED1V01Y201101CAC013`.

24 Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

25 Michele Mosca. Quantum algorithms. In *Encyclopedia of Complexity and Systems Science*, pages 7088–7118. Springer, 2009.

26 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

27 Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. Quantum information and computation for chemistry. *arXiv preprint arXiv:1706.05413*, 2017.

28 S. Patil, A. JavadiAbhari, C. Chiang, J. Heckey, M. Martonosi, and F. T. Chong. Characterizing the performance effect of trials and rotations in applications that use Quantum Phase Estimation. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 181–190, October 2014. `doi:10.1109/IISWC.2014.6983057`.

29 Archimedes Pavlidis and Dimitris Gizopoulos. Fast Quantum Modular Exponentiation Architecture for Shor's Factoring Algorithm. *Quantum Info. Comput.*, 14:649–682, May 2014. URL: `http://dl.acm.org/citation.cfm?id=2638682.2638690`.

30 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*, 5:4213 EP–, July 2014. `doi:10.1038/ncomms5213`.

31 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. `doi:10.22331/q-2018-08-06-79`.

32 M. Roetteler, K. M. Svore, D. Wecker, and N. Wiebe. Design automation for quantum architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1312–1317, March 2017. `doi:10.23919/DATE.2017.7927196`.

33 Jacob T. Seeley, Martin J. Richard, and Peter J. Love. The Bravyi-Kitaev transformation for quantum computation of electronic structure. *The Journal of Chemical Physics*, 137(22):224109, 2012. `doi:10.1063/1.4768229`.

34 Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. `doi:10.1137/S0097539795293172`.

35 R. S. Smith, M. J. Curtis, and W. J. Zeng. A Practical Quantum Instruction Set Architecture. *ArXiv e-prints*, August 2016. `arXiv:1608.03355`.

36 Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, January 2018. `doi:10.22331/q-2018-01-31-49`.

37 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, pages 7:1–7:10, New York, NY, USA, 2018. ACM. `doi:10.1145/3183895.3183901`.

38 Yasuhiro Takahashi and Noboru Kunihiro. A Quantum Circuit for Shor's Factoring Algorithm Using 2N + 2 Qubits. *Quantum Info. Comput.*, 6(2):184–192, March 2006. URL: `http://dl.acm.org/citation.cfm?id=2011665.2011669`.

39 Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. Programming the Quantum Future. *Commun. ACM*, 58(8):52–61, July 2015. `doi:10.1145/2699415`.

40 Dave Wecker, Bela Bauer, Bryan K. Clark, Matthew B. Hastings, and Matthias Troyer. Gate-count estimates for performing quantum chemistry on small quantum computers. *Phys. Rev. A*, 90:022305, August 2014. doi:10.1103/PhysRevA.90.022305.

41 J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109:735–750, March 2011. doi:10.1080/00268976.2011.552441.