The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation

CHARLES YUAN, MIT CSAIL, USA
MICHAEL CARBIN, MIT CSAIL, USA

Numerous quantum algorithms require the use of quantum error correction to overcome the intrinsic unreliability of physical qubits. However, quantum error correction imposes a unique performance bottleneck, known as T-complexity, that can make an implementation of an algorithm as a quantum program run more slowly than on idealized hardware. In this work, we identify that programming abstractions for control flow, such as the quantum `if`-statement, can introduce polynomial increases in the T-complexity of a program. If not mitigated, this slowdown can diminish the computational advantage of a quantum algorithm.

To enable reasoning about the costs of control flow, we present a cost model that a developer can use to accurately analyze the T-complexity of a program under quantum error correction and pinpoint the sources of slowdown. To enable the mitigation of these costs, we present a set of program-level optimizations that a developer can use to rewrite a program to reduce its T-complexity, predict the T-complexity of the optimized program using the cost model, and then compile it to an efficient circuit via a straightforward strategy.

We implement the program-level optimizations in Spire, an extension of the Tower quantum compiler. Using a set of 11 benchmark programs that use control flow, we empirically show that the cost model is accurate, and that Spire's optimizations recover programs that are asymptotically efficient, meaning their runtime T-complexity under error correction is equal to their time complexity on idealized hardware.

Our results show that optimizing a program before it is compiled to a circuit can yield better results than compiling the program to an inefficient circuit and then invoking a quantum circuit optimizer found in prior work. For our benchmarks, only 2 of 8 tested quantum circuit optimizers recover circuits with asymptotically efficient T-complexity. Compared to these 2 optimizers, Spire uses 54×–2400× less compile time.

CCS Concepts: • **Computer systems organization** → **Quantum computing**.

Additional Key Words and Phrases: quantum programming languages, quantum compilers

ACM Reference Format:
Charles Yuan and Michael Carbin. 2024. The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation. *Proc. ACM Program. Lang.* 8, PLDI, Article 167 (June 2024), 26 pages. https://doi.org/10.1145/3656397

1 INTRODUCTION

Quantum algorithms promise computational advantage over classical algorithms across numerous domains, including cryptography and communication [Bennett and Brassard 2014; Bennett et al. 1993; Proos and Zalka 2003; Shor 1997], search and optimization [Farhi et al. 2014; Grover 1996], data analysis and machine learning [Biamonte et al. 2017; Lloyd et al. 2014; Rebentrost et al. 2018], and physical simulation [Abrams and Lloyd 1997; Babbush et al. 2018; Childs et al. 2018].

The power of quantum algorithms is rooted in their ability to manipulate quantum information, which exists in a *superposition* of weighted classical states. A quantum computer may use *quantum*

Authors' addresses: Charles Yuan, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, chenhuiy@csail.mit.edu; Michael Carbin, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, mcarbin@csail.mit.edu.

ACM 2475-1421/2024/6-ART167
https://doi.org/10.1145/3656397

logic gates to modify the states and weights within a superposition, and *measure* quantum data to obtain a classical outcome with probability determined by the weights in the superposition.

A common representation of a quantum algorithm is as a *quantum circuit*, a sequence of quantum logic gates that operate over individual *qubits*, which are the quantum analogue of bits.

Error Correction. Whereas an idealized quantum computer can execute any quantum algorithm, a realistic device must contend with the fact that every known physical implementation of a qubit is unreliable, meaning its state becomes irreversibly corrupted after a small number of logic gates are performed. To execute the algorithms that possess a provable asymptotic advantage in time complexity, including Grover [1996]; Shor [1997], a quantum computer must employ *quantum error correction* to encode a reliable *logical* qubit within a number of unreliable physical qubits.

Resource Estimation. A quantum algorithm that executes more logic gates requires logical qubits to be more reliable, which in turn demands more physical qubits, a scarce hardware resource. Given an algorithm, it is thus essential to determine its time complexity in terms of the number of logic gates that it executes. This task, known as *resource estimation* [Hoefler et al. 2023; Leymann and Barzen 2020; Suchara et al. 2013], is key to recognizing the scale of hardware needed to execute a quantum algorithm and the problem size at which it offers advantage over classical algorithms.

T-Complexity Bottleneck. In principle, conducting resource estimation for a quantum algorithm involves simply writing it out as a circuit and counting the number of logic gates used. A practical challenge is that quantum error correction affects the available logic gates and their costs.

An idealized quantum computer supports any physically realizable quantum logic gate, including analogues of classical NAND gates known as *multiply-controlled NOT* (MCX) gates that are necessary for arithmetic [Gidney and Ekerå 2021; Rines and Chuang 2018] and memory [Low et al. 2018] within a quantum algorithm. By contrast, the prevailing *surface code* [Fowler et al. 2012] architecture for error correction that has been implemented in practice by quantum hardware from Google [Google Quantum AI 2023] and IBM [Takita et al. 2016] supports only a restricted set of gates known as the *Clifford+T* gates, into which all MCX gates must be decomposed.

In turn, the decomposition of MCX uses the single-qubit T gate, a performance bottleneck on the surface code. Unlike *Clifford* gates such as NOT and the two-qubit controlled-NOT (CNOT) that are natively supported by the code, the T gate is realized separately via *magic state distillation* [Bravyi and Kitaev 2005] at an area-latency cost[1] of about 10^2 times that of a CNOT gate [Gidney and Fowler 2019] and 10^{10} times that of a NAND gate in classical transistors [Babbush et al. 2021].

Although deriving an efficient quantum circuit requires navigating all of its gate costs, the order of magnitude increase in cost for T gates relative to other gates has contributed to a contemporary consensus that "The number of T gates … typically dominates the cost when implementing a fault tolerant algorithm" [Reiher et al. 2017]. It is therefore broad practice to quantify the runtime cost of a quantum algorithm under error correction using its *T-complexity* [Babbush et al. 2018], i.e. number of T gates, which is often greater than its number of MCX gates.

1.1 *T*-Complexity Costs of Control Flow in Quantum Programs

Resource estimation is made even more challenging by the reality that it is often impractical for a developer to explicitly write quantum circuits by hand. Instead, the developer uses *quantum programming languages* [Green et al. 2013; Paykin et al. 2017; Selinger 2004; Svore et al. 2018], which provide programming abstractions over quantum data that are ultimately compiled to circuits.

[1]By area-latency cost, we refer to the product of the number of qubits and the number of processing cycles of the device.

Control Flow. One abstraction provided by languages [Altenkirch and Grattage 2005; Bichsel et al. 2020; JavadiAbhari et al. 2014; Voichick et al. 2023; Ying et al. 2012; Yuan and Carbin 2022] is a quantum `if`-statement that conditions on the value of qubit in superposition. This concept of *control flow in superposition* enables algorithms for simulation [Babbush et al. 2018], factoring [Shor 1997], and search [Ambainis 2004] to be expressed as a program more concisely than without the abstraction. In turn, the language compiler produces a circuit for a program that utilizes quantum `if` by translating the abstraction into individual qubit-controlled logic gates.

Costs of Control Flow. The problem is that quantum error correction can make the use of control flow abstractions significantly more inefficient than on idealized hardware. In this work, we identify that the usage of programming abstractions for control flow in superposition, such as quantum `if`, in a program can cause its asymptotic T-complexity to be polynomially larger than the time complexity found by a standard analysis that assumes idealized hardware. This blowup arises because a quantum `if` compiles to significantly more T gates than it does to MCX gates.

In turn, a polynomial increase in T-complexity diminishes the theoretical advantage of quantum algorithms for tasks such as search [Brassard et al. 2002; Grover 1996] and optimization [Sanders et al. 2020] that have only polynomial advantage over classical algorithms. Moreover, emerging evidence suggests that "at least cubic or quartic speedups are required for a practical quantum advantage" [Hoefler et al. 2023], which implies that even a linear slowdown jeopardizes the practical advantage of an algorithm that is otherwise marginally over the cubic speedup threshold.

1.2 Cost Model for Accurately Predicting *T*-Complexity Costs

To optimize away this overhead, the developer must pinpoint the specific locations in a quantum program that incur the overhead. A challenge is that without the ability to accurately reason about the program at syntax level, the developer must repeatedly compile it to a large circuit and count its gates, which does not efficiently or precisely identify the cause of the slowdown.

As an alternative, we present a cost model for reasoning about the T-complexity of programming abstractions for control flow in superposition within a quantum program. Using the cost model, a developer can pinpoint the sources of slowdown through a syntax-level analysis that accurately determines the runtime cost of each program statement under quantum error correction.

1.3 Program-Level Optimizations for Mitigating *T*-Complexity Costs

Next, we present a set of *program-level optimizations* for quantum programs. Using them, a developer can rewrite a program to reduce its T-complexity, predict the T-complexity of the optimized program using the cost model, and then compile it to an efficient circuit by a straightforward strategy.

The first optimization, *conditional flattening*, identifies excess T gates caused by nested quantum `if`-statements, and removes these gates by introducing a temporary qubit and using it to flatten the structure of conditional statements. The second optimization, *conditional narrowing*, identifies excess T gates caused by statements that do not need to be placed under a quantum `if` and safely moves these statements outside the `if`, thereby narrowing the range of conditional statements.

We implement and evaluate these optimizations in Spire, an extension of the Tower [Yuan and Carbin 2022] quantum compiler. For a set of 11 benchmark programs that use control flow, Spire successfully recovers programs that are asymptotically efficient, meaning their T-complexity under error correction is equal to their time complexity on idealized hardware.

Our results show that optimizing a program before it is compiled to a circuit can yield better results than compiling the program to an inefficient circuit and then invoking a quantum circuit optimizer found in prior work. For our benchmarks, a majority of existing optimizers we tested do not recover circuits that are asymptotically efficient in T-complexity. Moreover, Spire's optimizations followed by an existing quantum circuit optimizer achieve better results than either approach alone.

Forms of conditional narrowing and flattening appear in prior work [Ittah et al. 2022; Seidel et al. 2022; Steiger et al. 2018]. Our novel contributions are to unify both optimizations as program rewrite rules, identify that they can mitigate the asymptotic slowdown caused by control flow, and empirically evaluate their effectiveness and speed relative to existing circuit optimizers.

1.4 Contributions

In this work, we present the following contributions:

- *Costs of Control Flow* (Section 3). We identify that programming abstractions for control flow in superposition can introduce polynomial overheads in the T-complexity of a program. These costs can diminish the advantage of a quantum algorithm under error correction.
- *Cost Model* (Section 5). We present a cost model that computes the T-complexity of a quantum program that utilizes control flow. Using the cost model, the developer can accurately analyze the runtime cost of a program under an error-corrected quantum architecture.
- *Program-Level Optimizations* (Section 6). We present two optimizations for quantum programs, *conditional flattening* and *conditional narrowing*. Using them, a developer can rewrite a program to reduce its T-complexity, predict the T-complexity of the optimized program using the cost model, and compile that program to an efficient circuit via a straightforward strategy.
- *Evaluation* (Sections 7 and 8). We implement the optimizations in Spire, an extension of the Tower quantum compiler. Using a set of 11 benchmark programs that contain control flow, we empirically show that the cost model is accurate, and that Spire's optimizations can mitigate the T-complexity costs of control flow and recover an asymptotically efficient program. By contrast, only 2 of 8 existing quantum circuit optimizers we tested recover circuits with asymptotically efficient T-complexity. Spire uses 54×–2400× less compile time than these 2 optimizers.

Implications. This work reveals challenges that must be overcome to fully realize the asymptotic advantage of quantum algorithms on an error-corrected quantum computer. By incorporating our cost model and optimizations, program optimizers may more precisely account for the architectural costs of error correction and the abstraction costs of control flow in a quantum program.

2 BACKGROUND ON QUANTUM COMPUTATION

This section overviews key concepts in quantum computation that are relevant to this work. For a comprehensive reference, please see Nielsen and Chuang [2010].

Superposition. The fundamental unit of quantum information is the *qubit*, a linear combination or *superposition* $\gamma_0 |0\rangle + \gamma_1 |1\rangle$ of the classical *basis states* 0 and 1, in which $\gamma_0, \gamma_1 \in \mathbb{C}$ are complex *amplitudes* satisfying $|\gamma_0|^2 + |\gamma_1|^2 = 1$ describing relative weights of basis states. Examples of qubits are classical $|0\rangle$ and $|1\rangle$, and the states $\frac{1}{\sqrt{2}}(|0\rangle + e^{i\varphi} |1\rangle)$ where $\varphi \in [0, 2\pi)$ is known as a *phase*.

More generally, a *quantum state* $|\psi\rangle$ is a superposition over n-bit strings. For example, $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is a quantum state over two qubits. Formally, multiple component states form a composite state by the *tensor product* $\otimes$, e.g. the state $|01\rangle$ is equal to $|0\rangle \otimes |1\rangle$. As is customary in quantum computation, we also use the notation $|0, 1\rangle$ to represent $|01\rangle = |0\rangle \otimes |1\rangle$.

Unitary Operator. A *unitary operator* U is a linear operator on quantum states that preserves inner products and whose inverse is its Hermitian adjoint $U^\dagger$. Formally, a unitary operator may be constructed as a circuit of *quantum gates*. The quantum gates over a single qubit include:

- Bit flip (X or NOT), which maps $|x\rangle \mapsto |1 - x\rangle$ for $x \in \{0, 1\}$;
- Phase flip (Z), which maps $|x\rangle \mapsto (-1)^x |x\rangle$;

- $\pi/4$ phase rotation (T), which maps $|x\rangle \mapsto e^{ix\pi/4}|x\rangle$;
- Hadamard (H), which maps $|x\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x |1\rangle)$.

A gate may be *controlled* by one or more qubits, forming a larger unitary operator. For example, the two-qubit CNOT gate maps $|0, x\rangle \mapsto |0, x\rangle$ and $|1, x\rangle \mapsto |1, \text{NOT } x\rangle = |1, 1 - x\rangle$. Generalized versions of CNOT are known as multi-controlled-X (MCX) gates. In particular, the MCX gate with two control bits is known as the Toffoli gate, and the MCX with zero controls is the NOT gate.

The *Clifford* gates [Gottesman 1998] are the quantum gates that can be constructed by compositions and tensor products of H, $S = T^2$, and CNOT. Examples of Clifford gates include $Z = S^2$ and $X = HZH$. By contrast, no MCX gate larger than CNOT is Clifford, and constructing e.g. a Toffoli gate requires the use of the non-Clifford T gate. The Clifford gates plus the T gate form the Clifford+T gates, the gate set of the predominant surface code for quantum error correction.

Measurement. Performing a *measurement* of a quantum state probabilistically collapses its superposition into a classical outcome. When a qubit $\gamma_0|0\rangle + \gamma_1|1\rangle$ is measured in the standard basis, the observed classical outcome is 0 with probability $|\gamma_0|^2$ and 1 with probability $|\gamma_1|^2$.

Entanglement. A state is *entangled* when it consists of two components but cannot be written as a tensor product of its components. For example, the *Bell state* [Bell 1964] $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled, as it cannot be written as a product of two independent qubits.

Given an entangled state, measuring one of its components causes the superposition of the other component to also collapse. For example, measuring the second qubit in the Bell state causes the state of the first qubit to also collapse, to either $|0\rangle$ or $|1\rangle$ with probability $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$ each.

Uncomputation. Entanglement means that in general, measuring or discarding a component of a quantum state can destroy the superposition of the remainder of the state. The consequence is that a quantum algorithm may not simply discard a temporary variable that it no longer needs, which could cause a superposition collapse that negates the possibility of quantum advantage. Instead, the algorithm must *uncompute* the variable [Bennett 1973; Bichsel et al. 2020], meaning it reverses the sequence of operations on that variable and returns it to its initial value of zero.

3 *T*-COMPLEXITY COSTS OF CONTROL FLOW IN QUANTUM PROGRAMS

In this section, we demonstrate how programming abstractions for control flow in superposition can cause a quantum program to have asymptotic time complexity worse than that found by its idealized theoretical analysis. These costs arise from the performance bottleneck of quantum error correction, and if not mitigated, can diminish the computational advantage of quantum algorithms.

3.1 Running Example

Quantum algorithms for search [Ambainis 2004; Grover 1996], game tree evaluation [Ambainis et al. 2010; Childs et al. 2007], combinatorial optimization [Bernstein et al. 2013], and geometry [Aaronson et al. 2020] utilize abstract data structures in superposition to achieve computational advantage. For example, they rely on a set to efficiently maintain a collection of items, check an item for membership, and add and remove items. In turn, an abstract set can be concretely implemented as a linked list, whose structure and contents exist in quantum superposition.

In Figure 1, we present a program in the language Tower [Yuan and Carbin 2022] to compute the length of a linked list. This `length` function accepts a pointer `xs` to the head of the list and a value `acc` that stores the number of list nodes traversed so far. Line 4 checks whether the list is empty, meaning that `xs` is null, and if so returns `acc`. If not, line 9 dereferences `xs` to obtain the pointer to the next list node, and line 11 adds 1 to the value of `acc`.

Recursion. On line 13, the function makes a recursive call. In Tower, all function calls are inlined by the compiler, and the values `n` and `n-1` statically instruct the compiler to unroll `length` to depth `n`. In the example, `n` is a concrete integer known at compile time, and `length` is effectively a family of functions whose `n`th instance returns the length of the list `xs` if it is less than `n`, or 0 otherwise.

Quantum Data. All data types in Tower denote data in quantum superposition. For example, when `xs` is a superposition of lists [], [1], and [1, 2, 3], the output of `length` is a superposition of the integers 0, 1, and 3. The `if`-statement on line 5 conditions on the value of `is_empty` in superposition, meaning it executes the `if`-clause on the classical states in the machine state superposition where `is_empty` is true, and the `else`-clause on all other states.

```
1  type list = (uint, ptr<list>);
2  fun length[n](xs: ptr<list>, acc: uint) {
3      with {
4          let is_empty <- xs == null;
5      } do if is_empty {
6          let out <- acc;
7      } else with {
8          let temp <- default<list>;
9          *xs <-> temp;
10         let next <- temp.2;
11         let r <- acc + 1;
12     } do {
13         let out <- length[n-1](next, r);
14     }
15     return out;
16 }
```

Fig. 1. Program computing the length of a list.

Uncomputation. The structure of the program in Figure 1 enables the use of uncomputation (Section 2) to clean up temporary values in the program. In Tower, an operator known as *un-assignment* **`let`** `x -> e` is defined as the reverse of the assignment **`let`** `x <- e`. Whereas assignment initializes `x` to zero and sets it to `e`, un-assignment resets `x` from `e` back to zero and deinitializes `x`.

In Figure 1, un-assignment does not appear explicitly but is performed implicitly by the `with`-`do` construct as follows. First, the `with`-block on lines 3 to 5 executes, initializing a variable `is_empty`. The `do`-block on lines 5 to 14 then executes, computing the result `out`. Then, the `with`-block is executed in reverse, with all assignments flipped to un-assignments and vice versa. That is, the inverse of lines 3 to 5 un-assigns and deinitializes `is_empty`. The function then returns `out`.

Equivalents of the `with`-`do` construct can be found in other quantum programming languages. Examples include the `within`-`apply` blocks of Q# [Svore et al. 2018] and the automatic uncomputation of scoped variables in Silq [Bichsel et al. 2020] and Qunity [Voichick et al. 2023].

3.2 Complexity Analysis and Diminished Advantage

Algorithms such as Aaronson et al. [2020]; Ambainis [2004]; Bernstein et al. [2013]; Grover [1996] offer theoretical advantage over classical algorithms that is sub-exponential, meaning it could be diminished if their implementation as a program introduces additional polynomial overhead.

Idealized Analysis. A standard analysis of `length` reveals that its time complexity is $O(n)$ where n is the recursion depth `n` from above. In this work, we assume that the bit width of integer and pointer registers is a small constant, with only the depth n of recursion considered a variable.[2] At each of n levels of recursion, `length` performs $O(1)$ work in primitive operations, and makes one recursive call. The recurrence $C(n) = O(1) + C(n-1)$ for time complexity yields $C(n) = O(n)$.

In Figure 2, we plot the empirical time complexity of `length` on an idealized quantum computer, as determined by compiling Figure 1 to a quantum circuit of multiply-controlled NOT (MCX) gates, and counting its *MCX-complexity*, i.e. number of MCX gates, which is $O(n)$ as above.

[2] For detailed discussion of the effect of a variable bit width on the T-complexity of a program, please see Appendix A.

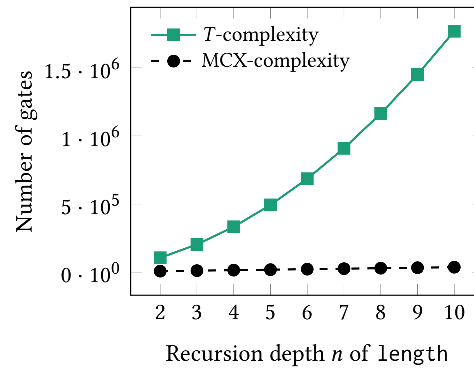


Fig. 2. Number of gates in the circuit of Figure 1.

Asymptotic Slowdown. Figure 2 also plots the empirical time complexity of `length` on a quantum computer with error correction, as found by compiling it to a circuit in the Clifford+T gate set, and counting its *T-complexity*, i.e. number of T gates.

The number of T gates is an appropriate metric because T gates act as the bottleneck of the *surface code* [Fowler et al. 2012], the prevailing quantum error correction code. On the surface code, realizing the T gate incurs an area-latency cost of about 10^2 times that of Clifford gates such as CNOT [Gidney and Fowler 2019] and 10^{10} times that of a NAND gate in classical transistors [Babbush et al. 2021].

As seen in Figure 2, the T-complexity of the program is not $O(n)$ but rather $O(n^2)$, meaning that a quantum algorithm that invokes `length` obtains diminished advantage under error correction. Such a slowdown does not fully erase the theoretical advantage of Ambainis [2004], which is $O(N^{1/3})$ where N is the size of the input. In Ambainis [2004], the depth of the data structure and hence of recursion is only poly-logarithmic in N, i.e. $O(\log^c N)$ for some constant c. However, such a slowdown jeopardizes instances of quantum search [Grover 1996] in which the advantage is $O(N^{1/2})$ and the depth n of each query is $O(N^{1/2})$ or greater.

3.3 *T*-Complexity Costs of Control Flow

The cause of the disparity is that on an error-corrected quantum computer, logic gates that are controlled by more bits are more costly to realize in terms of T-complexity. In turn, these control bits accrue in the compiled form of a control flow abstraction such as the quantum `if`-statement.

Compilation of Control Flow. To execute on a quantum computer, a program such as Figure 1 is compiled to a quantum circuit, a fixed sequence of logic gates controlled by individual bits. Each statement compiles to gates controlled by all of the qubits that lead to that control flow path.

To demonstrate this translation on a smaller scale, in Figure 3 we depict a simple program that uses quantum `if`-statements. Given Booleans `x`, `y`, and `z`, the program sets the value of output variables `a` and `b` to the negation of `z` and true respectively, when `x`, `y`, and `z` are all true. Though a toy example, this program exemplifies the same overheads of control flow as in Figure 1.

In Figure 4, we depict the circuit to which Figure 3 compiles, which has wires labeled with the name of each program variable. Gates labeled X denote NOT gates, while gates with black dots denote bit-controlled gates that execute only if all control bits, denoted by the dots, are true.

The nested quantum `if`-statements on lines 1 and 2 compile to a sequence of gates controlled by both `x` and `y`. Line 4 compiles to the first gate, a controlled-NOT (CNOT) gate that flips `t` based on the value of `z`. In turn, this CNOT is controlled by `x` and `y`. Next, the quantum `if` on line 6 compiles to gates controlled by `x`, `y`, and `z`. Line 7 compiles to three gates — a CNOT over `t` and `a`, surrounded by NOT gates on `t`. Line 8 compiles to the next gate, a NOT over `b`. Finally, the semantics of the `with`-block states that line 4 is reversed after the `do`-block, corresponding to the last gate.

Error Correction. If we were targeting an ideal quantum computer not constrained by hardware, then the circuit in Figure 4 consisting of MCX gates could serve as the final representation of the program. Indeed, the idealized analysis of `length` finds its MCX-complexity, which is linear.

By contrast, a computer that uses the surface code [Fowler et al. 2012] for error correction supports the restricted Clifford+T gate set, to which MCX gates larger than CNOT must be decomposed. In

```
if x {
  if y {
    with {
      let t <- z;
    } do {
      if z {
        let a <- not t;
        let b <- true;
} } } }
```

Fig. 3. Tower program that uses nested quantum `if`-statements.

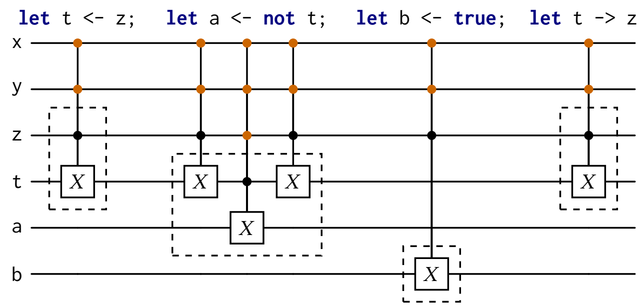


Fig. 4. Translation of Figure 3 to a circuit. On each multiply-controlled-NOT (MCX) gate, each orange control bit incurs T-complexity.

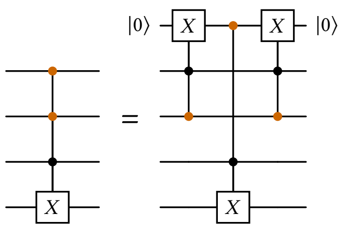


Fig. 5. Decomposing MCX to Toffoli.

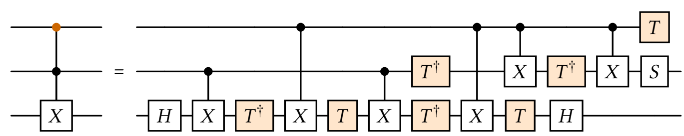


Fig. 6. Decomposing Toffoli into Clifford+T gates.

```
with {
  let t <- z;
  let s <- x && y && z;
} do {
  if s {
    let a <- not t;
    let b <- true;
} }
```

Fig. 7. Optimized version of Figure 3.

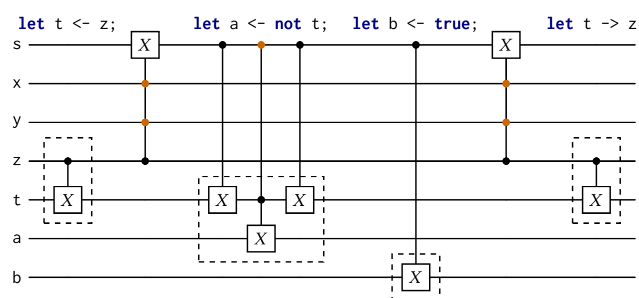


Fig. 8. Quantum circuit that corresponds to Figure 7.

Figure 5, we depict how an MCX gate decomposes into Toffoli gates by the process of Barenco et al. [1995]. Then, in Figure 6, we depict how Toffoli decomposes into Clifford+T gates.

The decomposition of an MCX to a Clifford+T circuit introduces T-complexity. For example, Figure 6 uses 7 T gates to decompose one Toffoli gate,[3] meaning that Figure 5 uses $3 \times 7 = 21$ T gates to decompose an MCX gate with 3 control bits. In general, Beverland et al. [2020, Proposition 4.1] prove that an MCX gate with $n \geq 2$ controls requires at least $n + 1$ T gates to realize. This lower bound is not reached in practice by Figures 5 and 6, which instead require $7 \times (2(n - 2) + 1)$ T gates.

Costs of Control Flow. In other words, on error-corrected quantum hardware, instructions become more costly to execute as the program's control flow becomes more deeply nested. To accurately predict performance under error correction, one must account for the T-complexity of each control bit beyond the first on each MCX gate — only the first is free in principle because CNOT is a Clifford gate. In Figure 4, we highlight these additional control bits in orange. In addition to the 6 MCX gates, the 13 orange controls cost at least $7 \times 2 \times 13 = 182$ T gates using Figures 5 and 6.

[3]As a technical note, the gate $T^\dagger = TSZ$ has a T-complexity of 1, as it can be realized using Clifford gates plus one T gate.

```
1  fun length[n](xs: ptr<list>, acc: uint) {
2   with {
3    let is_empty <- xs == null;
4   } do {
5    if is_empty { let out <- acc; }
6    else with {
7     /* elided: compute next, r */
8     let is_empty2 <- next == null;
9    } do {
10    if is_empty2 { let out <- r; }
11    else with {
12     /* elided: compute next2, r2 */
13     let is_empty3 <- next2 == null;
14    } do {
15     if is_empty3 { let out <- r2; }
16     else with {
17      let temp <- default<list>;
18      *next2 <-> temp;
19      let next3 <- temp.2;
20      let r3 <- r2 + 1;
21     } do {
22      let out <- length[n-3](next3, r3);
23  }}}}
24  return out;
25 }
```

Fig. 9. Version of Figure 1 inlined to 3 levels of recursion, depicting the nesting of conditionals.

```
fun length[n](xs: ptr<list>, acc: uint) {
 with {
  let is_empty <- xs == null;
  let not_empty <- not is_empty;
  /* elided: compute next, r */
  let is_empty2 <- not_empty && next == null;
  let not_empty2 <- not_empty && next != null;
  /* elided: compute next2, r2 */
  let is_empty3 <- not_empty2 && next2 == null;
  let not_empty3 <- not_empty2 && next2 != null;
  let temp <- default<list>;
  *next2 <-> temp;
  let next3 <- temp.2;
  let r3 <- r2 + 1;
 } do {
  if is_empty { let out <- acc; }
  if is_empty2 { let out <- r; }
  if is_empty3 { let out <- r2; }
  if not_empty3 {
   let out <- length[n-3](next3, r3);
 }}
 return out;
}
```

Fig. 10. Optimized version of Figure 9.

3.4 Cost Model for Accurately Predicting T-Complexity Costs

The increased cost of control flow under quantum error correction explains the discrepancy between the idealized analysis of `length` in Section 3.2 and the empirical gate counts in Figure 2. We now demonstrate how using our T-complexity cost model, a developer can conduct an analysis that pinpoints the overhead of control flow in a quantum program.

Running Example. In Figure 9, we illustrate a version of `length` in which the recursive call on line 13 of Figure 1 has been unfolded and inlined twice to reveal the nesting of `if`-statements. This program features three levels of nested `if`, highlighted in orange. When the program is compiled to MCX gates, each `if` becomes a sequence of control bits placed over its branches. For example, the gates corresponding to the assignment on line 5 are conditioned by `is_empty`.

The source of the asymptotic cost is that nested conditional statements compile to nested control bits. For example, the assignment on line 13 lies under two levels of `if`-statements, and compiles to a sequence of gates that are controlled by `is_empty` and `is_empty2`. Likewise, the assignment on line 15 is controlled by three bits, as are all of lines 17 to 20.

Analysis with Cost Model. Returning to the recursive form of `length` in Figure 1, we now use our cost model to repair the analysis of Section 3.2 to account for the T-complexity of control flow. Let $C^{\mathrm{MCX}}(n)$ denote the MCX-complexity and $C^{T}(n)$ the T-complexity of the program.

To compute $C^{T}(n)$, we start as before with the $O(1)$ primitive operations per level and the $C^{T}(n-1)$ term for the recursive call. Next, we account for the T-complexity of control flow. In Figure 1, the `if-else` on lines 5 to 14 incurs one control bit for each statement on lines 6 to 11, adding an $O(1)$ term. On line 13, the `if` incurs $O(1)$ cost for each of the $C^{\mathrm{MCX}}(n-1) = O(n)$

primitive operations in the recursive call. The final recurrence is:

$$C^T(n) = \underbrace{O(1)}_{\text{operations in level}} + \underbrace{C^T(n-1)}_{\text{recursive call}} + \underbrace{O(1)}_{\substack{\text{control flow over}\\\text{operations in level}}} + \underbrace{C^{\text{MCX}}(n-1)}_{\substack{\text{control flow over}\\\text{recursive call}}} = C^T(n-1) + O(n)$$

which yields $C^T(n) = O(n^2)$, agreeing with the empirical results in Figure 2.

3.5 Program-Level Optimizations for Mitigating *T*-Complexity Costs

We showed that control flow can incur asymptotic overhead in T-complexity when compiled using a straightforward strategy. We next present two *program-level optimizations* that rewrite the syntax of the program and produce a new program that then compiles using the same straightforward strategy to a circuit with reduced T-complexity. The first one, *conditional flattening*, can provide an asymptotic speedup while the second, *conditional narrowing*, yields additional constant speedups.

Conditional Flattening. In Figure 7, we present an optimized form of Figure 3 that has been subject to both optimizations. First, the *conditional flattening* optimization eliminates control bits that are introduced by nested `if`-statements, by flattening them via the use of temporary variables. Whereas the original program in Figure 3 uses three `if`-statements on lines 1, 2, and 6, the optimized program in Figure 7 introduces a variable `s` on line 3 and uses it in a single `if` on line 5.

The benefit can be seen in Figure 8, the circuit to which Figure 7 compiles. The gates to which lines 6 and 7 compile are now controlled by only `s` rather than `x`, `y`, and `z` as in the original circuit in Figure 4, saving 8 control bits or $7 \times 2 \times 8 = 112$ T gates. Though the computation of `s` adds 4 control bits, this cost is asymptotically constant with respect to the length of the body of the `if`.

Conditional Narrowing. Second, the *conditional narrowing* optimization eliminates control bits introduced by a `with`-`do` block under an `if`-statement, by moving the `if` into the `do`-block. In Figure 7, line 2 is no longer under an `if` as in the original Figure 3. As a result, in Figure 8, the first and last gates are not controlled by `x`, `y`, and `z`, saving 4 more control bits over Figure 4.

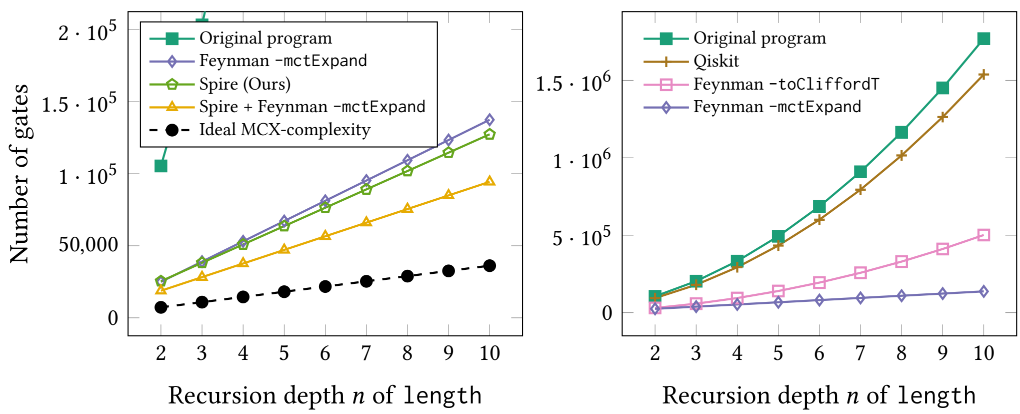
Running Example. In Figure 10, we depict the result of optimizations on the unfolded `length` program from Figure 9. Conditional flattening turns 3 levels of nested `if` to 1, such that assuming 8-bit registers, lines 17 and 18 save $7 \times 2 \times (2-1+3-1) \times 8 = 336$ T gates. Accounting for uncomputation, the use of temporary variables adds $7 \times 2 \times 2 \times 2 = 56$ T gates, for a net savings of $336 - 56 = 280$ T gates.

Next, conditional narrowing saves a further $7 \times 2 \times 4 \times 8 = 448$ T gates by moving lines 11 to 14 outside `if`-statements. Notably, the program remains safe even though pointer dereferences have been moved outside null checks. All writes to the observable output `out` remain guarded by appropriate checks, meaning uninitialized data never propagates to the output.

```
fun length[n](xs, acc: uint, b: bool) {
 with {
  let is_empty <- b && xs == null;
  let n_e <- b && xs != null;
  let temp <- default<list>;
  *xs <-> temp;
  let next <- temp.2;
  let r <- acc + 1;
 } do {
  if is_empty { let out <- acc; }
  let rest <- length[n-1](next, r, n_e);
  if n_e { out <-> rest; }
  let rest -> 0;
 }
 return out;
}
```

Fig. 11. Program that reflects the optimizations in Figure 10 back to the original form in Figure 1.

Efficient T-Complexity. When applied to the original `length` program from Figure 1, these optimizations produce the program depicted in Figure 11. In this program, recursion no longer takes place under nested `if`. As a result, in the T-complexity analysis, control flow incurs only $O(1)$ overhead, and the recurrence yields an asymptotically efficient $O(n)$ for the optimized program.



(a) Program-level optimizations in Spire.

(b) Existing quantum circuit optimizers.

Fig. 12. T-complexity of `length` after quantum circuit optimizers and program-level optimizations in Spire.

3.6 Comparison to Quantum Circuit Optimizers

In principle, an alternative to the approach of program-level optimizations – rewrite the program so that it straightforwardly compiles to a more efficient circuit – is to emit the asymptotically inefficient circuit of the original program and then attempt to recover an asymptotically efficient circuit using a general-purpose *quantum circuit optimizer* [Hietala et al. 2021; Kissinger and van de Wetering 2020; QuiZX Developers 2022; Sivarajah et al. 2020; Xu et al. 2023, 2022] that researchers have developed to remove and replace inefficient sequences of gates in circuits.

To compare these approaches, we implemented both program-level optimizations in Spire, an extension to the Tower compiler. In Figure 12a, we plot the T-complexity of `length` after Spire's optimizations only, and no circuit optimizer. In Figure 12b, we plot the T-complexity without Spire's optimizations, and only the Qiskit [Qiskit Developers 2021] and Feynman [Amy 2024; Amy et al. 2014] circuit optimizers. We also plot in Figure 12a the results of a combined approach — running Spire on the original program, compiling the optimized program to a circuit, and then running Feynman on that circuit. Lastly, we plot the idealized MCX-complexity from Figure 2 for reference.[4]

First, Qiskit and one configuration of Feynman do not produce a circuit with linear T-complexity, while Spire and a second configuration of Feynman do. A possible explanation for the difference is that conditional flattening is not captured by the rewrites of Clifford+T gates that Qiskit implements. By contrast, as we discuss in Section 8.5, the optimization can be captured by cancelling adjacent Clifford+Toffoli gates, enabling a configuration of Feynman using that strategy to succeed.

Next, Spire and Feynman together achieve better speedups than either alone — Feynman leaves behind some fraction of T gates that Spire can eliminate. As we discuss in Section 8.5, one challenge that a circuit optimizer must overcome to fully capture the effect of conditional narrowing is that the circuit optimizer must perform rewrites over an unbounded number of gates.

Finally, Spire takes only 0.05 s to emit an efficient circuit, whereas Feynman takes 2 minutes to do so in this case. The reason is that whereas the circuit optimizer must process a large circuit to shrink it down, Spire optimizes the program so that the large circuit is not created in the first place.

[4] As a technical note, the MCX-complexity is the performance on an idealized architecture and not the minimal T-complexity to implement the function. The MCX and T-complexities should be compared only in terms of asymptotics, not constants.

$$\begin{aligned}
\text{Type } \tau &::= () \mid \texttt{uint} \mid \texttt{bool} \mid (\tau_1, \tau_2) \mid \texttt{ptr}(\tau) \\
\text{Value } v &::= x \mid () \mid (x_1, x_2) \mid \overline{n} \mid \texttt{true} \mid \texttt{false} \mid \texttt{null}_\tau \mid \texttt{ptr}_\tau[p] \quad (n \in \text{UInt}, p \in \text{Addr}) \\
\text{Expression } e &::= v \mid \pi_1(x) \mid \pi_2(x) \mid uop\ x \mid x_1\ bop\ x_2 \\
\text{Operator } uop &::= \texttt{not} \mid \texttt{test} \qquad bop ::= \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \\
\text{Statement } s &::= \texttt{if}\ x\ \{\ s\ \} \mid s_1;\ s_2 \mid \texttt{skip} \mid x \leftarrow e \mid x \rightarrow e \mid H(x) \mid x_1 \Leftrightarrow x_2 \mid *x_1 \Leftrightarrow x_2
\end{aligned}$$

Fig. 13. Core syntax of the Tower quantum programming language.

4 TOWER LANGUAGE OVERVIEW

In this section, we briefly review the syntax and semantics of Tower [Yuan and Carbin 2022], a quantum programming language featuring abstractions for control flow in superposition.

Language Syntax. The Tower language features the data types of integers, tuples, and pointers, along with operations on these data types. In Figure 13, we depict the core syntax of the language.

In Tower, all recursive function definitions and calls are inlined by the compiler, producing a program that uses only the core syntax above [Yuan and Carbin 2022, Section 6]. In the example from Figure 1, the annotation `n` instructs the compiler to inline `length` into itself `n` times.

Apart from standard imperative programming features, Tower supports a number of constructs necessary for quantum programming. The *un-assignment* construct $x \rightarrow e$ uncomputes (Section 2) the value of x using the value of e. The construct $x_1 \Leftrightarrow x_2$ swaps the values of variables x_1 and x_2, and $*x_1 \Leftrightarrow x_2$ swaps a value stored in memory at pointer x_1 with the value of x_2.

We study a version of Tower extended with a statement $H(x)$ that executes a Hadamard gate (Section 2) on the Boolean variable x. Because the Hadamard and Toffoli gates are universal for quantum computation [Shi 2003], the availability of the Hadamard and NOT gates and the $\texttt{if}\ x\ \{\ s\ \}$ construct means that any quantum computation can be expressed as a Tower program.

Language Semantics. The type system of Tower assigns a type to each value or expression and determines whether a statement is well-formed. In Appendix B, we define typing for values and expressions, and the judgment $\Gamma \vdash s \dashv \Gamma'$, which states that the statement s is well-formed under a context Γ of variables and produces a context Γ' of the updated declarations after executing s.

The circuit semantics of Tower assigns to each program s a corresponding quantum circuit $C[\![s]\!]$ that can execute on a quantum computer. In Appendix B, we define this semantics and specifically how $C[\![s]\!]$ maps an input machine state $|R, M\rangle$ to an output machine state $|R', M'\rangle$. Here, R denotes a *register file* mapping variables to values and M denotes a *memory* mapping addresses to values.

In Tower, the dereferencing of a null pointer is a no-op, not a runtime error. When a variable is re-defined, the value of its corresponding register becomes the XOR of its old and new values.

Derived Forms. Each statement s in Tower is *reversible*, meaning that there exists a statement $\mathcal{I}[s]$ whose semantics are the reverse of s. Specifically, $\mathcal{I}[s_1;\ s_2]$ is $\mathcal{I}[s_2];\ \mathcal{I}[s_1]$. Similarly, $\mathcal{I}[x \leftarrow e]$ is $x \rightarrow e$ and vice versa, $\mathcal{I}[\texttt{if}\ x\ \{\ s\ \}]$ is $\texttt{if}\ x\ \{\ \mathcal{I}[s]\ \}$, and the reverse of any other s is s itself.

Based on this concept, we define the derived form $\texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ s_2\ \}$ as $s_1;\ s_2;\ \mathcal{I}[s_1]$, and use it to automate the insertion of uncomputation statements for variables within block scope. Memory allocation and deallocation desugar to core constructs, following the process described in Yuan and Carbin [2022, Section 5]. Other derived forms, such as the `if-else` construct, are described in Yuan and Carbin [2022, Appendix B] and similarly desugar to core constructs.

5 COST MODEL

In this section, we present a cost model that computes the T-complexity of a quantum program that utilizes programming abstractions for control flow in superposition. Using the cost model, a developer can perform a syntactic analysis that determines the runtime cost of a program on an error-corrected quantum architecture and pinpoint the sources of asymptotic slowdown.

Given a program s, the cost model quantifies the number of gates in the circuit $C[\![s]\!]$ to which it compiles, following the semantics of Section 4. More generally, the cost model also matches the compilation of other languages with quantum `if`, such as QML [Altenkirch and Grattage 2005], ScaffCC [JavadiAbhari et al. 2014], Silq [Hans and Groppe 2022], and Qunity [Voichick et al. 2023].

MCX-Complexity. We denote by $C^{\text{MCX}}(s)$ the MCX-complexity of the program s, which is formally defined as the number of gates in its compiled circuit $C[\![s]\!]$ when expressed in an idealized gate set consisting of arbitrarily controllable Clifford gates, which includes arbitrary MCX gates:

$$C^{\text{MCX}}(\texttt{skip}) = 0 \qquad C^{\text{MCX}}(s_1;\, s_2) = C^{\text{MCX}}(s_1) + C^{\text{MCX}}(s_2)$$
$$C^{\text{MCX}}(\texttt{if}\ x\ \{\ s\ \}) = C^{\text{MCX}}(s) \qquad C^{\text{MCX}}(s) = c_s^{\text{MCX}} \text{ for any other } s$$

where $0 \le c_s^{\text{MCX}} = O(1)$ represents the number of arbitrarily controllable Clifford gates, including MCX gates, used by the primitive operation s. This constant is determined by the implementation of s, and all primitive s satisfy $c_s^{\text{MCX}} > 0$ except for only `skip` or $x \leftarrow v$ or $x \rightarrow v$ where v has an all-zero bit representation for which no gates are emitted. The reason for why the `if`-statement does not increase the MCX-complexity is that the number of arbitrarily controllable Clifford gates, including MCX gates, does not change when more control bits are added to gates.

Theorem 5.1 (MCX-Complexity Soundness). *If s is well-formed, i.e. $\Gamma \vdash s \dashv \Gamma'$, then the number of arbitrarily controllable Clifford gates in $C[\![s]\!]$ is equal to $C^{MCX}(s)$, up to choices for c_s^{MCX}.*

Proof. By induction on the definition of $C[\![s]\!]$. The significant case is `if`, as explained above. □

T-Complexity. We denote by $C^T(s)$ the T-complexity of the program s, which is formally defined as the number of T gates in its compiled circuit $C[\![s]\!]$ when expressed in the Clifford+T gate set:

$$C^T(\texttt{skip}) = 0 \qquad C^T(s_1;\, s_2) = C^T(s_1) + C^T(s_2)$$
$$C^T(\texttt{if}\ x\ \{\ s_1;\, s_2\ \}) = C^T(\texttt{if}\ x\ \{\ s_1\ \}) + C^T(\texttt{if}\ x\ \{\ s_2\ \}) \qquad C^T(\texttt{if}\ x\ \{\ H(y)\ \}) = c^T_{CH}$$
$$C^T(\texttt{if}\ x\ \{\ y \leftarrow v\ \}) = C^T(\texttt{if}\ x\ \{\ y \rightarrow v\ \}) = 0 \text{ for value } v$$
$$C^T(\texttt{if}\ x\ \{\ s\ \}) = c^T_{\text{ctrl}} * C^{\text{MCX}}(s) + C^T(s) \text{ for other } s \qquad C^T(s) = c^T_s \text{ for other } s$$

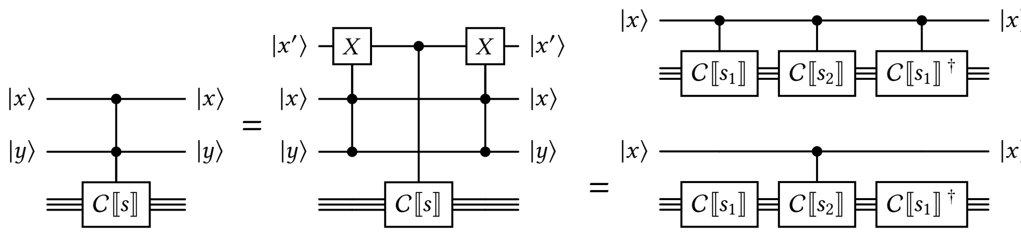
where $0 \le c_s^T = O(1)$ represents the number of T gates used by the primitive operation s, which is determined by the implementation of s. Simple s such as $x \leftarrow v$ and $H(x)$ have $c_s^T = 0$, whereas others such as $x \leftarrow y * z$ for which an arithmetic circuit must be instantiated have $c_s^T > 0$. The constant $0 < c^T_{CH} = O(1)$ represents the number of T gates required to implement a controlled-Hadamard gate. Using the construction of Lee et al. [2021, Figure 17], we have $c^T_{CH} = 8$. The constant $0 < c^T_{\text{ctrl}} = O(1)$ represents the number of T gates required to add an additional control bit to a multi-controlled gate. Using the decompositions in Figures 5 and 6, we have $c^T_{\text{ctrl}} = 2 \times 7 = 14$.

Theorem 5.2 (T-Complexity Soundness). *If s is well-formed, i.e. $\Gamma \vdash s \dashv \Gamma'$, then the number of T gates in $C[\![s]\!]$ is equal to $C^T(s)$, up to choices for the constants c_s^{MCX}, c_s^T, c^T_{CH}, and c^T_{ctrl}.*

Proof. By induction on $C[\![s]\!]$. There are three significant cases. The first is `if` x { $y \leftarrow v$ } and `if` x { $y \rightarrow v$ }, which add a control bit to a circuit $C[\![y \leftarrow v]\!]$ or $C[\![y \rightarrow v]\!]$ respectively that does not contain any controlled or Hadamard gates, meaning that the resulting T-complexity is zero.

The second is `if` x { $H(y)$ }, a controlled-Hadamard gate with cost c^T_{CH} by definition.

The third is `if` x { s } where s is $y \leftarrow e$ or $y \rightarrow e$ or `if` y { s' } or $y \Leftrightarrow z$ or $*y \Leftrightarrow z$. In these cases, the number of gates in $C[\![s]\!]$ that are not Clifford when one more control bit is added is proportional to $C^{\text{MCX}}(s)$, and adding a control for x incurs a T-complexity of c^T_{ctrl} at each such gate. □



(a) Conditional flattening optimization.

(b) Conditional narrowing optimization.

Fig. 14. Circuit equivalence rules that hold by direct reasoning on quantum circuits and visually demonstrate the soundness of the program-level optimizations. The notation ≡ denotes a collection of many registers.

6 PROGRAM-LEVEL OPTIMIZATIONS

In this section, we present *program-level optimizations* for quantum programs that utilize control flow in superposition. Using these optimizations, a developer can rewrite a program to reduce its T-complexity, predict the T-complexity of the optimized program using the cost model, and then compile the program to an efficient circuit using a straightforward strategy. Forms of these optimizations appear in prior work [Ittah et al. 2022; Seidel et al. 2022; Steiger et al. 2018], and we present in this section a novel unification of these optimizations as program rewrite rules.

6.1 Conditional Flattening Optimization

The conditional flattening optimization identifies instances in which control bits are introduced by nested `if`-statements and can be optimized by flattening the structure of `if`-statements. Specifically, the optimization performs these following program rewrite rules whenever possible:

$$\texttt{if}\ x\ \{\ \texttt{if}\ y\ \{\ s\ \}\ \} \rightsquigarrow \texttt{with}\ \{\ x' \leftarrow x\ \texttt{\&\&}\ y\ \}\ \texttt{do}\ \{\ \texttt{if}\ x'\ \{\ s\ \}\ \}$$
$$\texttt{if}\ x\ \{\ s_1;\ s_2\ \} \rightsquigarrow \texttt{if}\ x\ \{\ s_1\ \};\ \texttt{if}\ x\ \{\ s_2\ \}$$

Whereas the original program incurs many control bits over s, the optimized program computes a temporary value and uses it to control s using only one bit, yielding an asymptotic improvement:

Theorem 6.1. *When $C[\![s]\!]$ contains k MCX gates with at least one control and s falls under n levels of nested* `if`*, conditional flattening reduces the T-complexity of the program from $O(kn)$ to $O(k+n)$.*

Proof. By induction on the structure of s. For each of the $n-1$ layers of `if` that is removed, the T-complexity of the program reduces by k, while the inserted `with`-block has $O(1)$ T-complexity. □

We next show that this program-level optimization preserves the circuit semantics of a program with respect to its free variables, as formalized by the following definition:

Definition 6.2 (Circuit Equivalence). Given a set X of variables, we say that register files R_1 and R_2 are *equivalent*, denoted $R_1 \equiv_X R_2$, when they map the variables in X to equal values respectively and all other variables to zero. Given two sets X, X' of variables, we say that circuits C_1 and C_2 are *equivalent*, denoted $X \vdash C_1 \equiv C_2 \dashv X'$, when given any memory M and two register files R_1 and R_2 such that $R_1 \equiv_X R_2$, we have $C_1 |R_1, M\rangle = |R_1', M'\rangle$ and $C_2 |R_2, M\rangle = |R_2', M'\rangle$ where $R_1' \equiv_{X'} R_2'$.

Theorem 6.3 (Conditional Flattening Soundness). *Assume* $\Gamma \vdash \texttt{if}\ x\ \{\ \texttt{if}\ y\ \{\ s\ \}\ \} \dashv \Gamma'$. *Then, we have* $\operatorname{dom}\Gamma \vdash C[\![\texttt{if}\ x\ \{\ \texttt{if}\ y\ \{\ s\ \}\ \}]\!] \equiv C[\![\texttt{with}\ \{\ x' \leftarrow x\ \texttt{\&\&}\ y\ \}\ \texttt{do}\ \{\ \texttt{if}\ x'\ \{\ s\ \}\ \}]\!] \dashv \operatorname{dom}\Gamma'$.

Proof. The claim follows from a circuit equivalence that we visually depict in Figure 14a. □

6.2 Conditional Narrowing Optimization

The conditional narrowing optimization identifies instances in which control bits are introduced by a `with`-`do` block under an `if`-statement, which can be optimized by moving the `if`-statement under the do-block. Specifically, the optimization performs the following rewrite whenever possible:

$$\texttt{if}\ x\ \{\ \texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ s_2\ \}\ \} \rightsquigarrow \texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ \texttt{if}\ x\ \{\ s_2\ \}\ \}$$

The optimized program unconditionally executes s_1 and its reverse, for a constant improvement:

Theorem 6.4. *When $C[\![s_1]\!]$ contains k MCX gates with at least one control, the conditional narrowing optimization reduces the T-complexity of the program by an $O(k)$ additive term.*

Proof. By induction on the structure of s_1, where k controls are removed on s_1 and its reverse. □

Theorem 6.5 (Conditional Narrowing Soundness). *Let* $\Gamma \vdash \texttt{if}\ x\ \{\ \texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ s_2\ \}\ \} \dashv \Gamma'$. *Then,* $\mathrm{dom}\,\Gamma \vdash C[\![\texttt{if}\ x\ \{\ \texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ s_2\ \}\ \}]\!] \equiv C[\![\texttt{with}\ \{\ s_1\ \}\ \texttt{do}\ \{\ \texttt{if}\ x\ \{\ s_2\ \}\ \}]\!] \dashv \mathrm{dom}\,\Gamma'$.

Proof. The claim follows from a circuit equivalence that we visually depict in Figure 14b. □

7 IMPLEMENTATION: SPIRE QUANTUM COMPILER

As the artifact of this work, we implemented Spire, an extension of the Tower compiler that performs the optimizations of Section 6. In this section, we briefly describe the architecture of the Tower compiler, the transformations added by Spire, and the challenges that arose in implementation.

Compiler Overview. The Tower compiler has four main stages. First, given a Tower program, the lexer and parser construct its abstract syntax tree. Next, the compiler lowers the surface AST to the *core intermediate representation*, whose syntax is presented in Section 4. This lowering involves inlining all function calls and translating memory allocation and derived forms to core syntax.

Then, the compiler lowers the core IR to an *abstract circuit* that is analogous to classical assembly, with the abstractions of word-sized registers; arithmetic, logical, memory, and data movement instructions; and instructions controlled by registers. The compiler invokes a register allocator to map IR variables to registers and compiles `if`-statements to multiply-controlled instructions.

Finally, the compiler lowers the abstract circuit to a *concrete circuit* by instantiating each arithmetic, logical, memory, and data movement instruction as an explicit sequence of MCX gates. The compiler then emits the concrete circuit in the quantum circuit format of Mosca [2016].

Spire Transformations. We implemented Spire as a compiler pass that transforms the core IR. First, we modified the core IR to add `with`-`do` blocks, facilitating the conditional narrowing optimization. Next, we implemented a compiler pass that rewrites the core IR using the conditional flattening and conditional narrowing optimizations. As they are simple syntax rewrites, this pass constitutes only 12 lines of OCaml code, which we present in Appendix C. Then, we added a simple compiler pass that flattens the structure of `with`-`do` blocks before continuing to the next stage.

Downstream Challenges. Though the new passes are simple, they required detailed analysis and altered assumptions in the register allocation approach taken by the compiler. In Appendix D, we detail the challenge that arises and our solution, as a case study for quantum compiler developers.

8 EVALUATION

In this section, we evaluate our cost model and optimizations as measured by the T-complexity of a benchmark suite of quantum programs. We answer the following research questions:

RQ1. How accurately does the cost model predict the asymptotic T-complexity of programs?

RQ2. By how much do the program-level optimizations of conditional flattening and conditional narrowing improve the T-complexity of a quantum program?

RQ3. By how much do quantum circuit optimizers from existing work improve the T-complexity of a quantum program after it has been fully compiled to a circuit of logic gates?

RQ4. What is the effect on compilation time of performing the program-level optimizations, and how does it compare to the effect on compilation time of quantum circuit optimizers?

In Table 1, we list the benchmarks that we use throughout this evaluation and include in the paper artifact. They are data structure operations used by quantum algorithms for search [Ambainis 2004], optimization [Bernstein et al. 2013], and geometry [Aaronson et al. 2020], and include the `length` example from Section 3 and others such as insertion into a radix tree-based set.

In Sections 8.2 and 8.3, we also introduce `length-simple`, a simplified version of `length` that has the same asymptotic T-complexity but omits the primitive operations on lines 9 and 11. These lines perform a memory dereference and an addition operation respectively. Semantically, dropping them causes the `length` function to return an incorrect output. For compilation, dropping them results in a circuit whose size has the same asymptotic behavior but is scaled down by a fraction.

The reason we perform this simplification is to enable a comparison to existing quantum circuit optimizers. Without this simplification, the circuit would be two orders of magnitude larger, meaning that all but one of the existing optimizers we tested would take more than 1 hour to run.

8.1 RQ1: Accuracy of Cost Model

RQ1. How accurately does the cost model predict the asymptotic T-complexity of programs?

Methodology. To obtain the predicted asymptotic T-complexity, we performed the same analysis as in Section 3.4. We performed an asymptotic analysis because the values of constants in the cost function, in particular the costs of primitive operations such as arithmetic and memory, are difficult to determine theoretically and significantly affect the precision of non-asymptotic estimates.

As an example, the function `insert` in Table 1 inserts an element into a set data structure that is concretely implemented as a radix tree. This function invokes a string `compare` operation and a recursive call at each level, all under an `if`. Because the other operations in the program have equal or less T-complexity compared to `compare`, the overall T-complexity of `insert` is:

$$\begin{aligned} C^T_{\texttt{insert}}(d) &= \underbrace{C^T_{\texttt{compare}}(d)}_{\text{operations in level}} + \underbrace{C^{\text{MCX}}_{\texttt{compare}}(d)}_{\text{control flow in level}} + \underbrace{C^T_{\texttt{insert}}(d-1)}_{\text{recursive call}} + \underbrace{C^{\text{MCX}}_{\texttt{insert}}(d-1)}_{\text{control flow over recursive call}} \\ &= O(d^2) + O(d) + C^T_{\texttt{insert}}(d-1) + O(d^2) \end{aligned}$$

which solves to $C^T_{\texttt{insert}}(d) = O(d^3)$, an asymptotic increase over the MCX-complexity of $O(d^2)$.

To compute the empirical T-complexity, we used Spire (Section 7) with optimizations off to compile each program to MCX gates. We then counted T gates as follows: each MCX with $c \geq 2$ controls corresponds to $2(c-2)+1$ Toffoli gates as in Figure 5, and each Toffoli corresponds to 7 T gates as in Figure 6. To determine the scaling in the recursion depth n or d, we repeated the process for depths from 2 to 10 and found the lowest-degree polynomial that exactly fits the T-complexities.

To obtain the predicted and empirical MCX-complexity, we performed the same procedure as above, except that we used the MCX-complexity recurrence and counted the number of MCX gates.

Results. For each benchmark in Table 1, the cost model accurately predicts the asymptotic T-complexity, as confirmed by the matching empirical T-complexity. In particular, for each benchmark whose MCX-complexity is not constant, meaning the recurrence is nontrivial, it accurately predicts that the T-complexity of the unoptimized program is one degree higher than the MCX-complexity.

Table 1. List of benchmark programs and their MCX and T-complexities, in terms of the size n or depth $d = O(\log n)$ of the data structure. We report T-complexity both before and after program-level optimizations. "Predicted" reports the asymptotic MCX or T-complexity predicted by the cost model, and "Empirical" reports the MCX or T-complexity of the compiled circuit. Large empirical figures are reported in Appendix E.

	MCX-Complexity		T-Complexity Before Optimizations		T-Complexity After Optimizations	
Program	Predicted	Empirical	Predicted	Empirical	Predicted	Empirical
List						
– `length`	$O(n)$	$2246n + 32$	$O(n^2)$	$15722n^2 + 19292n + 3934$	$O(n)$	$12740n - 42$
– `sum`	$O(n)$	$2642n + 32$	$O(n^2)$	$18494n^2 + 19628n + 4298$	$O(n)$	$13272n - 42$
– `find_pos`	$O(n)$	$2294n + 32$	$O(n^2)$	$16058n^2 - 8820n + 6426$	$O(n)$	$12740n - 42$
– `remove`	$O(n)$	$4990n + 32$	$O(n^2)$	$34930n^2 + 26376n + 10304$	$O(n)$	$58912n - 12124$
Queue						
– `push_back`	$O(n)$	$2864n + 32$	$O(n^2)$	$20048n^2 + 11508n + 4634$	$O(n)$	$46256n - 13006$
– `pop_front`	$O(1)$	1452	$O(1)$	8456	$O(1)$	8456
String						
– `is_prefix`	$O(n)$	$4585n + 32$	$O(n^2)$	$64190n^2 - 11529n + 6545$	$O(n)$	$16758n - 42$
– `num_matching`	$O(n)$	$6052n + 5516$	$O(n^2)$	$84728n^2 + 129360n + 59710$	$O(n)$	$21826n + 18676$
– `compare`	$O(n)$	$4633n + 32$	$O(n^2)$	$97293n^2 + 10598n + 4781$	$O(n)$	$17773n - 42$
Set (radix tree)						
– `insert`	$O(d^2)$	$O(d^2)$ (App. E)	$O(d^3)$	$O(d^3)$ (Appendix E)	$O(d^2)$	$256914d^2 + 1413244d - 840$
– `contains`	$O(d^2)$	$O(d^2)$ (App. E)	$O(d^3)$	$O(d^3)$ (Appendix E)	$O(d^2)$	$134064d^2 + 687008d - 42$

8.2 RQ2: Effect of Program-Level Optimizations on *T*-Complexity

RQ2. By how much do the program-level optimizations of conditional flattening and conditional narrowing improve the T-complexity of a quantum program?

Methodology. For this question, we used Spire to execute each optimization on each benchmark program and found the empirical T-complexity by counting T gates in the same way as in RQ1.

Results. In Table 1, we present the T-complexity of each program after applying both optimizations. For each benchmark, the optimizations recover a program whose T-complexity is equal to the MCX-complexity, as determined both by the cost model and by circuit compilation.

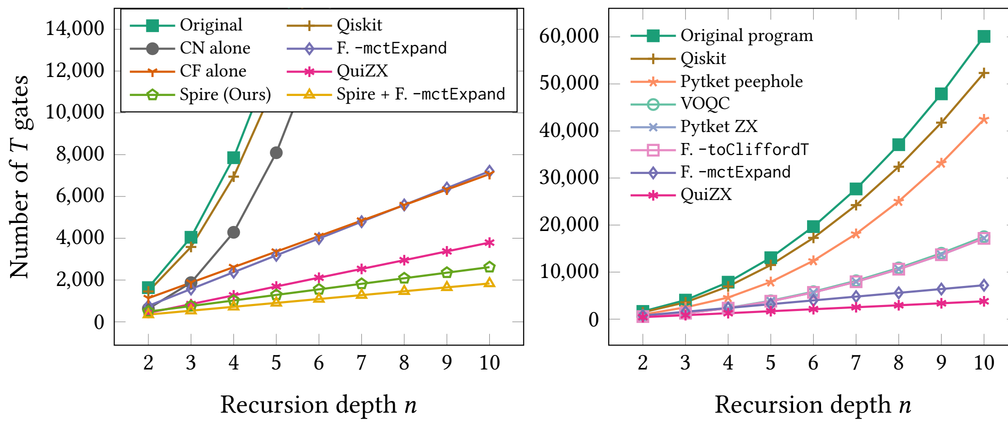
For `length` and `length-simplified`, the T-complexity improves from quadratic to linear. In Figure 15a, we plot the T-complexity of `length-simplified` after applying each of the optimizations in Spire. When used alone, conditional narrowing achieves 19.9% improvement over the original program at depth $n = 10$, and conditional flattening alone achieves 88.2% improvement. When Spire applies conditional narrowing on top of conditional flattening, conditional narrowing achieves a further 63.0% improvement, which stacks to 95.6% improvement end-to-end.

In Appendix F, we analyze the T-complexity that conditional flattening incurs due to its added uncomputation. Across all of the benchmarks in Table 1 at recursion depth $n = 10$, 0 to 4.81% (average 0.49%) of the T gates in the final compiled circuit correspond to the uncomputation that is introduced by conditional flattening. At depth $n = 2$, this figure is 0 to 2.85% (average 0.30%).

8.3 RQ3: Effect of Existing Circuit Optimizers on *T*-Complexity

RQ3. By how much do quantum circuit optimizers from existing work improve the T-complexity of a quantum program after it has been fully compiled to a circuit of logic gates?

Methodology. We evaluated the following optimizers: Qiskit [Qiskit Developers 2021], VOQC [Hietala et al. 2021], Pytket [Sivarajah et al. 2020], Feynman [Amy 2024; Amy et al. 2014], Quartz [Xu



(a) Program-level optimizations in Spire.

(b) Existing quantum circuit optimizers.

Fig. 15. T-complexity of `length-simplified` after program-level optimizations and quantum circuit optimizers. CF, CN, and F. abbreviate conditional flattening, conditional narrowing, and Feynman respectively.

et al. 2022], and QUESO [Xu et al. 2023]. We also evaluated QuiZX [QuiZX Developers 2022], a fast Rust port of PyZX [Kissinger and van de Wetering 2020] that produces outputs identical to PyZX.[5]

First, we used Spire to compile the `length-simplified` program to a MCX circuit. Notably, among the optimizers above, only Feynman directly accepts inputs containing MCX gates of arbitrary size, by means of a dedicated pass it provides to convert large MCX gates into Toffoli gates. By contrast, the other optimizers above do not accept MCX gates larger than Toffoli. For these optimizers, we used Feynman to preprocess the circuit into the Clifford+Toffoli or Clifford+CCZ gate sets accepted by each optimizer, without changing its T-complexity. Then, we executed each optimizer to generate a Clifford+T circuit, and then counted the T gates in the resulting circuit.

To the extent possible, we specified configurations that are indicated by prior literature:

- For Qiskit, we invoked `qiskit.compiler.transpile` with `optimization_level=3`.
- For VOQC, we invoked `Voqc.Main.optimize_nam`.
- For Pytket, we invoked two independent modes: `pytket.passes.FullPeepholeOptimise` and `pytket.passes.ZXGraphlikeOptimisation`, and report them separately below.
- For Feynman, we invoked two different configurations: `feynopt -mctExpand -O2` and `feynopt -toCliffordT -O2`, and report them separately below.
- For QuiZX, we invoked `quizx::simplify::full_simp`.

Results. In Figure 15b, we plot the T-complexity of the `length-simplified` program at various recursion depths, before and after applying each circuit optimizer. Of the tested optimizers, 6 of 8 do not asymptotically improve the T-complexity of the circuit from quadratic to linear. They achieve 0% to 71.4% improvement over the original circuit at depth $n = 10$.[6] Only Feynman `-mctExpand` and QuiZX obtain linear T-complexity, achieving 88.0% and 93.4% improvement respectively.

We do not plot Quartz and QUESO because the versions of these two optimizers available at the start of our experimentation require several hours to terminate for most of our benchmarks,

[5]As part of our evaluation, we ran PyZX for comparison with QuiZX, and observed that they produce circuits with identical T-complexity, though PyZX takes more time to produce the output. We thus do not report PyZX results separately.

[6]We note three results in Figure 15b that are close but distinct: at depth $n = 10$, VOQC obtains 17530 T gates, Pytket ZX obtains 17176 gates, and Feynman `-toCliffordT` obtains 17166 gates, which is about 2% fewer than VOQC.

even when the user specifies a 1-hour timeout. The partial results we obtained indicate that the T-complexity of their output circuits is quadratic rather than linear. At depth $n = 5$, Quartz achieves 37% improvement in T-complexity, and at $n = 2$, QUESO achieves 13% improvement. For more details on the methodology and results for these optimizers, please see Appendix G.

Notably, only select configurations of Feynman obtain asymptotic improvement in T-complexity. In Figure 15b, we plot the T-complexity Feynman obtains using two different flags: `-toCliffordT`, which is quadratic, and `-mctExpand`, which is linear. The difference is that the first configuration translates the circuit to the Clifford+T gate set before applying gate simplifications, whereas the second simplifies the original circuit in terms of Toffoli gates before translating to Clifford+T.

Spire's program-level optimizations also synergize with existing quantum circuit optimizers to achieve better results than either alone. In Figure 15a, we also plot the T-complexity of applying Spire's optimizations followed by Feynman `-mctExpand`, which for `length-simplified` achieves 96.9% improvement over the original program compared to 88.0% for Feynman alone. In Table 2, we summarize the T-complexity improvement of running either Feynman `-mctExpand` or QuiZX after Spire's optimizations. The latter achieves 98.1% improvement compared to 93.4% for QuiZX alone.

In Appendix H, we present more results showing that when the conditional narrowing optimization is used before Feynman or QuiZX, the output circuits are better than Feynman or QuiZX alone. These results indicate that even when a circuit optimizer achieves asymptotically efficient circuits, it can still benefit from the constant improvements provided by conditional narrowing.

8.4 RQ4: Effect of Optimizations on Compilation Time

RQ4. What is the effect on compilation time of performing the program-level optimizations, and how does it compare to the effect on compilation time of quantum circuit optimizers?

Methodology. To answer this question, we measured the time taken by Spire to emit a circuit for both the `length` and `length-simplified` programs, with program-level optimizations enabled or disabled. Then, we measured the time taken by Feynman `-mctExpand` and QuiZX to optimize the circuit emitted by Spire with optimizations enabled or disabled. All timings are reported as the mean and standard error of 5 runs on one core of an AMD Threadripper 1920X and 32 GB of RAM.

Results. Given the original `length` program at depth $n = 10$, Spire takes 0.08 s to emit a circuit without performing program-level optimizations, and 0.05 s with the optimizations. The reason that compilation time decreases is that while the optimizations take tens of microseconds to perform, they enable the compiler to save significant time generating controls in the output circuit.

In Table 2, we summarize the performance of Feynman `-mctExpand` and QuiZX on each circuit. When executed on the original `length` circuit, Feynman takes 121.96 ± 0.08 s; QuiZX exceeds available memory and does not terminate after 72 hours. By comparison, Spire alone yields comparable circuits in 0.05 s, which is 2400× faster than Feynman. When Spire's optimizations are run before Feynman, the smaller input circuit that is produced enables Feynman to take only 17.05 ± 0.01 s, a 7× improvement. These circuits remain large enough for QuiZX to be memory constrained.

8.5 Discussion

First, our results indicate that VOQC, Quartz, Pytket ZX, and Feynman `-toCliffordT` obtain an intermediate result in T-complexity that is higher than Feynman `-mctExpand` and QuiZX and lower than Qiskit and Pytket peephole. An explanation consistent with these results is that the first four optimizers implement the optimization of rotation merging [Nam et al. 2018] that merges phase rotations across an arbitrary number of gates, whereas the last two do not.

Next, one explanation for why Feynman `-mctExpand` and QuiZX reduce the asymptotic T-complexity of the program in our results is that they successfully identify and exploit the structure

Table 2. Summary of comparison and synergy between Spire and existing circuit optimizers, in terms of T-complexity reduction and compilation time. Figures are given for both `length` and `length-simplified` programs at depth $n = 10$. We show only optimizers that achieve linear T-complexity.

	`length-simplified`		`length`	
	T Reduction	Compile Time	T Reduction	Compile Time
Feynman `-mctExpand`	88.0%	0.54 s	92.2%	121.96 ± 0.08 s
QuiZX	93.4%	3510.80 ± 1.97 s	(consumes >32 GB RAM)	
Spire (Ours)	95.6%	**0.01** s	92.8%	**0.05** s
Spire + Feynman `-mctExpand`	96.9%	0.08 s	**94.7%**	17.05 ± 0.01 s
Spire + QuiZX	**98.1%**	1.18 s	(consumes >32 GB RAM)	

of Toffoli gates. Specifically, Feynman `-mctExpand` first cancels Toffoli gates in the circuit before translating them to Clifford+T gates. Meanwhile, QuiZX uses an internal representation known as ZX-calculus [Kissinger and van de Wetering 2020] that discovers long-range circuit structure at the expense of compile time, which in Table 2 is 14×–6500× longer than Feynman.[7]

By contrast, Qiskit, Pytket, VOQC, Quartz, and QUESO do not perform rewrites at the level of Toffoli gates. They instead either require the input to consist only of Clifford+T gates, or decompose all Toffoli gates in the input to them. As we show next, the value of the structure of Toffoli gates is that cancelling Toffoli gates can capture the effect of conditional flattening. By contrast, cancelling adjacent gates no longer captures this effect after Toffoli gates are lowered to Clifford+T gates.

Conditional Flattening. In Figure 16, we present a sub-program of Figure 3 with only the assignment to `a` that is controlled by `x`, `y`, and `z`, and its corresponding sub-circuit from Figure 4. We also depict the result of decomposing its MCX gates to Toffoli gates via the rule in Figure 5. Compared to Figure 8, the final circuit in Figure 16 incurs additional T-complexity from Toffoli gates.

Now suppose that the final circuit in Figure 16 is given to a quantum circuit optimizer. In general, to recover an asymptotically efficient circuit, the optimizer must eliminate all but a small number of Toffoli gates. In Figure 16, it must eliminate the redundant self-inverse gates in gray.

The problem is that adjacent Toffoli gates become difficult to identify when Toffoli gates have been decomposed into Clifford+T gates. In Figure 17, we depict the decomposition of a pair of Toffoli gates into a sequence of 32 Clifford+T gates by the standard rule in Figure 6. Because the decomposition of each Toffoli is asymmetric, the circuit optimizer cannot reduce this sequence to an empty circuit by merely cancelling adjacent Clifford+T gates.[8] To sidestep this problem, Feynman and Maslov et al. [2005]; Nam et al. [2018] perform rewrites on Toffoli gates before they are decomposed to Clifford+T gates. For other alternative approaches, see Appendix I.

Conditional Narrowing. Even worse, conditional narrowing cannot be captured by a quantum circuit optimizer that acts on gate windows of any finite size. The rule in Figure 14b removes control bits on $C[\![s_1]\!]$ and $C[\![s_1]\!]^\dagger$ when these sequences have been identified as inverses. The problem is that $C[\![s_2]\!]$ lies between them and can be of arbitrary length, meaning that without program structure, discovering the relationship between $C[\![s_1]\!]$ and $C[\![s_1]\!]^\dagger$ requires a window of unbounded size.

This fact contributes to an explanation for why in Section 8.3, some of the tested circuit optimizers leave behind some fraction of T gates that are otherwise captured by the conditional narrowing

[7]We note that Pytket ZX does not reduce the asymptotic T-complexity in our results even though it also uses the ZX-calculus. This discrepancy may be due to different optimization choices taken by Pytket and QuiZX.

[8]In particular, a GitHub issue open since 2021 (https://github.com/Qiskit/qiskit/issues/6740) describes the inability of Qiskit to optimize away the Clifford+T sequence corresponding to adjacent Toffoli gates as depicted in Figure 17.

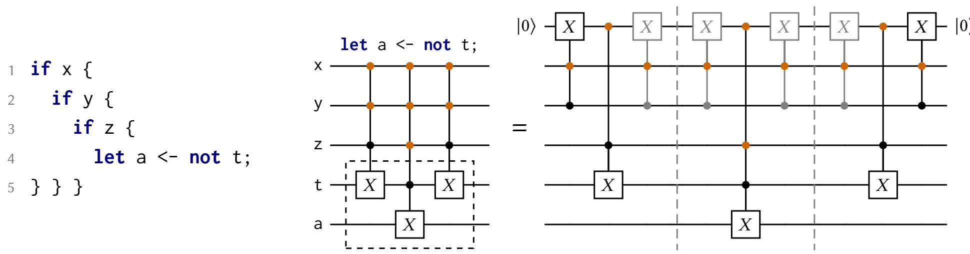


Fig. 16. Direct compilation of nested conditionals to a Clifford+Toffoli circuit using the MCX decomposition in Figure 5. The redundant Toffoli gates (gray) must be eliminated to obtain an efficient circuit.

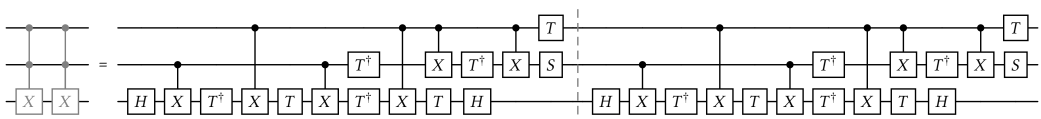


Fig. 17. Two adjacent Toffoli gates after the standard Clifford+T decomposition in Figure 6. Though equal to the empty identity circuit, this gate sequence cannot be reduced to such by adjacent gate cancellations alone.

optimization. In principle, a circuit optimizer can capture conditional narrowing using rewrites over an unbounded number of gates, such as by an appropriate implementation of rotation merging.

9 FUTURE DIRECTIONS

Benchmarking Optimizers. One explanation for why certain existing circuit optimizers do not asymptotically reduce the T-complexity of programs with control flow is that such programs are not a focus of the benchmarks on which optimizers have been primarily evaluated. The typical benchmarks, as in Xu et al. [2023, Appendix F], are circuits with up to 10^3 T gates that are built directly from logic gates, not compiled from quantum programs with control flow. This work does not evaluate on these benchmarks as they do not exhibit the asymptotic behavior of interest.

Instead, this work studies the asymptotic behavior of families of circuits that are compiled from programming abstractions and large enough to be relevant to the regime of practical quantum advantage. For example, Gidney and Ekerå [2021] project that $4 \cdot 10^8$ Toffoli gates are necessary to break 1024-bit RSA, and $3 \cdot 10^{10}$ Toffoli gates to break elliptic curve cryptography. At such scales, optimization techniques that are profitable and tractable for small circuits, such as small peepholes and ZX-calculus, become less effective and would benefit from higher-level program structure.

Consequently, it is important future work to develop more explicit implementations of quantum algorithms to serve as large-scale benchmarks for quantum compilation that may reveal other quantum programming abstractions whose costs must be considered and mitigated.

Architectural Bottlenecks. Aside from T-complexity, error-corrected architectures are also constrained by the number of qubits used by a computation. Conditional narrowing does not affect qubit usage, as it only removes control bits from statements. In Appendix F, we show that given a compiler that uses the MCX decomposition in Figures 5 and 6, conditional flattening introduces no more than $O(1)$ extra qubits in the circuit for the optimized program as compared to the unoptimized program. The reason is that the new temporary variable x' from the rule in Section 6.1 reuses a qubit that would exist in the compiled circuit for the program even without conditional flattening. This extra qubit – marked with $|0\rangle$ in Figure 5 – is introduced when the compiler decomposes all MCX to Clifford+T gates as needed for a program regardless of conditional flattening.

For sake of thoroughness, we note that alternatives to Figure 5 exist that use no extra qubits but use more T gates [Barenco et al. 1995, Section 7]. An important future direction is to explore the trade-offs of different MCX decompositions, and simultaneously optimize T-complexity alongside qubit complexity and other metrics such as *T-depth* and *quantum volume* [Cross et al. 2019].

Though this work focuses on the widely recognized bottleneck of T-complexity on the surface code architecture, the asymptotic costs it presents arise on any error-corrected quantum computer. Fundamentally, the Eastin-Knill theorem [Eastin and Knill 2009] states that no quantum error-correcting code can *transversely*, i.e. natively and efficiently, implement a gate set that is universal for quantum computation. Some gate – in the surface code, the T gate – is always a bottleneck.

For example, while Reed-Muller codes support an efficient T gate, they give up the Hadamard gate in exchange and are thus not universal for quantum computation [Zeng et al. 2011]. In general, strong evidence [Jochym-O'Connor et al. 2018; Newman and Shi 2018] indicates that a Toffoli or MCX gate will act as a performance bottleneck under any quantum error-correcting code.

Other Quantum Architectures. Apart from the surface code, the abstraction cost of control flow also occurs broadly on hardware architectures in which MCX gates must be decomposed to native gates. For example, on an architecture with only single and two-qubit gates such as CNOT, an MCX gate with many control bits compiles to a proportional number of CNOT gates, making it important to study further how to reduce the performance impact of two-qubit gates [Maslov 2016b].

10 RELATED WORK

T-Complexity Optimization. Optimizations for T-complexity have long been investigated in the literature of quantum algorithms. For example, instances of conditional narrowing and conditional flattening are used by physical simulation algorithms [Babbush et al. 2018, Figures 1, 6, and 7] to save control bits during state preparation and Hamiltonian selection respectively.

Researchers have proposed quantum compilers featuring variants of conditional narrowing [Ittah et al. 2022; Steiger et al. 2018] and separately of conditional flattening [Seidel et al. 2022]. Novel to this work is our unification of both optimizations as syntax rewrite rules, which produce high-level programs that can be analyzed by the cost model. Other novel contributions in this work are that we identify that these optimizations can mitigate the asymptotic slowdown caused by control flow, and empirically evaluate their effectiveness and speed relative to existing circuit optimizers.

Quantum Resource Analysis. Researchers have proposed frameworks [Avanzini et al. 2022; Liu et al. 2022; Olmedo and Díaz-Caro 2019] to analyze the expected runtime of a quantum program. Unlike our cost model, prior frameworks do not support reasoning for abstractions for control flow in superposition such as the quantum `if`-statement. In order to analyze a program featuring control flow, they require the developer to first lower all abstractions to explicit quantum logic gates.

However, as identified in this work, it is precisely this compilation process itself that introduces asymptotic overhead in T-complexity. Our cost model and optimizations enable the developer to identify and mitigate the costs without compiling the program to an asymptotically large circuit.

11 CONCLUSION

The practical realization of quantum algorithms requires designers of programming languages and compilers to reconcile the expressive power of programming abstractions with the performance bottlenecks of error correction. As this work shows, control flow incurs T-complexity costs that are significant yet can be mitigated by simple optimizations. Our work holds out the promise of enabling both expressive and efficient control flow abstractions in quantum programming.

Our work additionally demonstrates the value of a deep study of the interface between quantum programs and error-corrected hardware. This study and our results illuminate a path to a future that combines powerful techniques from classical compilers with search-based optimization of circuits to increase the efficiency of both current and future quantum software.

DATA AVAILABILITY STATEMENT

The artifact for this paper, including source code, benchmark programs, and evaluation package, is available on Zenodo [Yuan and Carbin 2024].

ACKNOWLEDGEMENTS

We thank Ellie Cheng, Tian Jin, Jesse Michel, Patrick Rall, Alex Renda, Agnes Villanyi, Logan Weber, and Cambridge Yang for helpful feedback on this work. This work was supported in part by the National Science Foundation (CCF-1751011) and the Sloan Foundation.

REFERENCES

Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. 2020. On the Quantum Complexity of Closest Pair and Related Problems. In *Computational Complexity*. https://doi.org/10.4230/LIPIcs.CCC.2020.16

Daniel S. Abrams and Seth Lloyd. 1997. Simulation of Many-Body Fermi Systems on a Universal Quantum Computer. *Phys. Rev. Letters* 79 (Sep 1997). Issue 13. https://doi.org/10.1103/PhysRevLett.79.2586

Thorsten Altenkirch and J. Grattage. 2005. A Functional Quantum Programming Language. In *ACM/IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1109/LICS.2005.1

Andris Ambainis. 2004. Quantum walk algorithm for element distinctness. In *IEEE Symposium on Foundations of Computer Science*. https://doi.org/10.1109/FOCS.2004.54

Andris Ambainis, A. M. Childs, B. W. Reichardt, R. Špalek, and S. Zhang. 2010. Any AND-OR Formula of Size N Can Be Evaluated in Time $N^{1/2+o(1)}$ on a Quantum Computer. *SIAM J. Comput.* 39, 6 (2010). https://doi.org/10.1137/080712167

Matthew Amy. 2024. Feynman: Quantum circuit analysis toolkit. https://github.com/meamy/feynman.

Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014). https://doi.org/10.1109/TCAD.2014.2341953

Martin Avanzini, Georg Moser, Romain Pechoux, Simon Perdrix, and Vladimir Zamdzhiev. 2022. Quantum Expectation Transformers for Cost Analysis. In *ACM/IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1145/3531130.3533332

Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. 2018. Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity. *Phys. Rev. X* 8, 4 (Oct 2018). https://doi.org/10.1103/PhysRevX.8.041015

Ryan Babbush, Jarrod R. McClean, Michael Newman, Craig Gidney, Sergio Boixo, and Hartmut Neven. 2021. Focus beyond Quadratic Speedups for Error-Corrected Quantum Advantage. *PRX Quantum* 2 (Mar 2021). Issue 1. https://doi.org/10.1103/PRXQuantum.2.010103

Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Phys. Rev. A* 52, 5 (Nov 1995). https://doi.org/10.1103/physreva.52.3457

J. S. Bell. 1964. On the Einstein Podolsky Rosen paradox. *Physics* 1 (Nov 1964). Issue 3. https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195

Charles H. Bennett. 1973. Logical Reversibility of Computation. *IBM Journal of Research and Development* 17, 6 (1973). https://doi.org/10.1147/rd.176.0525

Charles H. Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* 560 (2014). https://doi.org/10.1016/j.tcs.2014.05.025

Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Letters* 70 (Mar 1993). Issue 13. https://doi.org/10.1103/PhysRevLett.70.1895

Daniel J. Bernstein, Stacey Jeffery, Tanja Lange, and Alexander Meurer. 2013. Quantum Algorithms for the Subset-Sum Problem. In *Post-Quantum Cryptography*. https://doi.org/10.1007/978-3-642-38616-9_2

Michael Beverland, Earl Campbell, Mark Howard, and Vadym Kliuchnikov. 2020. Lower bounds on the non-Clifford resources for quantum computations. *Quantum Science and Technology* 5, 3 (May 2020). https://doi.org/10.1088/2058-9565/ab8963

Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum Machine Learning. *Nature* 549, 7671 (Sep 2017). https://doi.org/10.1038/nature23474

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386007

Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. 2002. Quantum Amplitude Amplification and Estimation. In *Quantum Computation and Information*. Vol. 305. https://doi.org/10.1090/conm/305/05215

Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Phys. Rev. A* 71, 2 (Feb 2005). https://doi.org/10.1103/physreva.71.022316

Sergey Bravyi, Ruslan Shaydulin, Shaohan Hu, and Dmitri Maslov. 2021. Clifford Circuit Optimization with Templates and Symbolic Pauli Gates. *Quantum* 5 (Nov. 2021). https://doi.org/10.22331/q-2021-11-16-580

Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (Sep 2018). https://doi.org/10.1073/pnas.1801723115

Andrew M. Childs, Ben W. Reichardt, Robert Spalek, and Shengyu Zhang. 2007. Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer. https://doi.org/10.48550/ARXIV.QUANT-PH/0703015 arXiv:0703015 [quant-ph]

Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. 2019. Validating quantum computers using randomized model circuits. *Phys. Rev. A* 100 (Sep 2019). Issue 3. https://doi.org/10.1103/PhysRevA.100.032328

Bryan Eastin and Emanuel Knill. 2009. Restrictions on Transversal Encoded Quantum Gate Sets. *Phys. Rev. Letters* 102, 11 (Mar 2009). https://doi.org/10.1103/physrevlett.102.110502

Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph]

Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A* 86, 3 (Sep 2012). https://doi.org/10.1103/PhysRevA.86.032324

Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (Apr 2021), 433. https://doi.org/10.22331/q-2021-04-15-433

Craig Gidney and Austin G. Fowler. 2019. Efficient magic state factories with a catalyzed $|CCZ\rangle$ to $2|T\rangle$ transformation. *Quantum* 3 (April 2019). https://doi.org/10.22331/q-2019-04-30-135

Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614 (02 2023). https://doi.org/10.1038/s41586-022-05434-1

Daniel Gottesman. 1998. Theory of fault-tolerant quantum computation. *Phys. Rev. A* 57, 1 (Jan 1998). https://doi.org/10.1103/physreva.57.127

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2491956.2462177

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/237814.237866

Julian Hans and Sven Groppe. 2022. Silq2Qiskit - Developing a Quantum Language Source-to-Source Translator. In *International Conference on Computer Science and Software Engineering*. https://doi.org/10.1145/3569966.3570114

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3434318

Torsten Hoefler, Thomas Häner, and Matthias Troyer. 2023. Disentangling Hype from Practicality: On Realistically Achieving Quantum Advantage. *Commun. ACM* 66, 5 (Apr 2023). https://doi.org/10.1145/3571725

David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-Based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3, Article 14 (Jun 2022). https://doi.org/10.1145/3491247

Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *ACM Conference on Computing Frontiers*. https://doi.org/10.1145/2597917.2597939

Tomas Jochym-O'Connor, Aleksander Kubica, and Theodore J. Yoder. 2018. Disjointness of Stabilizer Codes and Limitations on Fault-Tolerant Logical Gates. *Phys. Rev. X* 8 (May 2018). Issue 2. https://doi.org/10.1103/PhysRevX.8.021047

Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. In *International Conference on Quantum Physics and Logic*. https://doi.org/10.4204/eptcs.318.14

Joonho Lee, Dominic W. Berry, Craig Gidney, William J. Huggins, Jarrod R. McClean, Nathan Wiebe, and Ryan Babbush. 2021. Even More Efficient Quantum Computations of Chemistry Through Tensor Hypercontraction. *PRX Quantum* 2, 3 (Jul 2021). https://doi.org/10.1103/prxquantum.2.030305

Frank Leymann and Johanna Barzen. 2020. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology* 5, 4 (Sep 2020). https://doi.org/10.1088/2058-9565/abae7d

Junyi Liu, Li Zhou, Gilles Barthe, and Mingsheng Ying. 2022. Quantum Weakest Preconditions for Reasoning about Expected Runtimes of Quantum Programs. In *ACM/IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1145/3531130.3533327

Seth Lloyd, Silvano Garnerone, and Paolo Zanardi. 2014. Quantum algorithms for topological and geometric analysis of big data. *Nature Communications* 7, 10138 (Aug 2014). https://doi.org/10.1038/ncomms10138

Guang Hao Low, Vadym Kliuchnikov, and Luke Schaeffer. 2018. Trading T-gates for dirty qubits in state preparation and unitary synthesis. https://doi.org/10.48550/arXiv.1812.00954 arXiv:1812.00954 [quant-ph]

Dmitri Maslov. 2016a. Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. *Phys. Rev. A* 93 (Feb 2016). Issue 2. https://doi.org/10.1103/PhysRevA.93.022311

Dmitri Maslov. 2016b. Optimal and asymptotically optimal NCT reversible circuits by the gate types. *Quantum Information and Computation* 16, 13-14 (Oct 2016).

D. Maslov, G.W. Dueck, and D.M. Miller. 2005. Toffoli network synthesis with templates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 6 (2005). https://doi.org/10.1109/TCAD.2005.847911

Michele Mosca. 2016. Specification of the .qc file format. https://circuits.qsoft.iqc.uwaterloo.ca/about/spec/.

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 (May 2018). https://doi.org/10.1038/s41534-018-0072-4

Michael Newman and Yaoyun Shi. 2018. Limitations on Transversal Computation through Quantum Homomorphic Encryption. *Quantum Information and Computation* 18, 11-12 (Sep 2018). https://doi.org/10.26421/QIC18.11-12-3

Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* https://doi.org/10.1017/CBO9780511976667

Federico Olmedo and Alejandro Díaz-Caro. 2019. Runtime Analysis of Quantum Programs: A Formal Approach. https://doi.org/10.48550/arXiv.1911.11247 arXiv:1911.11247 [cs.LO]

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3009837.3009894

John Proos and Christof Zalka. 2003. Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves. *Quantum Information and Computation* 3, 4 (Jul 2003). https://doi.org/10.26421/QIC3.4-3

Qiskit Developers. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

QuiZX Developers. 2022. QuiZX: a quick Rust port of PyZX. https://github.com/Quantomatic/quizx.

Patrick Rebentrost, Brajesh Gupt, and Thomas R. Bromley. 2018. Quantum computational finance: Monte Carlo pricing of financial derivatives. *Phys. Rev. A* 98, 2 (Aug 2018). https://doi.org/10.1103/physreva.98.022321

Markus Reiher, Nathan Wiebe, Krysta M. Svore, Dave Wecker, and Matthias Troyer. 2017. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences* 114, 29 (Jul 2017). https://doi.org/10.1073/pnas.1619152114

Rich Rines and Isaac Chuang. 2018. High Performance Quantum Modular Multipliers. https://doi.org/10.48550/arXiv.1801.01081 arXiv:1801.01081 [quant-ph]

Yuval R. Sanders, Dominic W. Berry, Pedro C.S. Costa, Louis W. Tessler, Nathan Wiebe, Craig Gidney, Hartmut Neven, and Ryan Babbush. 2020. Compilation of Fault-Tolerant Quantum Heuristics for Combinatorial Optimization. *PRX Quantum* 1, 2 (Nov 2020). https://doi.org/10.1103/prxquantum.1.020312

Raphael Seidel, Sebastian Bock, Nikolay Tcholtchev, and Manfred Hauswirth. 2022. Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers. In *International Workshop on Programming Languages for Quantum Computing.*

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14 (Aug 2004). https://doi.org/10.1017/S0960129504004256

Peter Selinger. 2013. Quantum circuits of T-depth one. *Phys. Rev. A* 87 (Apr 2013). Issue 4. https://doi.org/10.1103/PhysRevA.87.042302

Yaoyun Shi. 2003. Both Toffoli and Controlled-NOT Need Little Help to Do Universal Quantum Computing. *Quantum Information and Computation* 3, 1 (Jan 2003). https://doi.org/10.26421/QIC3.1-7

Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct 1997). https://doi.org/10.1137/S0097539795293172

Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (2020). https://doi.org/10.1088/2058-9565/ab8e92

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

Martin Suchara, John Kubiatowicz, Arvin Faruque, Frederic T. Chong, Ching-Yi Lai, and Gerardo Paz. 2013. QuRE: The Quantum Resource Estimator toolbox. In *IEEE International Conference on Computer Design.* https://doi.org/10.1109/ICCD.2013.6657074

Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Real World Domain Specific Languages Workshop.* https://doi.org/10.1145/3183895.3183901

Maika Takita, A. D. Córcoles, Easwar Magesan, Baleegh Abdo, Markus Brink, Andrew Cross, Jerry M. Chow, and Jay M. Gambetta. 2016. Demonstration of Weight-Four Parity Measurements in the Surface Code Architecture. *Phys. Rev. Letters* 117 (Nov 2016). Issue 21. https://doi.org/10.1103/PhysRevLett.117.210505

Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3571225

Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/3591254

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/3519939.3523433

Mingsheng Ying, Nengkun Yu, and Yuan Feng. 2012. Defining Quantum Control Flow. https://doi.org/10.48550/ARXIV.1209.4379 arXiv:1209.4379 [quant-ph]

Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* https://doi.org/10.1145/3563297

Charles Yuan and Michael Carbin. 2024. *The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation.* https://doi.org/10.5281/zenodo.10729070

Bei Zeng, Andrew Cross, and Isaac L. Chuang. 2011. Transversality Versus Universality for Additive Quantum Codes. *IEEE Transactions on Information Theory* 57, 9 (2011). https://doi.org/10.1109/TIT.2011.2161917

Received 2023-11-16; accepted 2024-03-31