Giallar: Push-Button Verification for the Qiskit Quantum Compiler

Runzhou Tao
Columbia University
New York, NY, USA
runzhou.tao@columbia.edu

Yunong Shi
Amazon
New York, NY, USA
shiyunon@amazon.com

Jianan Yao
Columbia University
New York, NY, USA
jianan@cs.columbia.edu

Xupeng Li
Columbia University
New York, NY, USA
xupeng.li@columbia.edu

Ali Javadi-Abhari
IBM Research
Yorktown Heights, NY, USA
ali.javadi@ibm.com

Andrew W. Cross
IBM Research
Yorktown Heights, NY, USA
awcross@us.ibm.com

Frederic T. Chong*
The University of Chicago
Super.tech
Chicago, IL, USA
chong@cs.uchicago.edu

Ronghui Gu†
Columbia University
CertiK
New York, NY, USA
ronghui.gu@columbia.edu

Abstract

This paper presents Giallar, a fully-automated verification toolkit for quantum compilers. Giallar requires no manual specifications, invariants, or proofs, and can automatically verify that a compiler pass preserves the semantics of quantum circuits. To deal with unbounded loops in quantum compilers, Giallar abstracts three loop templates, whose loop invariants can be automatically inferred. To efficiently check the equivalence of arbitrary input and output circuits that have complicated matrix semantics representation, Giallar introduces a symbolic representation for quantum circuits and a set of rewrite rules for showing the equivalence of symbolic quantum circuits. With Giallar, we implemented and verified 44 (out of 56) compiler passes in 13 versions of the Qiskit compiler, the open-source quantum compiler standard, during which three bugs were detected in and confirmed by Qiskit. Our evaluation shows that most of Qiskit compiler passes can be automatically verified in seconds and verification imposes only a modest overhead to compilation performance.

CCS Concepts: • **Theory of computation → Quantum information theory**; **Program verification**.

Keywords: quantum computing, compiler verification, automated verification

ACM Reference Format:
Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523431

1 Introduction

Quantum compilers are essential components in the quantum software stack, bridging quantum applications to hardware. However, correctly implementing a quantum compiler is difficult due to two reasons: 1) quantum compilers have to obey quantum mechanics rules that are highly nonintuitive [48]; and 2) quantum compilers must perform heavy optimizations to fit quantum programs into real quantum devices of limited qubit lifetime and connectivity [21]. These challenges make quantum compilers error-prone. For example, Bugs4Q [49] finds 27 bugs in the Qiskit compiler [39], the most widely-used open-source quantum compiler with more than 100K users from 171 countries. Undetected bugs in the Qiskit compiler can corrupt the computation results of the millions of simulations and real runs on quantum devices. Eliminating bugs in quantum compilers becomes crucial for the success of near-term quantum computation.

*Frederic T. Chong is also Chief Scientist at Super.tech and an advisor to Quantum Circuits, Inc.

†Ronghui Gu is also the Founder of CertiK.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523431

In this paper, we present Giallar, a toolkit that helps programmers write quantum compiler passes and formally verify their correctness in a *push-button* manner. Giallar requires *no* manual annotations, invariants, specifications, or proofs about the implementation of the quantum compiler pass. Giallar performs verification to check if the compiler pass preserves the semantics of quantum circuits. An executable pass implementation is produced if the verification succeeds. If there is a bug, Giallar produces a counterexample to help identify and fix the cause.

The main challenge in applying formal verification to building correct quantum compilers is to minimize proof burden. Recent efforts [2, 15] have shown that it is feasible to manually verify the correctness of quantum compiler passes using interactive theorem provers such as Coq [5]. However, writing such proofs requires a significant time investment from formal verification experts, and the size of proofs can be several times or even more than an order of magnitude larger than that of the compiler implementation, making the proofs expensive to develop and maintain. For example, Hietala et al. [15] reported that verifying circuit mapping, a single transformation pass consisting of 70 lines of code, requires writing 2,100 lines of proofs. These verification frameworks are impractical for verifying fast-moving and frequently changing quantum compilers such as Qiskit, which has 683 commits on the main branch from 64 contributors in the first 10 months of the year 2021 [38].

To allow non-formal-verification-experts to develop correct quantum compilers without such a proof burden, Giallar provides fully automated reasoning. Conceptually, showing that a quantum compiler pass is correct involves proving that it preserves the semantics for any input quantum circuit in all possible execution paths. In practice, automating such a proof faces *classical* and *quantum* challenges.

On the one hand, a quantum compiler is a classical program that intensely uses unbounded loops and complex utility functions (containing nested loops and recursions) to perform transformations and optimizations based on the information of the whole circuit. These program features in general are hard to reason about automatically. For example, automated verification frameworks such as Alive [23] and Hyperkernel [33] require the input program to be loop-free and recursion-free, or only have bounded loops.

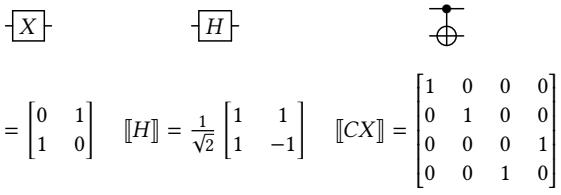
On the other hand, the correctness of a quantum compiler pass is usually defined as the semantics preservation property for quantum circuits, while efficient equivalence checking for general quantum circuits is still beyond reach. Previous quantum verification works [1, 4, 15] rely on users manually reasoning about the equivalence of quantum circuits either using the denotational semantics [34] (i.e., the matrix representation), which requires exponential time and memory to compute, or using the path-sum semantics [1], which only supports a restricted subset of quantum states. Neither of these approaches is feasible for automated verification of quantum compilers.

Our Giallar toolkit addresses the above challenges by leveraging domain-specific knowledge of quantum compilation. First, most of the unbounded loops in quantum compilers follow one of a few specific patterns to traverse the input quantum circuit. Giallar abstracts these patterns into three *loop templates* for users to write unbounded loops, whose loop invariants can be automatically inferred without any user input. The compiler pass containing unbounded loops then becomes symbolically executable by reducing the loops with the inferred invariants. For each loop, Giallar will also generate a separate proof goal that the symbolic execution of the loop body indeed retains the inferred invariants. Giallar formulates such proof goals as SMT problems and invokes Z3 [8] to solve them. Our three loop templates cover all the unbounded loops in all verified Qiskit passes, while new loop templates can be easily introduced to meet future needs. As for complex utility functions that contain nested loops and recursions, Giallar provides a verified library of utility functions that is shared by multiple passes, such that their invocations can be replaced by their specifications during the symbolic execution.

Second, instead of directly using the matrix representation to check the equivalence of the input and output quantum circuits, Giallar shows that the output quantum circuit can be obtained from the input circuit through a sequence of *equivalent rewrites*. To define such rewrite rules, we define symbolic representation and execution for quantum circuits, which are different from the symbolic execution of the compiler implementation mentioned above. All rewrite rules operate the symbolic quantum circuit and preserve the results of the symbolic execution. The set of rewrite rules provided by Giallar is general enough to cover common quantum compilations and small enough to enable efficient checks. The rewrite rules are manually verified in the Coq proof assistant [5], and the verification is done once and for all.

Third, in contrast to existing automated verification frameworks [23, 32, 44] that require users to provide specifications, Giallar introduces a set of Python virtual classes for different types of passes in Qiskit. Giallar can automatically generate proof obligations for the pass implementations inheriting these virtual classes.

We have used Giallar to implement and verify 44 out of 56 compiler passes in 13 versions (from v0.19 to v0.32) of the Qiskit compiler. Among 12 failed passes, eight passes deal with pulse-level behaviors, two passes rely on external solvers, one pass involves a randomized routing algorithm, and one pass produces an approximated circuit within a given error bound. These passes are not supported by Giallar, and are also costly or infeasible to manually verify using existing quantum verification frameworks [1, 2, 4, 15]. During the verification, we found three critical bugs in (and confirmed by) the Qiskit team, two of which are unique to



$$[\![X]\!] = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad [\![H]\!] = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad [\![CX]\!] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 1. Circuit diagram symbols (top) and the denotational semantics (bottom) for several 1-qubit and 2-qubit gates.

quantum computing. Our evaluation shows that most of the Qiskit compiler passes can be automatically verified in seconds and the verified compiler passes only have modest performance overhead compared with the unverified Qiskit implementation.

This paper makes the following contributions:

- The Giallar toolkit that allows non-formal-verification-experts to build provably correct quantum compilers in the face of frequent changes and new features.
- A domain-specific approach using loop templates, virtual classes for passes, and verified utility library to fully automate the verification of quantum compilers.
- A set of verified rewrite rules that enables the efficient equivalence checking for quantum circuits.
- A case study of implementing and verifying the compiler passes of Qiskit, the most widely-used quantum compiler, using Giallar. Three critical bugs have been found during the verification and confirmed by Qiskit.

2 Background

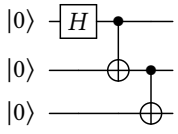
We first introduce the necessary background on the verification of quantum computing and quantum compilation. For more details of quantum computing, please refer to [27].

2.1 Quantum Basics

Quantum states are represented as 2-dimensional complex vectors, named qubits, e.g., $|0\rangle = (1,0)^T$ and $|1\rangle = (0,1)^T$. Quantum gates are operations of quantum states that can be represented by unitary matrices (see Figure 1). In contrast with classical computing, an n-qubit state (or gate) is represented by a 2^n-dimensional vector (or a $2^n \times 2^n$ vector), leading to the exponential cost in space and time required to directly simulate quantum programs using matrix and vector representations.

2.2 Quantum Program

Quantum programs process quantum states with quantum gates. They are often represented graphically with gates as nodes and qubits as wires (see Figure 1). The most widely-used quantum programming language is OpenQASM [6].



```
//GHZ circuit
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
h q[0];
cx q[0],q[1];
cx q[1],q[2];
```

Figure 2. Circuit diagram (left) and the OPENQASM IR (right) of a simple GHZ circuit [10].

$$\begin{aligned} [\![\texttt{skip}]\!]_{nqreg} &:= I_{qreg} \\ [\![U]\!]_{nqreg} &:= \texttt{matrix}(U_{q_1,\ldots,q_n}) \otimes I_{qreg \setminus \{q_1,\ldots,q_n\}} \\ [\![C_1\,;C_2]\!]_{nqreg} &:= [\![C_1]\!]_{nqreg} \times [\![C_2]\!]_{nqreg} \end{aligned}$$

Figure 3. Denotational semantics of quantum circuits in Giallar, where `matrix` denotes the unitary matrices of the quantum operations and `qreg` denotes the set of qubit registers.

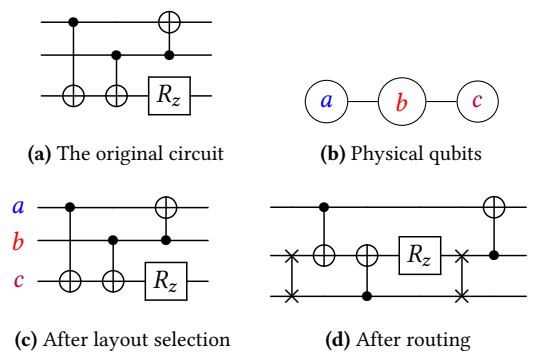
Figure 2 shows an example of a 3-qubit circuit in graphical representation and in OpenQASM.

Programs considered in Giallar follow a variant of the OpenQASM language as below.

$$\begin{aligned} P := \;& \texttt{skip} \\ \mid\; & U(q_1, \ldots, q_n) \\ \mid\; & P_1; P_2 \end{aligned}$$

Empty circuit is denoted as `skip`; applying an n-qubit gate on selected qubits $q_1, \ldots q_n$ is denoted as $U(q_1, \ldots, q_n)$; concatenation of two circuits P_1 and P_2 is denoted as $P_1; P_2$. Features in OpenQASM that are not supported by existing hardware such as classical control flow are not included in our syntax. Nevertheless, this syntax is general enough to support a wide range of gate representations, circuit transformations, and various targeting hardware. This restriction on syntax is also a common practice in previous work on manual verification of quantum compilers [15]. The input and output of each quantum compilation pass are both quantum programs.

Denotational semantics. Figure 3 shows the denotational semantics of a quantum circuit C, which is defined as its corresponding unitary matrix and is denoted as $[\![C]\!]_{nqreg}$, where *nqreg* is the number of qubits in the quantum register used in the circuit. The denotational semantics of an empty circuit with *nqreg* qubits is the identity matrix of size *nqreg*. The semantics for a quantum gate is the tensor product of its matrix representation on its qubit operands and the identity matrix on other unrelated qubits. The semantics of the concatenation of two quantum programs is the multiplication of their matrix representations.



(a) The original circuit **(b)** Physical qubits

(c) After layout selection **(d)** After routing

Figure 4. An example of layout selection and routing passes acting on a quantum circuit. (a) The original quantum circuit. Each line represents a logical qubit. (b) The target quantum computer's qubit arrangement, lines denote that two-qubit gates are allowed between the qubit pair. (c) After layout selection, the logical qubits in the circuit get mapped to physical qubits. At this point, the circuit is still not runable on the target quantum computer because there is a 2-qubit gate between a and c. (d) After routing, swap gates are added into the circuit so that all 2-qubit gates are allowed.

2.3 Quantum Compilation

Quantum compilation is the process of translating high-level quantum circuit descriptions into optimized low-level circuits that are executable on hardware. Most quantum compilers follow a design philosophy resembling that of LLVM [18]: circuit IRs are sequentially fed into a cascade of compiler components, called *passes*, to be transformed and optimized. The Qiskit compiler has *seven* types of compiler passes: layout selection, routing, basis change, optimizations, circuit analysis, synthesis, and additional assorted passes.

Layout selection and routing passes. A layout selection pass maps logical qubits in the program to physical qubits on a specific hardware. A routing pass ensures that the quantum circuit conforms to the topological constraints of the quantum hardware. For example, in Figure 4, the quantum hardware has 3 physical qubits a, b, and c with the topology constraints that 2-qubit gates can only be preformed between $a-b$ and $b-c$. The layout selection pass first assigns the three logical qubits in the circuit to a, b, and c. The routing pass then inserts swap gates so that all 2-qubit gates satisfy the topological constraints. Because CNOT gates cannot be preformed between a and c, a swap gate between b and c is inserted before the first CNOT gate. To perform the last CNOT gate, b and c need to be swapped back.

Other passes. A basis change pass helps with the decomposition of quantum circuits into the gate set supported by a target hardware backend. An optimization pass includes various circuit-rewriting-based optimizations such as gate cancellation [24], scheduling optimization [42], noise adaptation [28], and crosstalk mitigation [29]. A circuit analysis pass does not modify the circuits but returns important information about the circuits. A synthesis pass performs large unitary matrix decomposition. Additional assorted passes perform miscellaneous tasks such as circuit validation.

```
1 import giallar
2 class SimpleCXCancellation(GeneralPass):
3     def run(self, input):
4         remain = input.copy()
5         output = QCircuit()
6         while remain.size() != 0:
7             gate = remain[0]
8             if gate.isCXGate():
9                 next = next_gate(remain, 0)
10                g = remain[next]
11                if g.isCXGate() and
12                   g.qubits == gate.qubits:
13                    remain.delete(next)
14                else:
15                    output.append(gate)
16            else:
17                output.append(gate)
18            remain.delete(0)
19        return output
```

Figure 5. A simplified implementation of the CXCancellation pass.

2.4 The Z3Py Tool

Z3Py supports the symbolic execution of sequential Python code without loops and branches, and introduces two primitives `assume(cond)` and `assert(cond)`. The `assume(cond)` primitive adds the condition `cond` into the assumption list. The `assert(cond)` primitive calls the Z3 SMT solver to check whether the current symbolic execution state satisfies `cond` given the assumption list. For example, given the code snippet below,

```
assume(x >= 3)  | x >= 3 added to assumptions
y = x * x       | Symbolic execution: y = x * x
assert(y > x)   | SMT check x >= 3 -> x * x > x success
z = y + 1       | Symbolic execution: z = x * x + 1
assert(z > 10)  | SMT check x >= 3 -> x * x + 1 > 10 fail
```

Z3Py will output "verified" for the first assertion and a counterexample "$x = 3$" for the second assertion.

3 The Giallar Workflow

The goal of the Giallar toolkit is to allow quantum programmers without much formal verification background to write provably correct quantum compiler passes, which can be readily integrated into the open-source Qiskit compiler and used in real-world quantum experiments.

Giallar consists of five major components (see Figure 6): 1) The Giallar preprocessor that translates the compiler implementation into symbolically executable code; 2) A set of loop templates that can be used to infer loop invariants for unbounded loops in the pass implementation; 3) A verified

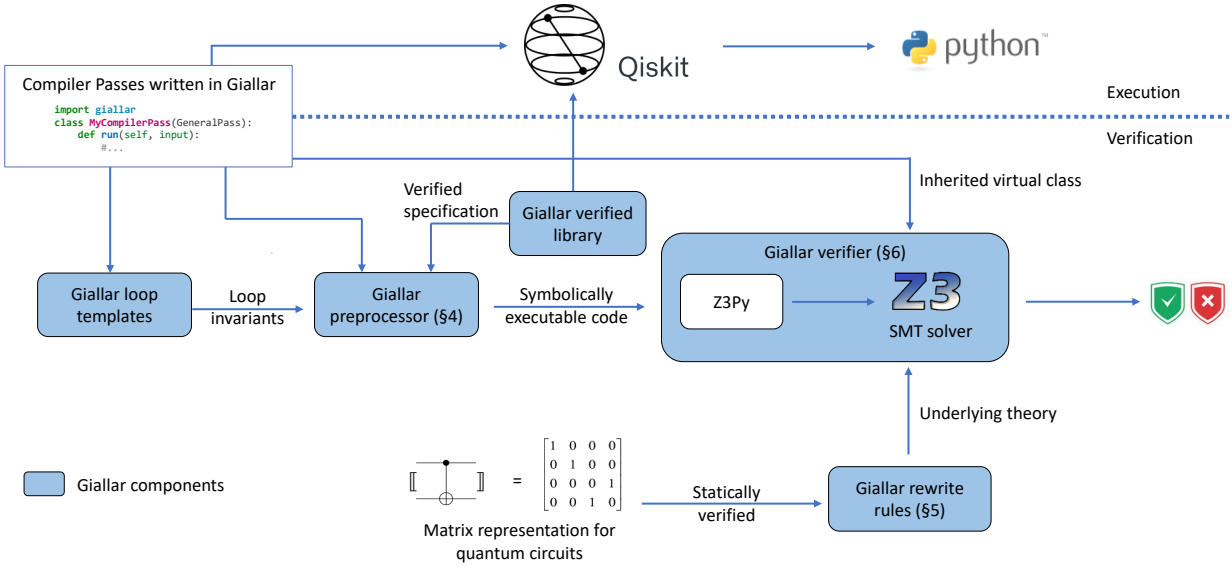


Figure 6. Giallar workflow.

library of utility functions; 4) A set of rewrite rules for quantum circuits, enabling the efficient equivalence checking for quantum circuits; 5) The Giallar verifier that generates the proof goal based on the type of the compiler passes and verifies if the symbolic execution results satisfy the goal using rewrite rules.

In the rest of this section, we will use a simplified version of CXCancellation pass implemented in Giallar (see Figure 5) as a running example to give a high-level overview on Giallar. The CXCancellation pass is an optimization pass that scans through the gates of the input circuit and cancels out adjacent CNOT (also called CX) gates that are on the same pair of qubits. The variable `remain` represents the part of the circuit that has not been scanned yet. During the scanning, if the current `gate` is not a CNOT gate, it will be directly appended to the `output` circuit. If the current `gate` is a CNOT gate, the pass will then find the next gate in the remaining circuit that shares the qubits with the current `gate`, denoted as `g`. If `g` is also a CNOT gate on the same pair of qubits, `gate` and `g` will be cancelled out and will not appear in the `output` circuit.

To enable the push-button verification for this pass, we need to address two challenges: 1) the pass implementation contains an unbounded loop that forbids the automated symbolic execution, and 2) checking the equivalence of input and output circuits using denotational semantics requires to solve constraints over matrices with arbitrary sizes, which is infeasible for existing solvers.

Figure 6 shows the workflow of the push-button verification for the CXCancellation pass using Giallar. The key steps are described below.

Infer loop invariants. The key to automate the symbolic execution for unbounded loops is to infer the loop invariants, which is hard for general-purpose programs but is tractable for quantum compiler passes with domain-specific knowledge. For example, the unbounded loop in the the CX-Cancellation pass scans through the input circuit to build the output circuit while maintaining a list of remaining gates. This loop pattern is quite common in many quantum compiler passes such as optimization passes. For this loop pattern, the loop invariant should be that the concatenation of the currently built part of the output circuit and the remaining gates must be equivalent to the input circuit, i.e., their denotational semantics defined using the matrix representations are equivalent for any *nqreg*:

$$[[\texttt{?output}; \texttt{?remaining\_gates}]]_{nqreg} \equiv [[\texttt{?input}]]_{nqreg}.$$

The Giallar preprocessor will statically analyze the loop implementation and infer the assignment to the variable placeholders; for example, the `remain` list should be assigned to the variable placeholder "`?remaining_gates`." After the variable assignment, the invariant becomes:

$$[[\texttt{output}; \texttt{remain}]]_{nqreg} \equiv [[\texttt{input}]]_{nqreg}.$$

Because the remaining gate list is empty at the end of the loop (i.e., the loop condition "`remain.size() != 0`" becomes false at line 6 in Figure 5), the Giallar preprocessor will replace the entire loop with an assumption that $[[\texttt{output}]]_{nqreg} \equiv [[\texttt{input}]]_{nqreg}$ and generate the following transformed code that will be symbolically executed by Z3Py later.

```
def run(self, input):
    output = uninterpreted_circuit()
    assume(equivalent(output, input))
    return output
```

To validate the inferred loop invariant, Giallar will generate a separate proof goal that the invariant holds on the symbolic execution results of the loop body.

Expand branch statements. To support branches in the Z3Py tool, Giallar generates the sequential code for each branch. As for the loop body in the CXCancellation pass, three pieces of sequential code will be generated: 1) `gate` is a CNOT gate and a matching gate is found to perform the cancellation; 2) `gate` is a CNOT gate and no matching gates are found; and 3) `gate` is not a CNOT gate. The branch conditions will be added as assumptions to the generated sequential code. For example, the transformed code for the first branch is as follows:

```
gate = remain[0]
assume(gate.isCXGate())
next = next_gate(remain, 0)
g = remain[next]
assume(g.isCXGate() and g.qubits == gate.qubits)
remain.delete(next)
remain.delete(0)
```

Replace utility functions with specifications. The Qiskit compiler implementations rely on many utility functions, which are shared by different passes and may contain code that is hard to automatically verify. Giallar extracts a library of utility functions and manually verifies all the functions with respect to their specifications. A compiler pass can then be retrofitted to invoke utility functions in Giallar's verified library. Such invocations will be replaced by the corresponding specifications of the utility functions during the symbolic execution. Take the `next_gate` utility function that is used in four optimization passes of Qiskit as an example. This function scans through the circuit and returns the index of the next gate that shares a qubit with the current gate. Instead of repeatedly verifying this function whenever it is invoked, Giallar replaces its invocations with the verified specification. For example, its invocation at line 9 in Figure 5 will be replaced by the following specification about the returned integer `x` of `next_gate(remain, 0)`: 1) `x` is a valid index of the `remain` circuit, i.e., $0 \leq$ `x` $<$ `remain.size()`; 2) `x` is after the gate 0, i.e., `x` > 0; 3) there is no gate between gates 0 and `x` that shares a qubit with gate 0; and 4) gate `x` shares a qubit with gate 0.

Verify proof goals with rewrite rules. The Giallar verifier first symbolically executes the transformed code using the Z3Py tool. The Z3 SMT solver will then be queried to solve the proof goals using the symbolic execution results. As for the CXCancellation pass, we need to prove that all three branches will preserve the equivalence of circuits, i.e., $[[\texttt{output; remain}]]_{nqreg} \equiv [[\texttt{input}]]_{nqreg}$. The only non-trivial case is the branch where a cancellation happens.

Because the circuits are equivalent before executing the branch, and the `input` and `output` circuits remain unchanged in this branch (since CNOT gates are cancelled out and will not be appended to `output`), we only need to prove that the `remain` list of gates has the same denotational semantics after cancelling out two adjacent CNOT gates. Such a proof goal can be stated using the following matrix representations:

$$\left(\texttt{matrix}(CNOT_{q_1,q_2}) \otimes I_{\{1,\ldots,nqreg\}\setminus\{q_1,q_2\}}\right)^2 = I_{2^{nqreg}}$$

For a quantum circuit of n qubits, the above proof goal requires the equivalence check of matrices with a size $2^n \times 2^n$, which is impractical to solve using any existing solvers. The fact that n, q_1, and q_2 are arbitrary makes such a check even harder.

Giallar introduces a set of rewrite rules to enable the equivalence check for quantum circuits at the symbolic level without the need to reason about their denotational semantics. These rules are also qubit-based, meaning that we only need to prove that the two related qubits (q_1 and q_2) are equivalent and do not need to worry about n and the relative location of q_1 and q_2. For example, Giallar provides the following two rewrite rules to cancel out two adjacent CNOT gates, with which the equivalence check for `remain` can be solved automatically and efficiently by Z3.

$$\texttt{app}(CNOT\ q_1\ q_2; CNOT\ q_1\ q_2, q_1) \equiv q_1$$
$$\texttt{app}(CNOT\ q_1\ q_2; CNOT\ q_1\ q_2, q_2) \equiv q_2.$$

4 The Giallar Preprocessor

The Giallar preprocessor aims to soundly transform the compiler source code, with complex control flow and external function calls, into simple sequential code with verification conditions, thus symbolically executable and verifiable using Z3Py. A verification condition is defined as a Hoare triple $\{P\}C\{Q\}$, with a pre-condition list P and a post-condition list Q [16]. The Hoare triple means that if the program state before executing the code C satisfies P, then after executing C, the resulting state must satisfy Q. Giallar uses `assume` and `assert` primitives in Z3Py to represent the pre- and post-conditions respectively. The Giallar preprocessor parses and transforms the pass implementation with complex control flows into sequential code and corresponding verification conditions.

Branch statements. The Giallar preprocessor expands all branch statements. The verification condition of a branch is expanded into two separate verification conditions—one for each branch. The branch condition will be added to the list of pre-conditions for the transformed sequential code representing the "true" branch, while the negation of the branch condition will be added to the list of pre-conditions for the "false" branch. These pre-conditions are then used to generate further verification conditions. Note that Giallar requires that all branch conditions must be representable as SMT formulas to enable the push-button verification. Although this expansion approach may lead to an exponential number of verification conditions, fortunately, the number

of branches in real Qiskit compiler passes is usually smaller than nine and remains tractable.

Loop statements. Unbounded loops are hard to automatically verify since their verification requires a sufficiently strong loop invariant, which is generally undecidable to compute and usually provided by the user. Fortunately, in the domain of quantum compilation, this problem can be practically solved as loops in quantum compiler passes often follow one of a few fixed patterns. Giallar provides *three* loop templates as library functions, which can be used to rewrite all the loops in the Qiskit compiler passes. Giallar then statically analyzes the loop implementation and automatically determines how the placeholders in the loop template can be mapped to the variables in the loop implementation.

Each of Giallar's loop template pre-defines one shape of the loop invariants. For example, `iterate_all_gates(circ, func)` is a template for loops that iterate over all the gates in `circ` and apply the same function `func` (i.e., the loop body) to each gate to generate the new circuit, denoted as `new_circ`. Since this compiler pass preserves the semantics of the circuits, the generated `new_circ` should be equivalent to a prefix of the original circuit. Thus, the invariant for this loop template is that, at the i-th iteration of the loop, `new_circ` should be equivalent to the first i gates in the original circuit. This loop template is implemented as follows:

```
1 def iterate_all_gates(circ, func):
2     # subgoal
3     assertion.push()
4     i = Int("i")
5     n = circ.size()
6     cur_circ = QCircuit()
7     assume(i >= 0)
8     assume(i + 1 < n)
9     assume(equiv_part(cur_circ, circ, i))
10    new_circ = func(cur_circ, circ[i])
11    assert(equiv_part(new_circ, circ, i+1))
12    assertion.pop()
13
14    ret_circ = DAG()
15    assume(equiv_part(ret_circ, circ, circ.size()))
16    return ret_circ
```

Lines 7-12 check if the loop body (i.e., `func`) preserves the loop invariant. In the verification process, when Giallar invokes this loop template, it will generate and try to prove the subgoal that if the current circuit `cur_circ` is equivalent to the first i gates of the original circuit `circ` before the i-th iteration, the newly generated circuit `new_circ` after this iteration must be equivalent to the first $i + 1$ gates of `circ`. Once this subgoal is proved, the loop invariant is valid and a new pre-condition stating that the result of the loop is a circuit that is equivalent to `circ` will be added into the pre-condition list (see line 15). To synthesize the exact loop invariant from the code, Giallar will infer the variable name in the loop body that should be mapped to the `circ` argument of `iterate_all_gates(circ, func)`.

The other two loop templates provided by Giallar are `while_gate_remaining` and `collect_runs`. The unbounded loop in the `CXCancellation` pass (see Section 3) can be implemented using the `while_gate_remaining` template, which maintains a remaining gate list to be scanned. The loop invariant for this template is that at each iteration of the loop, the concatenation of the currently built part of output circuit and the remaining gates is equivalent to the input circuit. The `collect_runs` template is the batch version of the `iterate_all_gates` template and is used in passes such as `commutative_cancellation` (see Section 7.2). In this template, the input circuit is partitioned into several batches and each loop round will transform one batch of the circuit into an equivalent circuit. The invariant for this template is that the currently built output circuit in the i-th iteration of the loop is equivalent to the combination of first i batches of the input circuit.

Utility function calls. Many compiler passes are implemented using some shared utility functions, including circuit manipulating functions such as `next_gate`, gate optimization functions such as `merge`, and coupling map related functions such as `shortest_path`. To enable the push-button verification for Qiskit compiler passes without the need to unfold and repeatedly verify these shared utility functions at all their invocations, Giallar pre-verifies all these functions with respect to their specifications and replaces their invocations with the specifications during the symbolic execution. Take the `next_gate` utility function invoked by four passes (CXCancellation, MergeAdjacentBarriers, RemoveFinalMeasure and RemoveDiagBeforeMeasure) as an example. Giallar models a quantum gate as a record type with two fields—an operation name and a qubit list (analogous to the opcode and operands in classical computing) and model a quantum circuit as a list of gates. Giallar verifies that the `next_gate` function meets the specification that scans through the circuit list and finds the index of the first gate that shares a qubit with the given gate. Although the library of utility functions is verified manually, this verification effort is done once and for all can be re-used in future Qiskit compiler passes.

Note that Giallar's verified utility library implements a quantum circuit as a list of gates, while the original Qiskit library implements a circuit as a directed acyclic graph (DAG) of gates. Giallar's design significantly simplifies the library's verification since lists are much easier to reason about in Coq than DAGs. To integrate passes implemented using our verified library into the Qiskit platform, Giallar provides conversion functions to convert circuits implemented in different data structures, as well as a Qiskit wrapper that performs the following steps: 1) it first converts the input DAG circuit from the Qiskit compilation flow to the OpenQASM IR; 2) then it invokes the compiler pass written using Giallar to compile the converted circuit represented as a gate list; and

3) it finally converts the compiled circuit back to the corresponding DAG representation. The evaluation section (see Section 8) later shows that Giallar's verified library, conversion functions, and the Qiskit wrapper will only introduce negligible performance overhead compared with the original implementation.

Non-critical statements. Many Qiskit compiler passes contain *non-critical* statements and function invocations that will not affect the generated circuits, such as the functions that output the number of gates and the number of tensor factors in the circuit for the debugging purpose. Take the NumTensorFactors pass which calculates the number of tensor factors in the circuit as an example. These *non-critical* statements will not affect our semantic-preserving proofs for the generated circuits and will be discarded during the pre-processing before the symbolic execution.

5 Rewrite Rules for Quantum Circuits

Many of the generated verification conditions can be proved directly in Z3Py, except for the equivalence assertion stating that two quantum circuits with variable lengths are equivalent, i.e., given any input quantum state, the two circuits will produce the same output quantum state. Checking the equivalence of quantum circuits using their matrix representations is intractable for Z3 due to the exponential computational cost, especially when the matrices have variable sizes. This is a unique challenge for quantum computing and is crucial for automated verification of quantum compiler passes. To address this challenge, besides the symbolic execution of pass implementations, we introduce a new layer of symbolic representation and execution for quantum circuits, as well as rules to reduce gates and perform circuit rewriting. Note that one should not confuse the symbolic execution of quantum circuits described in this section with the symbolic execution of pass implementations provided by Z3Py—the former takes quantum states as input/output while the latter takes quantum circuits as input/output.

Symbolic execution of quantum circuits. A multi-qubit quantum register Q is symbolically represented as an array of symbolic qubits $(q_1, \cdots, q_n)$. Giallar defines the symbolic function $\texttt{app}_{1q}(U, q)$ to denote the resulting qubit of applying the 1-qubit gate U on q, and defines $\texttt{app}_{2q}(U, q_1, q_2, k)$ to denote the k-th resulting qubit ($k \in \{1, 2\}$) of applying the 2-qubit gate U on q_1 and q_2. The result of applying the whole circuit C to a symbolic quantum register is represented as $\texttt{app}(C, Q)$, which can be executed by applying each gate in sequence on Q and is defined as follows:

$$\begin{aligned}
\texttt{app}(skip, Q) &\rightharpoonup Q \\
\texttt{app}(C_1; C_2, Q) &\rightharpoonup \texttt{app}(C_2, \texttt{app}(C_1, Q)) \\
\texttt{app}(U(i), (q_1, \ldots, q_n)) &\rightharpoonup (q_1, \ldots, \texttt{app}_{1q}(U, q_i), \ldots, q_n) \\
\texttt{app}(U(i, j), (q_1, \ldots, q_n)) &\rightharpoonup \\
&(q_1, \ldots, \texttt{app}_{2q}(U, q_i, 1), \ldots, \texttt{app}_{2q}(U, q_j, 2), \ldots, q_n).
\end{aligned}$$

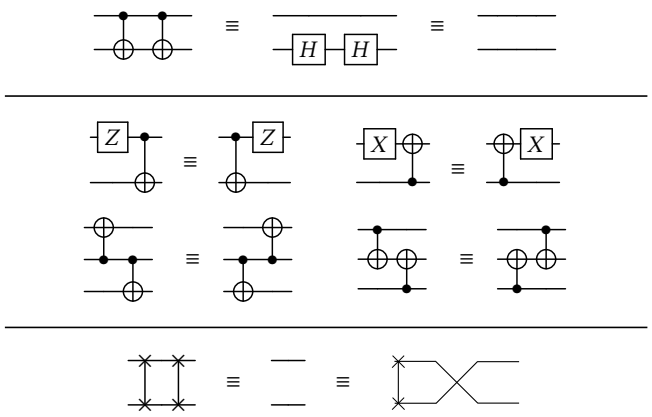


Figure 7. Examples of rules for reducing and rewriting circuits. They are cancellation rules (above), commutativity rules (2nd, 3rd line), and swap rules (bottom).

For example, applying the following GHZ circuit

$$GHZ := H(0); CX(0, 1); CX(1, 2)$$

on the register (q_0, q_1, q_2) will result in (q_0', q_1', q_2') such that

$$\begin{aligned}
q_0' &= \texttt{app}_{2q}\left(CX, \texttt{app}_{1q}(H, q_0), q_1, 1\right) \\
q_1' &= \texttt{app}_{2q}\left(CX, \texttt{app}_{2q}(CX, \texttt{app}_{1q}(H, q_0), q_1, 2), q_2, 1\right) \\
q_2' &= \texttt{app}_{2q}\left(CX, \texttt{app}_{2q}(CX, \texttt{app}_{1q}(H, q_0), q_1, 2), q_2, 2\right).
\end{aligned}$$

Rewrite rules. To efficiently check the equivalence of the symbolic execution of quantum circuits, Giallar introduces a set of 20 rules for reducing and rewriting gates that are inserted by compiler passes, eight of which are shown in Figure 7. For example, a $SWAP$ gate will swap the two qubit arguments and its application can be reduced by the following swap rules:

$$\begin{aligned}
\texttt{app}_{2q}(SWAP, q_1, q_2, 1) &\equiv q_2 \\
\texttt{app}_{2q}(SWAP, q_1, q_2, 2) &\equiv q_1.
\end{aligned}$$

The cancellation rules say that applying two adjacent CX gates on the same pair of qubits will not change the state:

$$\begin{aligned}
&\texttt{app}_{2q}(CX, \texttt{app}_{2q}(CX, q_1, q_2, 1), \texttt{app}_{2q}(CX, q_1, q_2, 2), 1) \equiv q_1 \\
&\texttt{app}_{2q}(CX, \texttt{app}_{2q}(CX, q_1, q_2, 1), \texttt{app}_{2q}(CX, q_1, q_2, 2), 2) \equiv q_2.
\end{aligned}$$

These rewrite rules are defined as Z3 `assertions` about the symbolic functions $\texttt{app}_{1q}$ and $\texttt{app}_{2q}$, and are added into the precondition list of the proof obligation.

Soundness proofs. The symbolic execution and rewrite rules for quantum circuits are treated as axioms in Giallar and are proven using the denotational semantics defined using the QWire matrix library [35] within the Coq proof assistant [5]. The soundness of symbolic execution of quantum circuits can be trivially verified by showing that the output state always has the same denotational semantics with the input state. Since all rewrite rules only deal with a small

number of gates, their soundness can also be proven easily using the matrix representation. We also prove in Coq that the equivalence guarantee of the rewrite rules on a subset of qubits can be extended to the global circuit with an arbitrary number of qubits. Again, although these soundness proofs are developed manually in Coq, they only need to be done once.

6 The Giallar Verifier

The Giallar verifier generates proof obligations for compiler passes as SMT problems and invokes Z3 to solve them with the rewrite rules.

Proof obligations for a compiler pass. Giallar does not require the user to explicitly provide the specifications to the compiler pass, which is usually required by other automated verification frameworks such as Yggdrasil [44] and Serval [32]. Instead, Giallar pre-defines the proof obligations for all seven types of quantum compiler passes. Aside from routing passes, the other six types of compiler passes share the same specification that the input and output circuits are equivalent, defined using the following virtual class:

```
class GeneralPass():
    @classmethod
    def test(cls):
        optimizer = cls()
        init_circ = QCircuit()
        out_circ = optimizer.run(init_circ)
        assert(equivalent(out_circ, init_circ))
        print(cls.__name__ + " verified")
```

In the above `test()` method, Giallar first generates a symbolic circuit `init_circ` to represent the input quantum circuit, then symbolically executes the pass implementation through `optimizer.run` to get the symbolic representation of the output circuit, and finally attempts to verify that these two circuits are equivalent.

To enable the automated generation of proof obligations, Giallar requires the user to retrofit the compiler passes (in the above six types) with the `GeneralPass` virtual class as the parent class. For example, the class definition for the CXCancellation pass is shown as follows:

```
class CXCancellationPass(GeneralPass):
    #implementation omitted
    #...
```

Different from the other six types, the routing passes may insert swap gates to make the circuit satisfy the qubit connectivity constraint (see Section 2.3), such that the output circuit may not be strictly equivalent to the input circuit. However, the output circuit is equivalent to the input circuit up to a permutation that represents all inserted swaps. Giallar provides a `RoutingPass` virtual class for users to implement routing passes, whose proof obligations will then be automatically generated by Giallar.

Verification for a compiler pass. We will use the CXCancellation pass (see Section 3) as a running example to show how the Giallar verifier works. This CXCancellation pass is an optimization pass and contains an unbounded loop maintaining two circuit variables `output` and `remain`. The variable `output` contains gates that have been scanned, while `remain` contains gates that have not been scanned yet. Each loop iteration will attempt to find two CNOT gates with the same pair of input qubits in `remain` and cancel them out.

The CXCancellation pass inherits the `GeneralPass` virtual class and the proof obligation generated by Giallar is to show that the input and output circuits are equivalent, which can be derived using the `while_gate_remaining` loop template. Giallar will then attempt to solve the following subgoal generated for the loop body:

$$\mathbf{P_1}: \quad \texttt{app}\,(\texttt{output}_{old}; \texttt{remain}_{old}, Q) \equiv \texttt{app}\,(\texttt{input}, Q)$$
$$\mathbf{G_1}: \quad \texttt{app}\,(\texttt{output}_{new}; \texttt{remain}_{new}, Q) \equiv \texttt{app}\,(\texttt{input}, Q)\,,$$

where the new symbolic values for variables `output` and `remain` are produced by the loop body from the old ones. To generate the relation between the old and new variables, the Giallar verifier uses Z3Py to symbolically execute the loop body, getting the following preconditions:

$$\mathbf{P_2}: \quad \texttt{output}_{new} = \texttt{output}_{old}$$
$$\mathbf{P_3}: \quad \texttt{remain}_{old}\texttt{[0]} = CX$$
$$\mathbf{P_4}: \quad \texttt{remain}_{old}\texttt{[x]} = CX$$
$$\mathbf{P_5}: \quad \texttt{remain}_{new} = \texttt{remain}_{old}\texttt{[1:x]}; \texttt{remain}_{old}\texttt{[x+1:]},$$

where `x` is the return value of `next_gate(remain, 0)`. For convenience, we use C_1 and C_2 to represent $\texttt{remain}_{old}\texttt{[1:x]}$ and $\texttt{remain}_{old}\texttt{[x+1:]}$ respectively, and thus we have

$$\texttt{remain}_{old} = CX; C_1; CX; C_2$$
$$\texttt{remain}_{new} = C_1; C_2.$$

Using $\mathbf{P_1} \sim \mathbf{P_5}$, we can rewrite the proof goal $\mathbf{G_1}$ as:

$$\mathbf{G_2}: \quad \texttt{app}\,(CX; C_1; CX; C_2, Q') \equiv \texttt{app}\,(C_1; C_2, Q')\,,$$

where Q' is the quantum state after applying $\texttt{output}_{old}$ or $\texttt{output}_{new}$ to Q. Giallar then performs symbolic execution on both sides of $\mathbf{G_2}$, shown as follows:

$$\begin{aligned}\textbf{Left:}\quad & \texttt{app}(CX; C_1; CX; C_2, Q') \\ & \rightharpoonup \texttt{app}\,(C_1; CX; C_2, \texttt{app}(CX, Q')) \\ & \ldots \\ & \rightharpoonup \texttt{app}\,(C_2, \texttt{app}(CX, \texttt{app}(C_1, \texttt{app}(CX, Q'))))\end{aligned}$$

$$\textbf{Right:}\quad \texttt{app}(C_1; C_2, Q') \rightharpoonup \texttt{app}(C_2, \texttt{app}(C_1, Q')).$$

Thus, after removing the same application of C_2 on both sides of $\mathbf{G_2}$, the proof goal becomes:

$$\mathbf{G_3}: \quad \texttt{app}\,(CX, \texttt{app}(C_1, \texttt{app}(CX, Q'))) \equiv \texttt{app}(C_1, Q').$$

Note that the specification of `next_gate(0)` also adds the following precondition that the gate `x` can be reordered to

the second gate of $\texttt{remain}_{old}$:

$$\mathbf{P_6}: \quad \forall Q_1, \texttt{app}\,(CX, \texttt{app}(C_1, Q_1)) \equiv \texttt{app}\,(C_1, \texttt{app}(CX, Q_1))\,.$$

Besides, Giallar's rewrite rule set contains a cancellation rule to cancel out two adjacent CX gates, which is introduced as the following precondition to the proof goal:

$$\mathbf{P_7}: \quad \forall Q_2,\ \texttt{app}\,(CX, \texttt{app}(CX, Q_2)) \equiv Q_2$$

The Giallar verifier finally encodes the preconditions and goals into the following formula and invokes Z3 to solve:

$$\mathbf{P_6} \wedge \mathbf{P_7} \wedge \neg \mathbf{G_3},$$

which is expanded into qubit-local formulas containing $\texttt{app}_{2q}$ of symbolic representation for quantum circuits (see Section 5). If the above formula is satisfiable, the compiler pass is incorrect and a counter-example is generated by the verifier. Otherwise when Z3 proves the above formula is unsatisfiable, we successfully verify that the compiler pass correctly preserves the semantics using Giallar.

The above formula is unsatisfiable, i.e., $\mathbf{P_6}$ and $\mathbf{P_7}$ imply $\mathbf{G_3}$, and can be proven using Z3. We show why $\mathbf{P_6}$ and $\mathbf{P_7}$ imply $\mathbf{G_3}$ as follows. By instantiating Q_1 with $\texttt{app}(CX, Q')$ and rewriting the left side of $\mathbf{G_3}$ using $\mathbf{P_6}$, the proof goal becomes:

$$\mathbf{G_4}: \quad \texttt{app}\,(C_1, \texttt{app}(CX, \texttt{app}(CX, Q'))) \equiv \texttt{app}(C_1, Q').$$

After removing the same application of C_1 on both sides of $\mathbf{G_4}$, the proof goal becomes:

$$\mathbf{G_5}: \quad \texttt{app}\,(CX, \texttt{app}(CX, Q')) \equiv Q',$$

which can be directly proven using the cancellation rule $\mathbf{P_7}$.

Requirements of the Giallar verifier. For a pass to be verified using Giallar, it needs to be written in a way that 1) each loop in the pass follows one of the three loop patterns introduced in Section 4; 2) it uses Giallar's verified library to represent quantum programs; and 3) the transformation of quantum gates must be expressible using the given rewrite rules. Note that new loop templates, verified library functions, and rewrite rules can be easily added.

7 Case Studies

In this section, we will present three case studies to show how we use Giallar to discover quantum-specific bugs during the push-button verification of the Qiskit compiler.

7.1 The `optimize_1q_gate` Pass

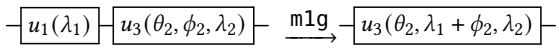
We first focus on the verification of the `optimize_1q_gate` pass and show that, using Giallar, we can reveal bugs that only arise in quantum software.

The `optimize_1q_gate` pass invokes the utility function `merge_1q_gate` to collapse a chain of 1-qubit gates into a single, more efficient gate [25], to mitigate noise accumulation. It operates on u_1, u_2, and u_3 gates, which are native gates in the IBM quantum devices. These gates can be naturally described as linear operations on the Bloch sphere; for example, u_1 gates are rotations with respect to the Z axis. For clarity, we list their matrix representations in Table 1.

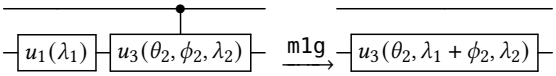
Table 1. Matrix representation of physical gates u_1, u_2 and u_3, where u_1 is a Z rotation on the Bloch sphere.

$$u_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}, \quad u_2(\phi, \lambda) = \frac{\sqrt{2}}{2} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\lambda+\phi)} \end{pmatrix}$$

$$u_3(\theta, \phi, \lambda) = \frac{\sqrt{2}}{2} \begin{pmatrix} \cos(\theta) & -e^{i\lambda}\sin(\theta) \\ e^{i\phi}\sin(\theta) & e^{i(\lambda+\phi)}\cos(\theta) \end{pmatrix}$$



(a) A correct merge of gates u_1 and u_3. The cuicuit is equivalent before and after merging.



(b) An incorrect merge. The new curcuit is not equivalent because u_3 was a controlled gate before merging.

Figure 8. Correct execution (top) and incorrect execution (bottom) of `merge_1q_gate`.

The `optimize_1q_gate` pass has two function calls. First, it calls the `collect_runs` method to collect groups of consecutive u_1, u_2, and u_3 gates. Then it calls `merge_1q_gate` to merge the gates in each group. The `merge_1q_gate` method (see Fig. 8a) first transforms the 1-qubit gates from the Bloch sphere representation to the unit quaternion representation [9] and then applies the `merge()` function to that representation to merge the rotations.

In Qiskit, all gates can be modified with a `c_if` or `q_if` method to condition its execution on the state of other classical or quantum bits. When proving that `merge()` does not change semantics of the quantum program, we found that in some cases the compiler pass attempts to optimize these gates without noticing that the gate is controlled by other (qu)bits, leading to an incorrect execution as in Figure 8b. For this reason, in our retrofitted implementation of this pass, we inserted checks that `gate1.q_if == False` and `gate1.c_if == False` before merging the 1-qubit gates.

Bugs similar to the one described above, which relates to how quantum circuit instructions can be conditioned, have been observed in Qiskit in the past [37, 39]. In the absence of the rigorous verification provided by tools like Giallar, such bugs are hard to discover. In practice, this is usually done via extensive randomized testing of input and output circuits, which does not provide any guarantee of finding faulty code. The results of `merge_1q_gate` here demonstrate that Giallar is effective for detecting quantum-related bugs.

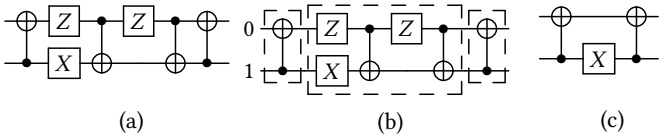


Figure 9. A working example of `commutation_analysis` and `commutative_cancellation`. (a) The un-optimized circuit, (b) `commutation_analysis` forming the commutation groups, and (c) `commutative_cancellation` cancels out self-inverse gates within each group.

7.2 The `commutation` Passes

The second bug found are in the `commutation_analysis` and `commutative_cancellation` passes which optimize Qiskit DAGCircuits using the quantum commutation rules and the cancellation rules pictured in Figure 7. First, `commutation_analysis` transforms the quantum circuit to a representation called commutation groups [42], where nearby gates that commute with each other are grouped together. Next, `commutative_cancellation` performs cancellation inside the newly formed groups. We give a working example in Figure 9. The circuit in Figure 9a is first partitioned into three parts such that all gates in one part commute with each other. For example, in the middle part in Figure 9b, the Z gate on qubit 0 commutes with the following $CNOT$ on qubits 0, 1, and the X gate on qubit 1 also commutes with $CNOT$ 0, 1. These commutativity relations can be verified in Coq using matrix semantics and included in the rewrite rules as shown in Figure 7.

A bug was discovered when we attempted to verify these two passes. In the original implementation, gates in one group are not guaranteed to be pairwise commutative. We found this violation comes from the fact that the commutation relation is in general not transitive. For example, denoting the commutation relation as $\sim$, if there are three quantum gates, A, B, C where $A \sim B$ and $B \sim C$, then $A \sim C$ is not guaranteed to be true. For this reason, gates that do not commute will be grouped together in the flawed version of this pass. In our fixed version, we make sure that the circuits compiled by these passes have a limited gate set where $\sim$ is indeed transitive. For example, in the gate set {CX, X, Z, H, T, u_1, u_2, u_3}, $\sim$ is transitive.

7.3 Termination in `routing` Passes

For passes with loops, besides proving that the output circuit is equivalent to the input circuit, we also need to prove that the pass will terminate. Among our three loop templates, `iterate_all_gates` and `collect_runs` are range-based for-loops and always terminate. The `while_gates_remaining` template is a while-loop and may not terminate. For the `while_gates_remaining` template, Giallar will generate a subgoal to prove that after each iteration, the number of remaining gates strictly decreases, i.e., at least one gate in the remaining gate list is removed when executing the loop body. Most of the passes always call the `QCircuit.delete()` method at least once in the loop body, and thus they always terminate. However, in one pass, namely the `look_ahead` routing pass, Giallar fails to prove its termination.

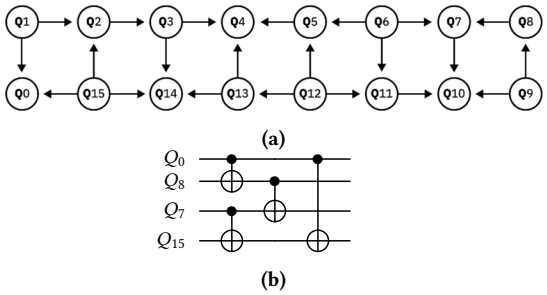


Figure 10. (a) The coupling map of the IBM 16 qubit device. Arrows indicate available CNOT directions (which does not affect the swap insertion step). (b) A counter-example generated by Giallar that shows Qiskit's `lookahead_swap` pass does not always terminate on the IBM 16 qubit device.

After closer inspection of the pass, we found a counter-example circuit on the coupling map of the IBM 16 qubit device, for which the `lookahead_swap` pass does not terminate (see Figure 10). As shown in the coupling map of the IBM 16 qubit device (see Figure 10a), a 2-qubit gate may not be available for two physical qubits. The `lookahead_swap` pass is designed to insert swap gates such that, for every 2-qubit gate, its two logical qubits are mapped to physical qubits allowing that gate. The previous `lookahead_swap` pass implementation repeatly finds one `swap` gate to minimize the total distance of all 2-qubit gates at each iteration until all gates are mapped. In the counter-example, if the four logical qubits is mapped to physical qubits Q_0, Q_8, Q_7 and Q_{15} (see Figure 10b), then the total distance will not change by inserting any single swap gate. In this special case, the pass implementation will insert a swap gate between Q_1 and Q_2. However, the total distance of the resulting map still will not change by inserting any single swap gate. Thus, another swap gate will be inserted between Q_1 and Q_2. By applying the swap rules as in Figure 7, these two swap gates will cancel out and the `lookahead_swap` pass will restart from the initial state and will not terminate.

We fixed the bug by inserting the swap gate at a *random* location when the total distance will not change by inserting any single swap gate. The randomization introduced by this fix will avoid keeping inserting swap gates at the same location that can cancel out. Because the choice of `swap` gates does not affect the correctness of the pass, such randomness does not prevent the correctness verification. We proved the correctness of this randomized version by modeling random numbers as arbitrary symbolic numbers.

8 Evaluation

To demonstrate its effectiveness at verifying quantum compiler passes, we implemented and evaluated Giallar on 56 compiler passes in 13 versions (from v0.19 to v0.32) of the Qiskit compiler that are listed on the Qiskit website [36]. The implementation consists of 3.6k lines of code for the Giallar framework, including quantum circuit-related data structures, loop templates, virtual classes, the Giallar preprocessor, and the Qiskit wrapper, and 168 lines of code for utility functions.

Using Giallar, we have successfully verified 44 out of the 56 compiler passes. Among all 12 passes that Giallar fails to verify, eight passes are scheduling passes that deal with pulse instructions which are at a lower abstraction level than quantum gates. Same as the previous verification frameworks for quantum computing [1, 15], Giallar only supports the abstraction at the quantum gate level and cannot reason about pulse-level behaviors. The other four passes that we do not verify are *StochasticSwap*, *CrosstalkAdaptiveSchedule*, *BIPMapping*, and *UnitarySynthesis* passes. *StochasticSwap* uses a randomized routing algorithm that Giallar does not yet support. *CrosstalkAdaptiveSchedule* and *BIPMapping* are passes that invoke Z3 and CPLEX solvers respectively to find the output circuit. Giallar does not have formal semantics for solvers and therefore cannot model these two passes. *UnitarySynthesis* is a synthesis pass that produces an approximated circuit when the exact circuit includes gates that cannot be performed on the given quantum hardware. This cannot be verified without a proper way to specify and reason about the error bound of the approximated compilation. Note that all the above failed passes are also costly or infeasible to manually verify using previous verification frameworks for quantum compilers [1, 15].

Experiment setup. Our evaluation only focuses on the 44 verified passes. We have evaluated the performance of verifying these 44 passes, as well as running verified passes to compile real quantum circuits compared with the unverified Qiskit passes. The system ran with Python 3.8.7 and Z3 4.8.12. All verification and compilation tasks ran on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM and a 512GB Intel SSD Pro 600p.

Verification performance. Table 2 gives the result of the verification for all 44 Qiskit passes. The verification of all passes completed in less than 30 seconds.

Table 2 also lists the number of subgoals to be proved after preprocessing each pass. We can see that even if theoretically there may be an exponential number of subgoals when there are many branch statements, there are at most eight subgoals for all Qiskit passes.

Compiler performance. We have also evaluated our verified compiler and the original unverified Qiskit compiler on a series of quantum circuits from QASMBench [20]. The benchmark includes 48 quantum circuits with various near-term quantum applications, including quantum state preparation (cat_state, bell, ghz_state), quantum arithmetic (adder), quantum chemistry simulation (ising), quantum machine learning (dnn), and other famous quantum algorithms (deutsch, qft,

Table 2. Verification result of the 44 verified passes in Qiskit.

Pass name	Pass LOC	#sub-goals	Verif. time(s)
ApplyLayout	11	2	0.7
SetLayout	8	1	0.7
TrivialLayout	10	1	0.7
Layout2qDistance	19	1	0.7
DenseLayout	77	1	0.7
NoiseAdaptiveLayout	192	1	0.7
SabreLayout	62	1	0.7
CSPLayout	52	1	0.7
EnlargeWithAncilla	8	1	0.8
FullAncillaAllocation	8	1	0.7
BasicSwap	36	4	2.4
LookaheadSwap	100	3	3.5
SabreSwap	96	3	3.8
Unroller	23	3	1.5
Unroll3qOrMore	23	3	1.4
Decompose	23	3	2.0
UnrollCustomDefinitions	22	3	1.5
BasisTranslator	119	3	1.5
Optimize1qGates	32	3	25.1
Optimize1qGatesDecomp	32	3	25.2
Collect2qBlocks	9	1	0.7
ConsolidateBlocks	19	3	1.4
CXCancellation	24	4	4.2
CommutationAnalysis	6	1	0.7
CommutativeCancellation	17	3	1.3
RemoveDiagBeforeMeasure	24	3	18.1
RemoveReseatInZeroState	16	3	1.3
Width	8	1	0.6
Depth	8	1	0.6
Size	9	1	0.6
CountOps	8	1	0.7
CoutOpsLongestPath	8	1	0.7
NumTensorFactors	8	1	0.7
DAGLongestPath	8	1	0.9
CheckMap	19	1	0.7
CheckCXDirection	19	1	0.7
CheckGateDirection	19	1	0.7
CXDirection	29	4	2.3
GateDirection	55	8	5.6
MergeAdjacentBarriers	24	4	4.1
BarrierBeforeFinalMeasure	22	4	19.7
RemoveFinalMeasure	20	3	2.9
DAGFixedPoint	17	1	0.6
FixedPoint	17	1	0.6
Sum	1,366	95	145.4

grover, qaoa). These quantum circuits contain up to 27 qubits and up to 5,000 quantum gates.

We ran all 48 circuits in the benchmark using the (most computationally expensive) lookahead swap pass for the Qiskit implementation and the Giallar implementation. Figure 11 summarizes the running time of all 31 benchmark circuits that Qiskit succeeded. We can see that Giallar successfully compiled all these 31 circuits as well. For smaller circuits, the performance overhead of Giallar was at most 0.5 seconds, while for larger circuits, the performance overhead was at most 10%. The overhead mainly came from loading the verified library in Python, the data structure conversion between Qiskit DAG and Giallar list representations, and the Qiskit wrapper. The results show that the formal correctness guarantee of Giallar introduces only a modest compilation performance overhead.

Reusability. Most of Giallar's rewrite rules and utility functions are shared across passes. Among all three classes of rewrite rules shown in Figure 7, the cancellation rules are used by optimization passes including CommutativeCancellation, CXCancellation, Optimize1qGates, and ConsolidateBlocks. The commutativity rules are used in the CommutativeAnalysis and CommutitiveCancellation passes. The swap rules are used in all routing passes. The utility functions in Giallar include: 1) the circuit manipulating operations (e.g., `next_gate`) which are used among all passes; 2) the coupling map related operations (e.g., `shortest_path`) that are used in all routing passes; and 3) the circuit merging and gate expanding operations `merge` which are used in the ConsolidateBlocks and Optimize1qGate passes.

Adding new passes. Our experience shows that many new passes introduced can be verified automatically. Giallar was first developed for Qiskit 0.19 (see our technical report [43]). When we applied Giallar to verify the 16 new passes introduced by Qiskit 0.32, 15 out of 16 were verified automatically. The failed one uses an ECR gate that Giallar did not model. We added the symbolic execution and the corresponding rewrite rule for this new gate feature to enable its verification.

Limitations. Compared with the manual verification framework [15], Giallar has a larger trusted computing base, including the Giallar implementation, the Z3 SMT solver, the Coq proof checker, the symbolic execution of Z3Py, and the equivalence of data structures defined in Python, Coq and Z3. We note that such a larger trusted computing base is common in prior work on push-button verification [32, 33].

Giallar matches loops against a pre-defined set of loop templates. While applicable to the current Qiskit compiler, Giallar may fail to handle future compiler versions with new patterns of loops. We would like to explore how to integrate previous work on loop invariant learning [41, 47] to remove the loop templates in the future.

Giallar's rewrite rule set is incomplete and may not be able to prove the equivalence for some future passes. The symbolic execution for quantum circuits is also incomplete and does not model some gate features. New rewrite rules and more gate support may be needed to support future Qiskit passes. Besides, the rewrite rules for quantum circuits in Giallar only supports local equivalence of quantum states. It does not support non-local quantum circuit optimizations that are implemented using other quantum state representations, such as the phase polynomial representation [1, 2, 30] and the state vector quantum assertion [14].

Besides, Giallar is only designed to prove that the compiled quantum circuits preserve the semantics of the input circuits and satisfy the topological constraints given by the coupling map. Non-critical statements and function invocations which do not affect the generated circuits will be discarded during the preprocessing, such that the correctness of these statements and the generated debugging information is not verified.

9 Related Work

Verified quantum compilation. Following the approach of verifying classical compilation [11, 12, 19], previous efforts on verifying quantum compilation mainly rely on interactive theorem provers to construct manual proofs. ReVerC [2] is a verified compiler for reversible circuits, a subset of quantum circuits that are easier to reason about. The verification is done in F^* [26]. ReQWire [40] verifies ancillae uncomputation, one specific step of quantum compilation, using Coq [5]. VOQC [15] verifies several quantum optimization passes such as circuit mapping and gate cancellation using Coq. Compared with Giallar, these frameworks have the benefit of a smaller trusted computing base, and may support more complicated compilation passes whose proof goals are not provable by an SMT solver. However, these frameworks require both formal verification expertise and a significant proof burden. Moreover, all these works only verified one static version of their passes, and their proofs cannot adapt to the fast changing real-world quantum compilers such as Qiskit. In contrast, Giallar is a push-button verification toolkit, allowing developers without much formal verification background to write and verify compiler passes. With Giallar, newly developed or modified passes can be easily introduced and automatically verified.

Equivalence checking for quantum circuits. Algorithms to perform efficient equivalence checking for quantum circuits have been discussed from the view of quantum algorithms [46], quantum communication protocols [3], and verification of compilation [1]. Given two concrete quantum circuits C_1 and C_2, these checking algorithms can answer if C_1 and C_2 are equivalent. However, to verify a quantum compiler, we must prove that for *any* quantum circuit C, the compiled circuit C' is equivalent to C. This is beyond

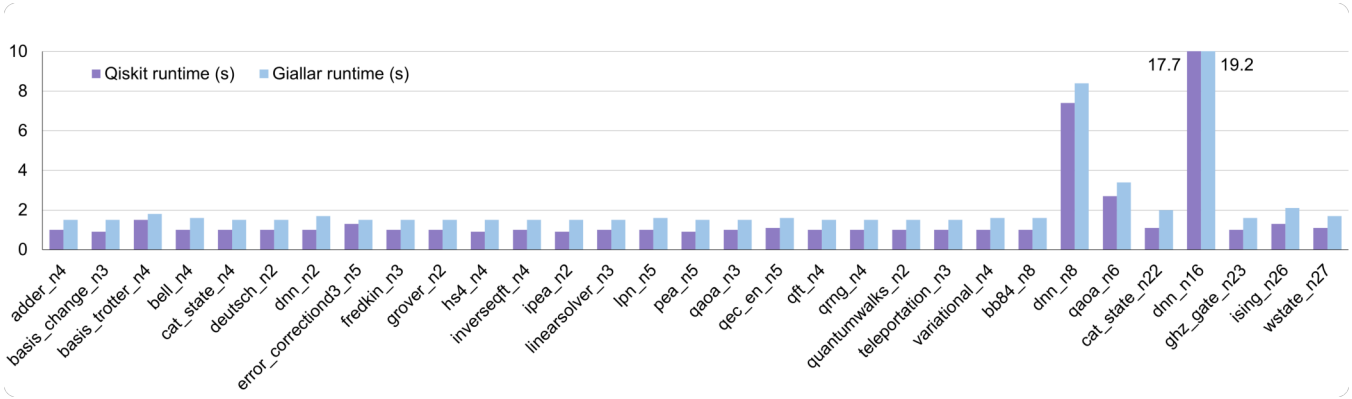


Figure 11. Comparison of the compilation performance of Qiskit and Giallar on QASMBench.

reach for existing checking algorithms, but can be efficiently handled by Giallar's rewrite rules.

Verified quantum programs. There are several quantum programming environments that support verifying the correctness of quantum programs such as QWire [35], $Q\,|SI\rangle$ [22] and QBricks [4], or verifying the error bounds such as the logic of quantum robustness [17] and Gleipnir [45]. These frameworks are built for verifying quantum programs rather than for quantum compilers that transform quantum programs but are themselves classical programs.

Automatic translation validation. In classical computing, there is also a long line of work of automatic translation validation [31], which verifies the semantic equivalence of given source and target programs in a specific translation. Automatic translation validation can be used to provide guarantee to a complicated compiler that is hard to directly verify, even when complex loops are involved [7, 13]. However, it cannot guarantee the correctness of the compiler for *all* source programs and has to be run at each compilation. In contrast, Giallar verifies the correctness of the Qiskit compiler and the verification was done once and for all.

Push-button verification in classical computing. Push-button verification has been applied to building verified compilers, file systems, OS kernels, and system monitors [23, 32, 33, 44]. Many of our technical designs and verification ideas are heavily influenced by these previous works. However, these verification frameworks cannot support unbounded loops, nor are they directly applicable to quantum circuits.

10 Conclusion

Giallar is a push-button verification toolkit that, for the first time, enables automated verification for real-world quantum compiler passes. To verify unbounded loops, Giallar introduces loop templates whose loop invariants can be automatically generated. Giallar introduces the quantum circuit rewrite rules to efficiently check the equivalence of quantum circuits, and provides a verified library for writing new compiler passes and modularizing their proofs. Using Giallar, we have successfully verified 44 (out of 56) passes of Qiskit, the most widely used Quantum compiler, and detected three crucial bugs (two are quantum-specific) in the original Qiskit implementation. The approach we establish with Giallar paves the way for end-to-end verification of a complete quantum software toolchain, an important step towards practical near-term quantum computing.

Acknowledgments

We thank our shepherd, Sorav Bansal, and the anonymous reviewers for valuable feedbacks that help improving this paper. We thank Bryce Monier and John Zhuang Hui for conducting parts of the experiment and providing helpful comments on earlier drafts. This work is funded in part by three Amazon Research Awards, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080; in part by EPiQC, an NSF Expedition in Computing, under grants CCF-1730082/1730449; in part by STAQ under grant NSF Phy-1818914; in part by NSF Grant No. 2110860; by the US Department of Energy Office of Advanced Scientific Computing Research, Accelerated Research for Quantum Computing Program; and in part by NSF OMA-2016136 and the Q-NEXT DOE NQI Center. Ronghui Gu is the Founder of and has an equity interest in CertiK. Frederic T. Chong is Chief Scientist at Super.tech and an advisor to Quantum Circuits, Inc.

References

[1] Matthew Amy. 2019. Towards Large-scale Functional Verification of Universal Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan 2019), 1–21. https://doi.org/10.4204/EPTCS.287.1

[2] Matthew Amy, Martin Roetteler, and Krysta M Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17)*. 3–21. https://doi.org/10.1007/978-3-319-63390-9_1

[3] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. 2011. *Equivalent Quantum Circuits.* Technical Report. Universidad de Valladolid, Dpto. Teoria de la Senal e Ing. arXiv:1110.2998v1

[4] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs.. In *Proceedings of the 30th European Symposium on Programming (ESOP '21)*. 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

[5] The Coq Development Team. 2012. *The Coq Reference Manual, version 8.4.* Available electronically at http://coq.inria.fr/doc.

[6] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. arXiv:1707.03429

[7] Manjeet Dahiya and Sorav Bansal. 2017. Black-box equivalence checking across compiler optimizations. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS '17)*. 127–147. https://doi.org/10.1007/978-3-319-71237-6_7

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, Vol. 4963. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[9] Philippe Gille and Tamas Szamuely. 2009. *Central Simple Algebras and Galois Cohomology.* Cambridge University Press. https://doi.org/10.1017/cbo9780511607219

[10] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going Beyond Bell's Theorem.* Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

[11] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. 2015. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 595–608. https://doi.org/10.1145/2775051.2676975

[12] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. 646–661. https://doi.org/10.1145/3296979.3192381

[13] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428289

[14] Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2018. Using Hoare Logic for Quantum Circuit Optimization. arXiv:1810.00375

[15] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (2021). https://doi.org/10.1145/3434318

[16] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[17] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative robustness analysis of quantum programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290344

[18] Chris Lattner and Vikram Adve. 2003. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* Technical Report. UIUC. http://llvm.cs.uiuc.edu/

[19] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[20] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2021. QASMBench: A Low-level QASM Benchmark Suite for NISQ Evaluation and Simulation. arXiv:2005.13018

[21] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the Qubit Mapping Problem for NISQ-era Quantum Devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 1001–1014. https://doi.org/10.1145/3297858.3304023

[22] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. 2018. $Q|SI\rangle$: A Quantum Programming Environment. In *Lecture Notes in Computer Science*. Vol. 11180. Springer, 133–164. https://doi.org/10.1007/978-3-030-01461-2_8

[23] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 22–32. https://doi.org/10.1145/2737924.2737965

[24] Dmitri Maslov, Gerhard W. Dueck, Michael Miller, and Camille Negrevergne. 2008. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 3 (2008), 436–444. https://doi.org/10.1109/TCAD.2007.911334

[25] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. 2018. Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments. arXiv:1809.03452

[26] Microsoft. 2016. F*. https://github.com/FStarLang/FStar/

[27] Frank Mueller, Greg Byrd, and Patrick Dreher. 2020. Programming Quantum Computers: A Primer with IBM Q and D-Wave Exercises. https://sites.google.com/ncsu.edu/qc-tutorial

[28] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASLPOS '19)*. 1015–1029. https://doi.org/10.1145/3297858.3304075

[29] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 1001–1016. https://doi.org/10.1145/3373376.3378477

[30] Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. 2018. Automated Optimization of Large Quantum Circuits with Continuous Parameters. *npj Quantum Information* 4, 1 (2018), 1–12.

[31] George C Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the 21st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. 83–94. https://doi.org/10.1145/349299.349314

[32] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 225–242. https://doi.org/10.1145/3341301.3359641

[33] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 252–269. https://doi.org/10.1145/3132747.3132748

[34] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

[35] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (PLDI '17)*. 846–858. https://doi.org/10.1145/3009837.3009894

[36] Qiskit 0.32.0 Documentation 2021. Available at https://qiskit.org/documentation/index.htm.

[37] Qiskit Bug Report 2019. Available at https://github.com/Qiskit/qiskit-terra/issues/1871.

[38] Qiskit Github repository 2021. Available at https://github.com/Qiskit/qiskit-terra.

[39] Qiskit Terra Github issue page 2018. Available at https://github.com/Qiskit/qiskit-terra/issues.

[40] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. arXiv:1901.10118

[41] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *Proceedings of the 8th International Conference on Learning Representations (ICLR '20)*. https://openreview.net/forum?id=HJlfuTEtvB

[42] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. 2019. Optimized Compilation of Aggregated Instructions for Realistic Quantum Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 1031–1044. https://doi.org/10.1145/3297858.3304018

[43] Yunong Shi, Runzhou Tao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. 2019. *CertiQ: A Mostly-automated Verification of a Realistic Quantum Compiler*. Technical Report. arXiv:1908.08963

[44] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

[45] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T Chong, and Ronghui Gu. 2021. Gleipnir: toward practical error analysis for Quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. 48–64. https://doi.org/10.1145/3453483.3454029

[46] George F. Viamontes, Igor L. Markov, and John P. Hayes. 2007. Checking Equivalence of Quantum Circuits and States. In *Proceedings of the 2007 International Conference on Computer-Aided Design (ICCAD '07)*. 69–74. https://doi.org/10.1109/ICCAD.2007.4397246

[47] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. 106–120. https://doi.org/10.1145/3385412.3385986

[48] Vladimir Zamdzhiev. 2016. Quantum Computing: the Good, the bad, and the (not so) Ugly! Invited Talk at University of Oxford.

[49] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. 2021. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. 1373–1376. https://doi.org/10.1109/ASE51524.2021.9678908