QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization

DAVID ITTAH, ETH Zurich
THOMAS HÄNER and VADYM KLIUCHNIKOV, Microsoft Quantum
TORSTEN HOEFLER, ETH Zurich

We propose an IR for quantum computing that directly exposes quantum and classical data dependencies for the purpose of optimization. The ***Quantum Intermediate Representation for Optimization*** **(QIRO)** consists of two dialects, one input dialect and one that is specifically tailored to enable quantum-classical co-optimization. While the first employs a perhaps more intuitive memory-semantics (quantum operations act on qubits via side-effects), the latter uses value-semantics (operations consume and produce states) to integrate quantum dataflow in the IR's **Static Single Assignment (SSA)** graph. Crucially, this allows for a host of optimizations that leverage dataflow analysis. We discuss how to map existing quantum programming languages to the input dialect and how to lower the resulting IR to the optimization dialect. We present a prototype implementation based on MLIR that includes several quantum-specific optimization passes. Our benchmarks show that significant improvements in resource requirements are possible even through static optimization. In contrast to circuit optimization at run time, this is achieved while incurring only a small constant overhead in compilation time, making this a compelling approach for quantum program optimization at application scale.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Quantum computing;**

Additional Key Words and Phrases: Quantum compilation, dataflow optimization, intermediate representation, MLIR

ACM Reference format:
David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Trans. Quantum Comput.* 3, 3, Article 14 (June 2022), 32 pages.
https://doi.org/10.1145/3491247

1 INTRODUCTION

In recent years, the quantum programming landscape has seen a boom of new languages, tools, and environments [5, 6, 9, 12, 17–19, 27, 30–32]. The focus of these projects varies widely, ranging from low-level interfaces for prototype hardware [9, 30] to high-level algorithm development [5, 12, 32]. When tasked with supporting common components of the compilation stack, these projects have

Authors' addresses: D. Ittah and T. Hoefler, ETH Zurich, Raemistrasse 101, 8092 Zurich, Switzerland; emails: david.ittah@mail.mcgill.ca, htor@inf.ethz.ch; T. Häner and V. Kliuchnikov, Microsoft Quantum, 1 Microsoft Way, Redmond, WA 98052, USA; emails: thhaner@microsoft.com, v.kliuchnikov@gmail.com.

2643-6817/2022/06-ART14 $15.00
https://doi.org/10.1145/3491247

opted for one of two approaches; either (1) a complete (re-)implementation of these components or (2) re-use of existing infrastructure by embedding the domain-specific language in a widely-used programming language such as Python. Indeed, most quantum programming languages that target **Noisy Intermediate-Scale Quantum (NISQ)** [28] hardware are embedded in a classical programming language. Such **embedded domain-specific languages (eDSLs)** can be viewed as libraries that generate a data structure representing a quantum circuit. These data structures can be seen as very simple and flat **intermediate representations (IRs)** without any control flow.

In light of the large number of quantum gates required to achieve practical quantum speedups, however, it is clear that a more advanced IR is needed to support large-scale algorithms. Indeed, many chemistry applications of practical interest require between 10^9 and 10^{15} gates [29, 33] and applications in the domain of cryptography similarly lead to programs with 10^{10} gates [11, 14]. Consequently using flat data structures such as lists of gates[1] or **directed acyclic graphs (DAGs)** to represent quantum programs is infeasible at application scale. Instead, a special purpose IR that incorporates classical and quantum control flow is needed.

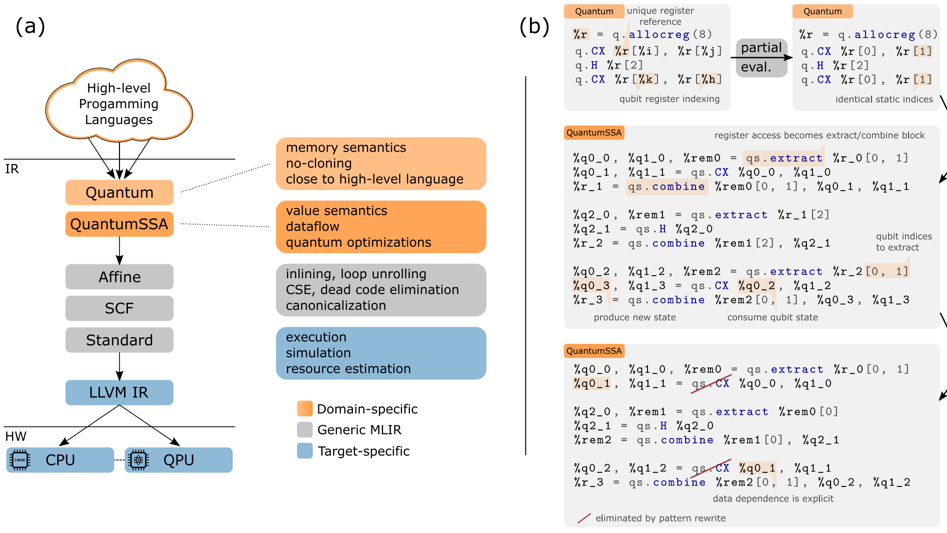
Designing a more advanced IR for quantum programs poses some unique challenges not present in classical computing. Most notably, "values" held in qubits cannot be copied due to the no-cloning theorem of quantum mechanics [25, Box 12.1]. Existing quantum IRs such as the recently proposed QIR [10] therefore opt to only represent *references* to quantum data, while operations on qubits are modeled via side-effects. Unfortunately, such a representation severely limits reuse of existing compiler components, which often rely on the dataflow to be explicit in the IR. As a remedy, we introduce a quantum-analog of SSA[2] where dataflow is explicit.

The no-cloning theorem might also seem to rule out a large set of classical program optimizations that rely on the duplication of values across the program such as **common sub-expression elimination (CSE)**. However, such techniques are useful for optimization of mixed quantum-classical programs, as illustrated by the example in Figure 1(b): While two successive CNOT (commonly CX) gates may be eliminated from the IR if they are applied to the same qubits, it is difficult to carry out such an optimization in the presence of dynamic accesses to quantum registers, especially if dataflow is not explicit in the IR. However, if existing optimization passes such as partial evaluation succeed at inferring, e.g., that `i=k=0` and `j=h=1`, then the two CNOTs may be removed by leveraging our optimization dialect, which directly exposes (quantum and classical) dataflow. Classical program optimization thus immediately increases the usefulness of quantum-specific optimization passes.

1.1 Quantum Multi-Level IR

In this work, we introduce the ***Quantum Intermediate Representation for Optimization* (QIRO)**, an IR for universal quantum computation that leverages MLIR [21] to support quantum-classical co-optimization. In contrast to existing IRs for quantum computing, we design our optimization dialect in a way such that data dependencies are explicit for both quantum and classical variables. This enables the use of existing infrastructure as well as the development of future quantum-specific optimization passes that leverage dataflow analysis. Figure 1(b) depicts how a code fragment is translated from our input dialect (*Quantum*) to the optimization dialect (*QuantumSSA*). An IR-specific optimization transforms the code in the middle section to that of the bottom section by consolidating `combine` and `extract` instructions, which are used to represent quantum register accesses in the optimization dialect. Optimizing such patterns in the IR enables

[1]Representing a quantum program as a flat list of gates is similar to representing a classical program as a list of ALU instructions without any control flow.
[2]Static Single Assignment (SSA) is an IR property where each variable is assigned exactly once, see Section 2.1.



```
%r = q.allocreg(8)
q.CX %r[%i], %r[%j]
q.H %r[2]
q.CX %r[%k], %r[%h]
```

```
%r = q.allocreg(8)
q.CX %r[0], %r[1]
q.H %r[2]
q.CX %r[0], %r[1]
```

```
%q0_0, %q1_0, %rem0 = qs.extract %r_0[0, 1]
%q0_1, %q1_1 = qs.CX %q0_0, %q1_0
%r_1 = qs.combine %rem0[0, 1], %q0_1, %q1_1

%q2_0, %rem1 = qs.extract %r_1[2]
%q2_1 = qs.H %q2_0
%r_2 = qs.combine %rem1[2], %q2_1

%q0_2, %q1_2, %rem2 = qs.extract %r_2[0, 1]
%q0_3, %q1_3 = qs.CX %q0_2, %q1_2
%r_3 = qs.combine %rem2[0, 1], %q0_3, %q1_3
```

```
%q0_0, %q1_0, %rem0 = qs.extract %r_0[0, 1]
%q0_1, %q1_1 = qs.CX %q0_0, %q1_0

%q2_0, %rem1 = qs.extract %rem0[0]
%q2_1 = qs.H %q2_0
%rem2 = qs.combine %rem1[0], %q2_1

%q0_2, %q1_2 = qs.CX %q0_1, %q1_1
%r_3 = qs.combine %rem2[0, 1], %q0_2, %q1_2
```

Fig. 1. (a) Proposed compilation stack incorporating our IR infrastructure, along with the features and main concepts of the different levels. (b) In the input dialect (Quantum), operations act on unique qubit references via side-effects (memory-semantics). We designed our optimization dialect to facilitate dataflow analysis and to maximally leverage existing optimization infrastructure. Therefore, operations in the optimization dialect (QuantumSSA) consume and return quantum states (value-semantics) and quantum register accesses are translated to pairs of extract/combine instructions.

subsequent optimizations (such as the CX-CX elimination) by exposing quantum data dependencies inside registers.

The use of MLIR presents significant benefits over a single-level IR such as LLVM. For example, our approach leverages several application-specific abstractions from MLIR such as classical (low-level) computation, domain-specific quantum computation, and specialized affine loop representations. Furthermore, language or domain-specific optimizations are comparatively difficult to implement in LLVM due to a lack of high-level information. LLVM's rigidness implies that domain-specific operations must be represented by opaque functions, and additional types by opaque pointers (the approach taken by QIR, see Section 7). In contrast, MLIR provides multiple abstractions via which to interact with the IR on a high level, even across application domains. This includes, for instance, extensible types that can be mixed and matched, operation attributes for compile-time information, and operation traits for the reuse of passes in an extensible system (see Section 2).

Finally, MLIR features an extensive validation mechanism for operations, types, and traits. We leverage this mechanism to express invariants of the IR and to statically enforce constraints on quantum operations wherever possible.

1.2 Contributions

We design and implement QIRO, a novel IR for quantum-classical co-optimization in MLIR. Our dual dialect approach guarantees simple translation from high-level quantum programming languages as well as broad re-use of existing compilation infrastructure for optimization.

In short, our main contributions are:

- We propose an optimization dialect, which can be seen as a quantum-analog of SSA. The optimization dialect (1) is compatible with the no-cloning theorem and (2) allows for reuse of existing compiler components (quantum and classical dataflow is explicit in the IR).

- We define a higher-level IR that serves as an input dialect with the same semantics that are commonly used for quantum IRs, so-called *memory-semantics* (quantum operations are modeled as side-effects).
- We leverage existing work (MLIR) to implement the two proposed dialects, including the lowering from the input dialect to the optimization dialect.
- We describe how a variety of existing quantum programming languages can be mapped to our input dialect.
- We show that our IR enables optimization and resource estimation at application-scale up to 5-6 orders of magnitude faster than existing frameworks that rely on optimization at run time such as Qiskit [9] and ProjectQ [31].
- We implement several optimization passes that aim to reduce the quantum resource requirements (operations and qubits). We demonstrate that for Shor's algorithm, practically all (~99.8%) savings identified by ProjectQ's run-time circuit optimization may be obtained statically, at significantly lower cost in terms of optimizer run time.

By directly exposing quantum and classical dataflow, QIRO enables future quantum and quantum-classical optimizations that make use of this information. Combined with its advantages in terms of compilation time, our IR may serve as a helpful tool for resource estimation of optimized quantum-classical programs. In turn, this allows for more efficient hardware-software co-design and for achieving a quantum advantage for real-world problems once fault-tolerant quantum computers become available.

2 BACKGROUND

This section provides a short introduction to classical IRs, compiler optimization, and quantum computing. For more in-depth treatments of these subjects, we refer the reader to the textbooks by Aho et al. [1] and Nielsen and Chuang [25], respectively.

2.1 Intermediate Representation

In general, intermediate representations (IR) are extremely useful for the implementation of multi-language/multi-architecture compiler suites, as well as simplifying and enhancing verification, analysis, and optimization tasks. While various forms of intermediate representations exist, SSA and SSA-based IRs have played a key role in the design of our approach, so we will briefly introduce these concepts below.

Static Single Assignment. A common property of intermediate representations in compilers of imperative languages, SSA form mandates that every value is assigned exactly once (in a static sense), while imposing no restrictions on the number of uses. In order to handle assignments to a variable from different control paths, IRs traditionally rely on the use of a pseudo-operation called the ϕ-function. At points of control flow merges, these ϕ-functions represent the selection of the correct value from a set of assignments according to the actual execution path. This encodes the *static* uncertainty of where a value in use was defined, while also obeying the single-definition rule. The code listing below illustrates the use of ϕ-functions with a translation from pseudo-code (left) to SSA-form (right).

```
if (cond)
    x = 6
else
    x = 4
y = x*2
```

$\longrightarrow$

```
if (cond)
    x1 = 6
else
    x2 = 4
x3 = phi(x1,x2)
y1 = x3*2
```

Having a representation in SSA form is of great advantage to certain optimizations as data dependencies are explicit in the program structure. Reaching definitions analysis[3] becomes obsolete, def-use graphs remain compact, and dataflow analysis is simple and sparse, all of which are beneficial to optimizations that rely on such information.

SSA-based IRs. SSA form has been used for many internal compiler representations, including GCC's GIMPLE, LLVM IR, and others. We briefly describe LLVM as it has had a strong influence on the design of MLIR, which we use in this work.

LLVM [20] models an architecture close to traditional processors with a RISC-like instruction set. It features an infinite set of registers represented by *SSA values* (of primitive type: Boolean, integer, floating-point, pointer), which exposes the dataflow of the program. This allows LLVM to perform transformations without expensive dataflow analysis. Control flow is also explicit in the IR by organizing function bodies into basic blocks, i.e., sections of linear code that always end in a terminator operation that transfers control to another block. Merging control flow with regards to SSA value definitions is handled using explicit ϕ instructions, in direct correspondence with theoretical ϕ-functions.

2.2 MLIR

In contrast to the rigidity of LLVM, MLIR [21] provides an extensible representation with infrastructure for transformations, analysis, and debugging. There exists a variety of mechanisms to enable and manage extensibility in MLIR.

An extension to MLIR is structured into a *dialect*, akin to a namespace, in which all specialized types, operations, and other IR objects are defined. An *operation* is the fundamental unit of execution in MLIR, similar to instructions in LLVM. Each operation defines its own semantics, allowing dialects to represent constructs at arbitrary levels of abstraction. It is important to note that this freely extensible system need not result in a proliferation of disjoint and self-contained dialects. Instead, dialect components such as operations, but also types and transformations, can be freely mixed and reused across dialects.

Moreover, MLIR provides an extensive IR validation mechanism, with which to verify requirements upon IR construction, invariants across transformations, and so on. Verifiers can be defined at the level of operations, types, and traits. These constructs are vital to the extensible system, and encourage designers to focus on reusable and modular components, without inhibiting dialect-specific implementations where appropriate.

The following table summarizes the syntax of relevant MLIR components described in this section. In order to avoid naming collisions between dialects, operation and type names besides the built-in ones are prefixed with a dialect shorthand.

Construct	**Syntax**	**Example**
SSA Value	`%`	`%0`
Symbol (e.g. function name)	`@`	`@mod`
Block name	`^`	`^while`
Dialect Prefix	`dp.`	`std.`
Dialect Operation	`dp.name`	`scf.for`
Dialect Type	`!dp.name`	`!linalg.range`

[3] A definition of a variable **d** reaches a point in the code **p** if there is an execution path from **d** to **p** with no occurrence along that path that ends the validity of d [1]. This form of dataflow analysis becomes obsolete in SSA form since every variable has exactly one definition.

```
1   func @mod(%a: i64, %N: i64) -> i64 {
2       %cond_0 = cmpi "uge", %a, %N : i64
3       cond_br %cond_0, ^while(%a: i64), ^ret(%a: i64)
4
5       ^while(%a_0: i64):
6           %a_1 = subi %a_0, %N : i64
7
8           %cond_1 = cmpi "uge", %a_1, %N : i64
9           cond_br %cond_1, ^while(%a_1: i64), ^ret(%a_1: i64)
10
11      ^ret(%res: i64):
12          return %res : i64
13  }
```

Fig. 2. Modulo function implemented in MLIR.

The MLIR code of a modulo function in Figure 2 will be used to illustrate the concepts that follow. At its core, MLIR employs a functional form of SSA, which distinguishes itself from traditional ϕ-based SSA forms (such as the one found in LLVM) via the use of *block arguments*. Basic blocks (e.g. L5, L11) that form the nodes in the control-flow graph use block arguments for values that are defined in multiple parent blocks. Every block must end in a *terminator* operation that determines where to transfer control next (e.g. jump, conditional branch (L9), return (L12)). Terminators that transfer control to a block with arguments must provide the desired values, similar to a function call. SSA values in MLIR can appear as either block arguments (e.g. `%a_0` on L5), operation operands (e.g. `%a_0` on L6), or operation results (e.g. `%cond_1` on L8), and always have associated type information (e.g. `i64`) available. Compile-time arguments to operations can also be stored in *attributes* instead of SSA values, such as the `"uge"` attribute telling the `cmpi` operation to perform an 'unsigned greater or equal than' comparison (L8).

Another advantage of MLIR is its capability to represent hierarchical code. Operations can define *regions*, which themselves contain other operations, allowing for arbitrary nesting. A function in MLIR (e.g. `func @mod` on L1) is such an operation with a single nested region (L2-L12) containing the function body. As a consequence, loop nests for example need not be represented as linearized control flow via blocks and branches, but can use nested loop operations if more appropriate. The affine dialect makes extensive use of this, and we will later see how this benefits resource estimation (see Section 5.4). The code listing below illustrates the difference in representation between linearized control flow (left) and structured control flow (right), omitting some boilerplate in computing the loop condition to highlight the structural difference.

```
%i = constant 0 : index
br ^header
^header:
    cond_br %loop_cond, ^body, ^exit
^body:
    cond_br %if_cond, ^true, ^false
^true:
    ...
    br ^more_body
^false:
    ...
    br ^more_body
^more_body:
    br ^header
^exit:
```

$\Longleftrightarrow$

```
scf.for %i = 0 to 100 {
    scf.if %if_cond {
        ...
    } else {
        ...
    }
}
```

Several transformation mechanisms are available in MLIR. Every operation can implement specialized hooks for canonicalization and folding purposes. A DAG-to-DAG pattern rewriter simplifies the implementation of transformations that can be expressed as a simple replacement of one DAG pattern with another. A DAG pattern constitutes of a set of operations connected by the usage of values (via arguments) and their definitions (via return values), so-called def-use chains (e.g. the operations `subi` (L6) and `cmpi` (L8) in Figure 2 are linked via the def-use chain of the value `%a_1`). The pattern rewriter can then identify patterns by traversing def-use chains in the IR, and transform any matches accordingly. Besides the pattern rewriting framework, general operation passes can be written that arbitrarily modify an operation and all those nested within. Finally, further infrastructure is provided to simplify dialect lowering and type conversions.

2.3 Quantum Computing

Quantum State. The quantum analog of a classical bit is a quantum bit or qubit. Whereas a classical bit can be in one of two states at any given time, a qubit is in a complex superposition of two basis states $|0\rangle, |1\rangle$ corresponding to the values 0 and 1, respectively. The state $|\psi\rangle$ of a qubit can thus be written as

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle,$$

where $\alpha_0, \alpha_1 \in \mathbb{C}$ are complex numbers, so-called *probability amplitudes*, such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$. As the name suggests, these complex numbers are related to probabilities. Namely, measuring a qubit yields a classical bit equal to 0 or 1 with probability $p = |\alpha_0|^2$ or $|\alpha_1|^2 = 1-p$, respectively. Measurement also collapses the state onto the observed outcome, meaning that the post-measurement state will be $|0\rangle$ or $|1\rangle$.

The quantum state $|\phi\rangle$ of n qubits may be written as a superposition over all 2^n n-bit strings,

$$|\phi\rangle = \alpha_0 |\underbrace{0\cdots 0}_{n}\rangle + \cdots + \alpha_{2^n-1} |\underbrace{1\cdots 1}_{n}\rangle,$$

where the amplitudes again satisfy the normalization condition $\sum_i |\alpha_i|^2 = 1$. Usually, the n-bit strings are interpreted as integers, resulting in a shorter notation:

$$|\phi\rangle = \sum_i \alpha_i |i\rangle.$$

Measuring all n qubits collapses the state onto $|i\rangle$ with probability $p_i = |\alpha_i|^2$ and yields the outcome i.

In contrast to the state of a classical bit, general quantum states cannot be copied due to the no-cloning theorem [25, Box 12.1]. Specifically, the theorem implies that there exists no unitary operator (see below) U such that $U|\phi\rangle|0\rangle = |\phi\rangle|\phi\rangle$ for all states $|\phi\rangle$.

Quantum Operations. Similarly to classical computers, the state of a quantum computer may be altered by applying quantum operations to qubits. These operations can be represented as unitary[4] matrices $U \in \mathbb{C}^{2^n \times 2^n}$, and the state after applying a quantum operation with matrix-representation U is

$$|\phi'\rangle = U|\phi\rangle,$$

where $|\phi\rangle$ is interpreted as a column vector of amplitudes $(\alpha_0, \ldots, \alpha_{2^n-1})^T$ and then multiplied with the matrix U. The inverse of an operation is defined by its Hermitian adjoint (conjugate transpose), denoted $U^\dagger$. Operations that are hermitian[5] form their own inverses, and commonly appear in optimization techniques.

[4]Recall that U is unitary if $U^\dagger U = UU^\dagger = \mathbb{1}$.

[5]Recall that U is hermitian if $U = U^\dagger$.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

$$R(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \quad Rz(\theta) = e^{-i\theta Z/2} = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

$$CX = (\mathbb{1} - |1\rangle\langle 1|) \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Fig. 3. Common single- and multi-qubit gates. Rx and Ry (not shown) are defined analogously to Rz, with $Rx(\theta) = e^{-i\theta X/2}$ and $Ry(\theta) = e^{-i\theta Y/2}$. The collection of gates shown is a superset of the universal gate set $\{CX, H, S, T\}$.

A few common single-qubit gates are shown in Figure 3, such as the Hadamard gate H, the Pauli X, Y, Z gates, and the T and S gates. Moreover, we use the standard definition from Nielsen and Chuang [25, Exercise 4.1] for the rotation gates R, Rx, Ry, Rz. Many important multi-qubit gates are controlled versions of single-qubit gates, which apply the single-qubit gate only on the subspace where all control qubits are equal to $|1\rangle$. An n-ary controlled single-qubit gate U can be written as

$$^{c}U = (\mathbb{1} - |1 \cdots 1\rangle\langle 1 \cdots 1|) \otimes \mathbb{1} + |1 \cdots 1\rangle\langle 1 \cdots 1| \otimes U,$$

where $\otimes$ is the Kronecker product and $|i\rangle\langle i|$ denotes the projector onto $|i\rangle$. For example, the controlled NOT operation (or $CNOT/CX$) is a controlled X gate and is also shown in Figure 3.

A set of quantum logic gates $\mathcal{G}$ is termed *universal* if for any unitary $U \in 2^n \times 2^n$ and precision parameter ϵ, there exists a finite sequence of gates $S = G_m \ldots G_2 G_1$ where $G_i \in \mathcal{G}$ such that $\max_{|\psi\rangle} ||(S - U)|\psi\rangle|| \leq \epsilon$. That is, $\mathcal{G}$ can be used to approximate U to arbitrary precision. An example of a commonly used universal gate set is the Clifford+T set $\{CNOT, H, S, T\}$.

Execution Model. We model quantum computation using a classical "host" computer in combination with a quantum co-processor, or **quantum processing unit (QPU)**, with bidirectional real-time communication available. The QPU must be able to support (at minimum) state preparation, a universal gate set, and measurement. A quantum program then consists of a combination of classical and quantum instructions. In each step of the program, the classical host may send sequences of quantum instructions to the quantum co-processor for execution. The responsibility of managing and communicating with the quantum co-processor falls on the quantum **runtime environment (RTE)**, in particular for such tasks as qubit allocation. Consequently, the intricacies of quantum memory management are not considered for the compilation process presented in this work. Sequences of quantum operations may be seen as quantum circuits similar to classical logic circuits. Once the circuit has been executed, the co-processor can return measurement outcomes to the classical host, which may also use these outcomes to alter the execution path.

We distinguish two phases of a program's lifecycle for the purposes of optimization, defined below:

- **compile time:** The program is analyzed and transformed in a fully static way, without the execution of either classical or quantum program parts. In particular, program inputs are unavailable at this stage.
- **run time:** Strictly speaking, this term might be used to describe the execution stage of the program on quantum hardware. However, in the context of optimizations, we also consider any circuit generation phases to be *at run time*. The reason is that such phases already perform "run-time" operations such as classical program execution, control flow resolution, and

program input propagation, clearly distinguishing it from the static scenario above. The execution of classical meta-programs that perform circuit generation in eDSLs falls into this category.

3 A QUANTUM PROGRAMMING STACK

An overview of the proposed quantum programming stack is presented in Figure 1(a). We envision that high-level quantum programming languages are translated to QIRO in the proposed input dialect (labeled *Quantum* in the diagram), which can then be lowered to the optimization dialect (labeled *QuantumSSA*). Code examples for both dialects can be found in Figure 1(b) and Figure 4. We designed the optimization dialect specifically to enable quantum-classical co-optimization and to enable maximal reuse of compiler components by exposing data dependencies explicitly in the IR. In addition to quantum-specific optimization passes, the IR may thus be optimized using classical transformation passes such as inlining, loop unrolling, and **common subexpression elimination (CSE)**. In a last target-specific step, the optimization dialect may be lowered, e.g., to LLVM IR [20] for simulation, execution on hardware, or resource estimation.

The design of QIRO is guided by the following principles:

(1) Lowering of existing programming languages into the IR must be simple.
(2) The IR must be capable of supporting state-of-the-art optimization algorithms (quantum and classical).
(3) The IR should enable re-use of existing compilation infrastructure.

We propose two MLIR dialects in order to separate the first requirement from the other two, which are quite different in their nature. As their names suggest, the goal of the input dialect is to enable simple and efficient lowering from existing quantum programming languages, while the optimization dialect is geared toward enabling maximal re-use of existing infrastructure and supporting a wide range of optimizations. Structurally, the two dialects primarily differ in the semantics of how quantum operations interact with qubits.

The input dialect represents qubits using *memory-semantics*. That is, qubit allocation returns a unique reference to a qubit. Quantum operations that act on such qubit references do not consume the qubit value, and affect the quantum state via side-effects. Qubit registers function in a similar way, in that allocation returns a unique reference to a register of newly-allocated qubits. An immediate benefit of using memory-semantics is that the structure of the IR inherently prevents a program from violating the no-cloning theorem. As each operation always interacts with the state of the processor via side-effects, there simply is no mechanism available via which a quantum state could be copied. We note, however, that it is not possible to statically guarantee that the same qubit is not passed multiple times to the same quantum operation (and thus aliased), since we allow for quantum register access using dynamic indices. Such cases may be addressed by emitting code that performs this check at run time, e.g. through the RTE.

By contrast, the optimization dialect can be viewed as a quantum-version of SSA that emulates *value-semantics*. By this, we mean that quantum operations consume and return quantum state values instead of operating on qubits via side-effects. These values represent the state of a qubit at a particular time-step in the execution of the quantum program. Note however that such quantum state values are never computed as they would be in a classical setting, instead they merely provide a representation for the purposes of SSA. Using SSA in this way comes with similar benefits to its classical counterpart: (1) the dataflow graph is made explicit in the IR and (2) quantum operations in this dialect are free of side-effects, facilitating optimization (e.g. dead-code elimination for operations whose return values are never consumed).

Table 1. Types Defined by the Quantum Dialects

Type	Quantum	QuantumSSA
Qubit	`!q.qubit`	`!qs.qstate`
Qubit Register	`!q.qureg<n>`	`!qs.rstate<n>`
Native 1-Qubit Gate	`!qs.u1`	
Native 2-Qubit Gate	`!qs.u2`	
Circuit	`!qs.circ`	
Controlled Op	`!qs.cop<n, baseT>`	

n - register size/number of control qubits.
baseT - type of the underlying operation.

4 THE QUANTUM DIALECTS IN DETAIL

In this section, we discuss the two proposed dialects in detail, including how to map existing quantum programming languages to the input dialect and how to lower the input dialect to the optimization dialect.

4.1 Describing Quantum Programs

In MLIR, all operations are grouped into self-contained units called *modules*, allowing the compiler to process these in parallel. The region of a module is then usually composed of subroutine definitions and external declarations. We distinguish between classical and quantum subroutines. Purely classical subroutines can be placed inside an MLIR *function*, while quantum program segments can be placed inside quantum functions termed *circuits*.

The quantum dialects feature a powerful instruction set with which to interact with the quantum co-processor. This set is composed of all the components for universal quantum computation: qubit initialization into a known state, qubit readout (measurement), and a universal gate set (a set of unitary transformations with which all unitary transformations can be approximated to arbitrary precision).

Additionally, to represent operations at a higher level of abstraction, our IR also features meta-operations (sometimes called functors [32]) that modify existing operations in some way. Unless natively supported by the target architecture, these will be lowered via standard or user-defined decomposition routines before execution on the quantum processor.

4.2 Types

QIRO defines qubit and quantum register types that are used to represent all quantum data. Higher-level type abstractions, such as integers, fixed-point numbers, and so on, can be implemented on top of these basic types. Each quantum dialect has its own version of the qubit and register types, which reflects the distinction between value- and memory-semantics of the two dialects. We introduce several additional types to represent quantum operations themselves: a basic type for native single- and two-qubit gates, a type for circuits, and a type for controlled operations (containing the base operation type and number of control qubits). We provide the full list of types in Table 1. The dialect shorthands are `q.` for the input dialect (Quantum) and `qs.` for the optimization dialect (QuantumSSA).

4.3 Operations

The quantum operations can be broadly separated into four categories: qubit management, native gates, meta-operations, and user-defined operations. An overview can be found in Table 2.

Table 2. Operations Defined by the Quantum Dialects (Dialect Prefixes and Types Omitted)

Qubit Management	**Native Gates**	**Meta-Operations**	**User-defined Operations**
`%qb = alloc`	`H/X/Y/Z/S/T %q`	`%op = ctrl %op, %q`	`circ @name(%arg..) {...}`
`%r = allocreg(n)`	`R/Rx/Ry/Rz(`ϕ`) %q`	`%op = adj %op`	`call @name(%arg..)`
`free %qb`	`CX %qb, %q`		`%circ = getval @name`
`freereg %r`	`SWAP %qb, %qb`		`apply %circ(%arg..)`
`%m = meas %q`			
`%qb.., %r = extract %r[i..]`			
`%r = combine %r[i..], %qb..`			

%qb - qubit value, **%r** - register value, **%q** - value of either quantum data type.
%op - any quantum op, **@name** - circuit symbol, **%circ** - circuit value.
n, **i** - integers, $\boldsymbol{\phi}$ - floating point, **%arg** - any value.

Qubits and registers are allocated and initialized to the $|0\rangle$ state using the appropriate operations. Allocation errors due to space constraints or other reasons are expected to be handled by the runtime environment. To reinitialize (or reset) a qubit, one can perform a combination of measurement and conditional bit-flip. Note that qubit resources must be explicitly freed, which allows us to enforce that quantum resources may not be used after deallocation, since the freeing operations act as sinks for quantum state values. Measurement operations return a classical bit value or an array thereof, depending on whether the input is a qubit or a register. Note that measurements are performed in the z-basis by default.

The usual single- and two-qubit gates used in the literature are provided as native gate operations and it is straightforward to extend the gate library. All native gates (excluding SWAP) are overloaded to accept both qubits and registers as their target. The latter can be interpreted as a *foreach* loop, meaning the gate is applied to all qubits inside the given register. Rotation gates additionally accept a continuous angle parameter as either a constant (operation attribute) or a variable (SSA value). A *hermitian* trait is attached to all self-inverse gates, which is used in peephole optimizations.

As the current design of MLIR does not enable operations to directly act on other operations, meta-operations instead act on SSA values *representing* a quantum instruction. These values can be obtained from native gates by omitting the target qubit operands, which must be passed to the meta-operation instead. Multiple meta-operations can also be chained by only supplying target qubit operands to the final one.

Circuits act as "quantum functions", in that they form a grouping of operations (quantum and classical) that only have access to values provided as function arguments and those created inside. To enable maximal flexibility, circuits are allowed to accept and return any type of values. There are two ways to invoke a quantum circuit. A direct *call* via the circuit name, and an indirect *application* via a generated circuit value. Indirect circuit application is intended for circuits that have been modified by meta-operations. The code listing below illustrates the use of both constructs.

```
q.circ @qft(%r, %n) {
    // define custom QFT operation
}

q.call @qft(%r, %n) // apply QFT

%qft = q.getval @qft
%qft_inv = q.adj %qft
q.apply %qft_inv(%r, %n) // apply inverse QFT
```

Table 3. Quantum Programming Constructs and their Representation in High-level Languages and QIRO

Constructs	Q#	Qiskit	Silq	QIRO

Allocation	
	`using (q = Qubit()) {}`
	`using (r = Qubit[n]) {}`
		`resArr = ClassicalRegister(n)`
		
		`r = QuantumRegister(n)`
		`c = QuantumCircuit(r, resArr)`
			
			`q := 0:𝔹`
			`r := array(n,0:𝔹):𝔹[]`
				
				`%q = q.alloc -> !q.qubit`
				`%r = q.allocreg(n) -> !q.qureg<n>`

Deallocation	automatic	automatic	automatic	`q.free %q : !q.qubit`
				`q.freereg %r : !q.qureg<n>`

Measurement	`let res = M(q)`
	`let resArr = MultiM(r)`
		`c.measure(r[i], resArr[i])`
		`c.measure(r, resArr)`
			`res := measure(q)`
			`resArr := measure(r)`
				`%res = q.meas %q : !q.qubit -> i1`
				`%resArr = q.meas %r : !q.qureg<n> -> memref<nxi1>`

Native gates	`H(q)`
	`Rz(ϕ, q)`
	`CNOT(qc, qt)`
		`c.h(r[i])`
		`c.rz(ϕ, r[i])`
		`c.cx(r[i], r[j])`
			`q := H(q)`
			`q := rotZ(ϕ,q)`
			–
				`q.H %q : !q.qubit`
				`q.Rz(ϕ) %q : !q.qubit`
				`q.CX %qc, %qt : !q.qubit, !q.qubit`

User operations	`function Foo(args..) : resT {}`
	`let ret = Foo(args..)`
	
	`operation Bar(args..) : Unit {}`
	`Bar(args..)`
		`def Foo(args..):`
		`ret = Foo(args..)`
		
		`bar = QuantumCircuit(m)`
		`Bar = bar.to_instruction()`
		`c.append(Bar, r[0, m])`
			`def Foo(args..) {}`
			`ret := Foo(args..)`
			
			`def Bar(args..) {}`
			`args.. = Bar(args..)`
				`func @Foo(args..) -> !resT {}`
				`%ret = call @Foo(args..) : (!argT) -> !resT`
				
				`q.circ @Bar(args..) -> !resT {}`
				`q.call @Bar(args..) : !argT -> !resT`

Meta-operations	`Controlled X(qc, qt)`
	
	`Adjoint X(q)`
	
	`Adjoint Bar(q)`
		`CX = XGate().control()`
		`c.append(CX, [r[i], r[j]])`
		
		`Xdg = XGate().inverse()`
		`c.append(Xdg, r[i])`
		
		`Bar = bar.to_instruction()`
		`BarA = Bar.inverse()`
		`c.append(BarA, r[0, m])`
			`if qc {`
			`  qt := X(qt)`
			`}`
			`q := reverse(X)(q)`
			
			`q := reverse(Bar)(q)`
				`%X = q.X -> !q.gate`
				`q.ctrl %X, %qc, %qt : !q.gate, !q.qubit, !q.qubit`
				
				`q.adj %X, %qt : !q.gate, !q.qubit`
				
				`%Bar = q.getval @Bar -> !q.circ`
				`%BarA = q.adj %Bar : !q.circ -> !q.circ`
				`q.apply %BarA(%q) : !q.circ(!q.qubit)`

Conditionals	`if (res = One) {`
	`  X(q)`
	`}`
		`c.x(r[i]).c_if(resArr[j], 1)`	`if res {`
			`  q := X(q)`
			`}`
				`scf.if %res {`
				`  q.X(%q) : !q.qubit`
				`}`

Loops	`for (i in 0 .. Length(r)) {}`	`for i in range(0, len(r)):`	`for i in [0, n) {}`	`affine.for %i = 0 to %n {}`
				`scf.for %i = %c0 to %n step %c1 {}`

			- - - - *register size must be kept around as a value* - - - -	

	`repeat {`
	` ...`
	` let res = M(q)`
	`} until (res == Zero)`
		–	`while res {`
			`  ...`
			`  res = measure(q)`
			`}`
				`^repeat:`
				`  ...`
				`  %cond = q.meas %q : !q.qubit -> i1`
				`  cond_br %cond, ^repeat, ^next`

Automatic here refers to scope-based deallocation. As a special case in Qiskit, all operations must be called from a circuit object (e.g. `c.h()`). This object-oriented notation is not to be confused with dialect prefixes in QIRO (`q.` and `qs.`).

4.4 Meta-Operations

Meta-operations require some special care, as they are intended to modify other quantum operations in a way that must be consistent with the laws of quantum mechanics. For native gates, this requirement is always satisfied. However, it is only legal to apply meta-operations to circuits (i.e. user-defined operations) that are also *unitary*. In practice this implies that such circuits must not contain any measurements, and may only contain calls to pure functions, conditions which are asserted on the input IR.

Standard lowering routines for meta-operations on *circuits* can be placed anywhere in the pass-pipeline, such as the one used in the benchmarks of Section 6. They may also enable certain optimization opportunities, as described in Section 5.3. At the lowest level, adjoint/control decomposition of native gates may be left to the runtime environment, until the final control qubit count or adjoint parity is known.

QIRO also supports custom implementations of adjoint and controlled versions of circuits. These must (1) be marked with a special attribute to identify them as adjoint/control decompositions, and (2) follow the respective naming convention to identify them with the original circuit.

4.5 Mapping Quantum Languages to the Input Dialect

Since QIRO is capable of representing languages based on the widely used circuit model of quantum computing, many such languages are able to benefit from performant static code optimization enabled by our design. In Table 3 we show how high- and low-level constructs map to our IR from a selection of languages with different foci, namely Q# (high-level quantum and classical code), Qiskit (NISQ/eDSL), and Silq (intuitive algorithm development).

```
q.circ @ENT(%qb, %r, %n) {
  q.H %qb
  affine.for %i = 0 to %n {
    q.CX %qb, %r[i]
  }
}
```

↓

```
qs.circ @ENT(%qb_0, %r_0, %n) {
  %qb_1 = qs.H %qb_0                             loop-carried values
  %qb_2, %r_1 = affine.for %i = 0 to %n ↩
        iter_args(%qb_i_0 = %qb_0, %r_i_0 = %r_0) {
    %qt_0, %rem = qs.extract %r_i_0[%i]
    %qb_i_1, %qt_1 = qs.CX %qb_i_0, %qt_0
    %r_i_1 = qs.combine %rem[%i], %qt_1
    affine.yield %qb_i_1, %r_i_1
  }
  qs.return %qb_2, %r_1              next iteration values
}
                final value of each quantum argument
```

Fig. 4. An entanglement circuit is lowered from the input dialect to the optimization dialect. The qubit %qb is entangled with a register %r of size %n using an affine loop. In the optimization dialect, loop structures make use of loop-carried values to enable value-semantics.

Qubit and register allocation, measurement, and low-level quantum gates all map to our IR in a straightforward fashion. Higher-level quantum operations should be translated to circuits, whereas functions are reserved for purely classical code. An advantage of MLIR is the ability to express both conditional expressions and for loops in the form of structured control flow, using operation nesting rather than the flat block structure traditionally used in SSA. More complex control flow can be represented using blocks, such as the while or repeat until success loops shown in Table 3. Whenever possible, for loops should be mapped to the affine dialect to take advantage of the dialect's powerful optimization passes. For further information we refer to Appendix A, where we discuss how Q# constructs may be mapped to our IR in more detail.

4.6 Lowering to the Optimization Dialect

The input dialect is lowered to the optimization dialect by recursively traversing the nested structure inside a module in a post-order fashion. This ensures that nested operations are replaced first, since these need to be available when constructing the new parent operation. The main purpose of the lowering pass is to transform from memory-semantics into value-semantics, as well as to transform some convenience features into explicit code. Figure 4 depicts a small example program lowered to the optimization dialect.

Gates. In the input dialect, gates act on qubit references implicitly via side-effects. To convert this behaviour to value-semantics, a gate operation needs to generate a new state value for each unique quantum argument. During lowering, the transformation pass keeps an updated map of the most recent state value for each qubit reference. Gate arguments are then replaced with the latest state value, and the map is updated with the return values of the quantum operation.

Circuits. Similar to MLIR functions, circuits are isolated portions of code that can only reference those SSA values defined in the body and argument list. Thus, the lowering pass can keep a local map for qubit state values when entering a circuit, which is initially populated with the circuit's arguments. The signature must also be updated to include the return type of each quantum argument. Block terminators that transfer control flow back out of the circuit (often implicit in the input dialect) are replaced with a `return` operation that returns all values present in the qubit state map when reaching the end of the circuit (see Figure 4).

Loops. `For` loops from both the ***structured control flow*** **(SCF)** and *affine* dialects are lowered to value-semantics by exploiting loop-carried values for quantum states, as shown in Figure 4. The `iter_args` parameter defines these loop-carried values and provides their initial values. Closing off the loop body, the `yield` operation returns the values to be passed to the next iteration, or to be returned upon reaching the final iteration.

Register access. At its core, SSA form is best suited to represent dataflow of *scalar* variables. When dealing with aggregate structures such as arrays (qubit registers) and global memory, it has traditionally been difficult to directly represent these via SSA [1]. Consequently, memory is frequently modeled and represented separately, such as in LLVM and MLIR. Memory references can be used to `load` memory elements into scalar SSA values, which can be easily operated on, and eventually `stored` back into memory.

This problem is also side-stepped in the input dialect, which conveniently represents qubit register access via indices whenever the register value is used. This preserves the uniqueness of qubit and register references required by the input dialect. However, it does obscure data dependencies between individual qubits of the register in the dataflow graph. To enable optimizations requiring single qubit dataflow, we make these data dependencies *locally* explicit in the optimization dialect. Similar to the memory load/store model, individual qubits or qubit slices are `extracted` from registers and re-`combined` after use. Whenever possible, we use static data analysis to consolidate such extract/combine operations in order to directly expose the dataflow of register elements. In this way, we can exploit proper SSA semantics inside optimization passes, without needing to handle register dataflow analysis for every optimization. This is in line with our design principle of modularizing reusable components.

5 TRANSFORMATIONS & ANALYSES

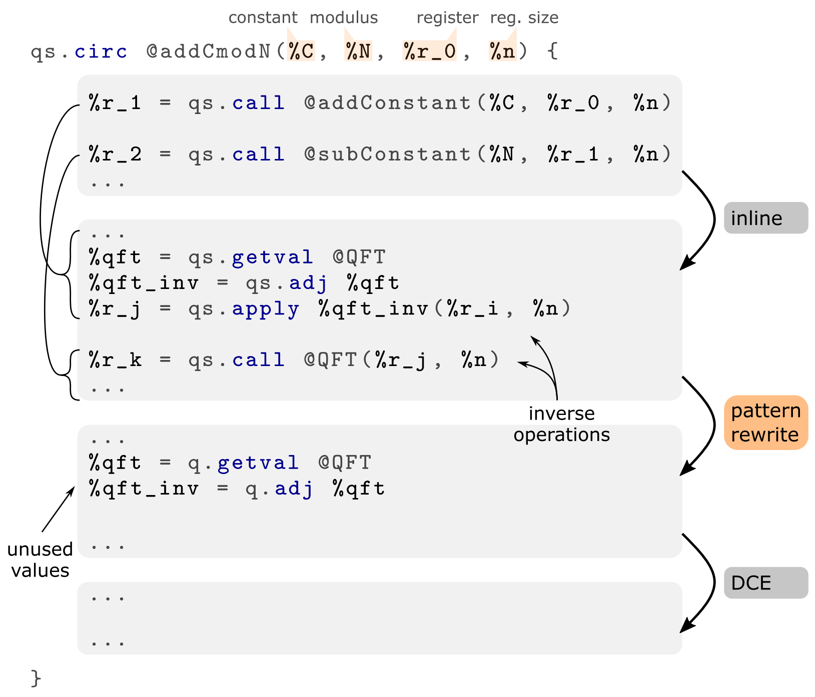
QIRO supports and facilitates the implementation of many transformations and analyses important to compilation of mixed quantum-classical programs. In particular, static optimization passes are much better suited for large-scale quantum program optimization, in contrast to NISQ-focused, run-time optimization systems, as we show in Section 6. Furthermore, quantum resource estimation is a compute-intensive analysis that can profit from our proposed IR and compilation stack.

5.1 Classical Optimizations

MLIR provides us with a variety of useful transformations applicable to a program written in our IR, maximizing reuse of compiler components where appropriate. The following MLIR passes can be used out-of-the-box on mixed classical-quantum programs, with the exception of *Inlining* which was slightly modified to work on quantum callables.[6]

Canonicalization. While this technically refers to bringing code into a single “canonical” form, this pass includes many important optimizations such as *constant folding* and ***dead code elimination*** **(DCE)**. Note that DCE not only applies to classical operations, but also directly to side-effect free quantum operations whose return values go unused (such as in Figure 5) due to the SSA structure of the optimization dialect.

[6]While the list provided here focuses on classical passes that affect the *quantum* program parts, purely classical program sections can naturally be optimized with the usual passes both at the IR-level within MLIR, and at the backend-level e.g. within LLVM.



```
qs.circ @addCmodN(%C, %N, %r_0, %n) {

    %r_1 = qs.call @addConstant(%C, %r_0, %n)

    %r_2 = qs.call @subConstant(%N, %r_1, %n)
    ...

    ...
    %qft = qs.getval @QFT
    %qft_inv = qs.adj %qft
    %r_j = qs.apply %qft_inv(%r_i, %n)

    %r_k = qs.call @QFT(%r_j, %n)
    ...

    ...
    %qft = q.getval @QFT
    %qft_inv = q.adj %qft

    ...

    ...

    ...
}
```

Fig. 5. Application example showing different passes interacting with each other to remove superfluous computations. This modular adder routine makes successive calls to constant-addition routines which each begin and end by invoking the QFT and QFT† operations, respectively.

CSE. This pass can eliminate duplicate (classical) expressions and replace them with a single value. Similar to the canonicalizer, this can improve knowledge of dataflow inside registers, similar to what is demonstrated in Figure 1(b).

Inlining. Traditional function inling is generally useful for speeding up the computation of small functions, trading program size for speed. Moreover, this pass was slightly adapted to inline quantum circuits, which is often vital to expose optimization opportunities. As the instruction sequences inside circuits are often pre-optimized by hand, many more opportunities arise once multiple circuit calls are merged together (as demonstrated in Figure 5).

Affine loop unrolling. Loops in a quantum program expressed via the affine dialect can be unrolled (full or by a factor) using this pass. In the same vein as the inlining pass, unrolling allows for the optimization of quantum gates on the boundary of loop iterations. One should note that the loop boundary optimization described below is an alternative to using affine loop unrolling which can be applied to non-affine loops as well.

5.2 Quantum Circuit Optimizations

It is clear that traditional approaches to *circuit* optimization can be supported by running optimizers on sections consisting only of quantum operations, say e.g. a loop body. We note, however, that the interaction between quantum and classical optimizations, such as peephole optimization and loop-unrolling, may further increase the impact on the quantum resource requirements. Moreover, many identity-based heuristics found in quantum circuit optimization papers [24] are naturally suited for implementation using MLIR's pattern rewriter, as these represent simple DAG-to-DAG transformations. Whereas the classical optimizations given above are already available in MLIR, the quantum optimization passes below were re-implemented in QIRO to demonstrate its capability to support optimization passes.

5.2.1 Local Register Dataflow. This pass is not based on existing optimizations, instead being devised for the specifics of QIRO's design. As a precursor to other optimizations, we wish to consolidate `extract`/`combine` operations that occur when translating register accesses from the input dialect to the optimization dialect (see Figure 1(b)). To enable optimizations in these scenarios, we introduce a transformation pass that merges `combine` instructions with subsequent `extracts` whenever they are linked in the use-def chains and satisfy the following restrictions: If there exists overlapping qubit indices, these can be removed from both operations extending the qubit values lifetime from the first block into the second. If all indices are distinct, we delay re-combining qubits from the first block until the end of the second block, merging the two `combine` operations. With static indices, these assertions can always be made. With dynamic indices, on the other hand, we perform this optimization only if dataflow analysis on the indices can guarantee one of the conditions. Additionally, `combine-combine`, `extract-extract`, and empty `extract-combine` patterns are optimized in a similar fashion to expand the regions of locally available register dataflow.

5.2.2 Peephole Optimizations. Simple peephole (or window) optimizations are straightforward to implement on the optimization dialect using MLIR's pattern rewriting framework. Unitary gate cancellation is among the most common ones, where we differentiate between two cases: two identical but Hermitian (or self-inverse) gates, and a general gate followed by its unitary inverse. Additionally, a quantum analog to classical constant folding, namely merging parametrized rotation gates, also falls in this category.

Hermitian Operations. Matching is performed on all quantum operations carrying the Hermitian trait. As every quantum data type in the input must also be present in the output, we follow the use-def chain of each quantum operand value to its defining operation. If the operation is the same for all such operands, is the same *kind* as the operation we started with, and all remaining arguments are identical, we have found a match. Replacing the matched pattern is then as simple as replacing all uses of the returned values from operation 2 by the input values of operation 1, and erasing both operations.

```
%a1, %b1 = qs.CX %a0, %b0
%a2, %b2 = qs.CX %a1, %b1
%a3 = qs.H %a2
```

$\longrightarrow$

```


%a3 = qs.H %a0
```

General Operations. In the more general case, we must extend the matching over the `adjoint` meta-operation, but otherwise remains largely the same. Starting from either the unitary gate or the adjoint op, we trace back all quantum data operands for a match, and additionally follow the unitary operand of the adjoint op to its definition to determine if the operation kind matches.

The replacement step consists of erasing the matched adjoint and unitary gate operations, along with replacing all uses of the quantum data operands. We note that this pattern may be extended straightforwardly from native operations to user-defined quantum circuits.

```
%a1 = qs.T %a0
%t = qs.T
%a2 = qs.adj %t, %a1
%a3 = qs.H %a2
```

$\longrightarrow$

```

%t = qs.T

%a3 = qs.H %a0
```

Merging of Rotations. Again we follow a similar procedure for matching two adjacent (in this case rotation) gates on the use-def graph, this time replacing them with a single operation that has as parameter the sum of the individual rotation angles. The new rotation angle is computed by a classical operation inserted in front of the rotation, or directly inserted into the operation in case of static arguments.

```
%a1 = qs.Rz(0.1) %a0
%a2 = qs.Rz(0.3) %a1
%a3 = qs.H %a2
```

$\longrightarrow$

```
%a1 = qs.Rz(0.4) %a0

%a3 = qs.H %a1
```

5.2.3 *Loop Boundary Optimization.* This optimization analyzes loop structures across their iteration boundaries, which is particularly effective on loop bodies with a symmetric structure, such as compute/uncompute segments. Repeating such a body multiple times leads to many redundant instructions that undo and redo the same operation at the end and beginning of every iteration. The optimization is performed by walking use-def chains from both ends of the loop body and keeping track of matches, similar to previously described peephole optimizations. Operations that cancel can be hoisted out of the loop body and placed right before and after the loop operation (for the first and last iteration). A similar optimization is possible for rotations, which can be merged across loop iterations.

```
%a1 = scf.for %i=0 to 6 ↩
    iterargs(%a_0 = %a) {
  %a_1 = qs.H %a_0
  %a_2 = qs.T %a_1
  %a_3 = qs.H %a_2
  yield %a_3
}
```

$\longrightarrow$

```
%a0 = qs.H %a
%a1 = scf.for %i=0 to 6 ↩
    iterargs(%a_0 = %a0) {
  %a_2 = qs.T %a_0
  yield %a_2
}
%a2 = qs.H %a1
```

5.3 Partial Lowering & Decompositions

In the process of translating a quantum program to executable code, it is necessary to *decompose* complex quantum operations into simpler gates supported by the instruction set of the target architecture. At the lowest level, this is best left to an architecture-aware backend, but, where sensible, one should aim to perform these decompositions within the IR so as to leverage the multi-level rewrite infrastructure. These decompositions can then be interleaved with optimization passes to maximize optimization potential. For example, adjoint and controlled versions of user-defined operations are fully expressible within the IR, and can benefit from optimizations pre- and post-decomposition.

Adjoint-Circuit Lowering. A standard adjoint lowering pass on *unitary* circuits can be implemented by generating a new circuit in which the order of quantum operations has been reversed, and the adjoint meta-operation is applied to each of the operations inside. An optimization opportunity arises here for native gates with the `hermitian` trait, as such gates are self-inverse and need not be modified with an adjoint operation.

Controlled-Circuit Lowering. Similarly, controlled *unitary* circuits can be lowered by generating a new circuit and propagating the control meta-operations to each quantum operation inside. The qubits and registers upon which the circuit is controlled must be passed as new arguments to the generated circuit. Another optimization opportunity arises here when using attributes, provided by the frontend, to mark special `compute`/`uncompute` sections in the code [15]. This allows omitting the control propagation on these sections, while still producing the same computation.

5.4 Resource Estimation

One particularly efficient way to generate *resource* (or quantum gate) counts, is to strategically lower and replace quantum operations by classical ones in a way that preserves the structure of the quantum program, and increments simple counters for each measured resource [23]. In this process, all adjoint and controlled circuits must first be lowered using the passes described above. Then, for each native gate, a formula can be provided to indicate the decomposition cost of their controlled and adjoint versions in terms of the tracked resources. Thus, all native gate invocations can directly be replaced with classically computed counter increments.

Any remaining and unused operations from the quantum dialects must be stripped, and circuit definitions and calls must be converted to standard MLIR functions. Special care must be taken when removing measurements, so as to provide a conservative estimate for computations that depend on measurement outcomes [23]. Finally, the purely classical IR can be lowered to LLVM IR using standard MLIR infrastructure, and subsequently compiled into an executable outputting the final resource counts for a given input size. A simplified implementation of this resource estimator was used to compute the rotation gate counts in Section 6.3. We note that the run time for computing resource estimates may be reduced further using custom compiler passes such as the ones employed by Meuli et al. [23].

5.5 Run-time Optimization

We stress that while it is beneficial to run optimizations at compile time, certain optimizations may provide additional benefits at run time, and should be seen as complementary to our work. This is especially important for quantum programs where the quantum circuits being executed on the QPU depend heavily on run-time parameters (user-input as well as qubit measurements). In such cases, run-time optimizations may further reduce resource requirements. Additionally, mapping of a quantum program to specific quantum computer architectures often introduces new optimization opportunities as well, due to transformations required by varying qubit layouts and connectivities, as well as differing native gate sets.

6 EVALUATION

We evaluate the performance of our IR for optimization and resource estimation on the example of Shor's algorithm. Similarly, we evaluate the effectiveness of standard optimizations performed at compile time compared to those performed at run time on the same algorithm. Factoring large numbers is anticipated to be one of the earlier applications of large-scale quantum computers with proven exponential speed-up. With the number of elementary gates growing quickly with the input size, compilation systems that rely on building up large circuit data structures in Python (such as ProjectQ and Qiskit) are slow at optimization and estimating resource requirements. With the projected need for error-corrected computation, non-Clifford gates such as general rotation gates are expected to make up the bulk of computation time. We thus focus on reporting the number of single-qubit rotations during our evaluation. Results from our prototype implementation are labeled QIRO.

Our evaluation uses an implementation of Shor's algorithm by Beauregard [4], excluding manual optimizations such as canceling the (inverse) QFTs of subsequent Fourier adders [7], resulting in an identical implementation to the one in ProjectQ [31]. We implemented the algorithm directly in QIRO's input dialect as a starting point for our compilation pipeline. This implementation can be found in full in Appendix B. The program is then subjected to the following steps:

Table 4. Pass Sequence used Within QIRO to Run Optimizations and Perform Resource Estimation on Shor's Algorithm

Pass	Description
−convert-mem-to-val	Converts from the input to the optimization dialect, switching qubit semantics in the process.
−lower-ctrl	Converts controlled circuit to new circuit with controls propagated to operations inside, does not propagate on compute/uncompute sections indicated by `compute`/`uncompute` attribute.
−strip-circ	Removes unused circuit definitions.
−canonicalize	Built-in MLIR canonicalization pass, also performs DCE, constant folding, and operation rewrite patters, including the *local register dataflow* patters.
−strip-circ	Removes unused circuit definitions.
−circuit-inline	Inlines circuit calls with the corresponding circuit body (except where `no_inline`/`no_inline_target` attribute present).
−strip-circ	Removes unused circuit definitions.
−canonicalize	Built-in MLIR canonicalization pass, also performs DCE, constant folding, and operation rewrite patters, including the *local register dataflow* patters.
−strip-circ	Removes unused circuit definitions
−quantum-gate-opt	Performs hermitian gate cancellation, adjoint gate cancellation, adjoint circuit cancellation, rotation gate folding, controlled-rotation gate folding, and loop-boundary optimization.
−canonicalize	Built-in MLIR canonicalization pass, also performs DCE, constant folding, and operation rewrite patters, including the *local register dataflow* patters.
−count-resources	Replaces quantum gates with resource counters.
−convert-scf-to-std	Built-in MLIR conversion from structured control flow to standard dialect.
−convert-vector-to-llvm	Built-in MLIR conversion from vector to llvm dialect. (Final resource counts are printed using the `vector.print` operation.)
−convert-std-to-llvm	Built-in MLIR conversion from standard to llvm dialect.

input-to-optimization dialect lowering ⟶ program optimization ⟶ resource estimation conversion ⟶ translation to LLVM IR ⟶ LLVM compilation + linking ⟶ execution

During this process, we collect three different benchmark metrics: the compilation time (everything up to the execution step), the execution time, and the resource estimates reported by the generated executable. Note that this executable is a classical one, stripped of all quantum instructions for the purpose of resource estimation as described in Section 5.4. We thus evaluate how quickly our framework is able to perform optimizations (compilation time), how quickly it can provide resource estimates (compilation + execution time), and how effective its static optimizations are (reported resource requirements). Our results are then compared to those obtained in ProjectQ and Qiskit, where the optimization and resource estimation process used is the standard one for the respective framework, with similar optimization levels. We note that Q# does not currently offer quantum program optimizations and is thus unsuitable for this comparison.

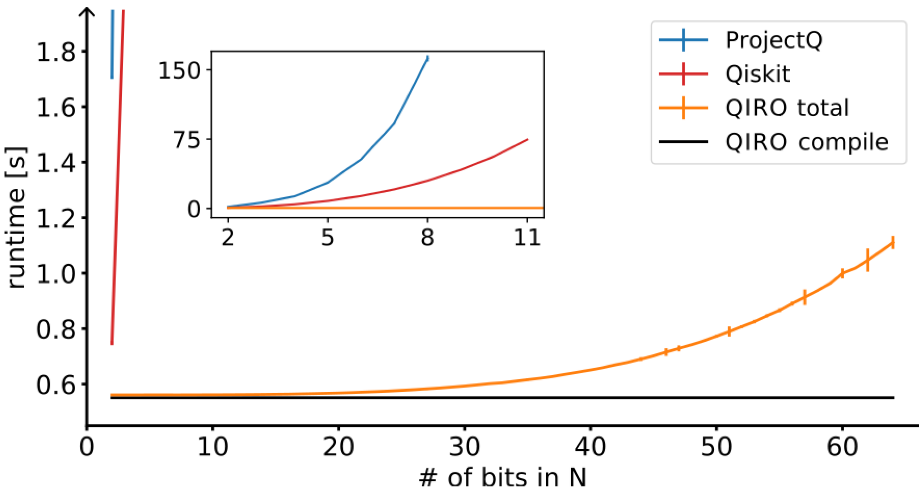


Fig. 6. Execution time to process Shor's algorithm by different frameworks and output resource counts. ProjectQ and Qiskit perform circuit optimizations at run time, whereas QIRO performs these exclusively at compile time. The number to factor is chosen as $N = 2^n - 1$ with $n \in [2, 64]$. The insert is a scaled version to show large measurement values.

6.1 Experimental Setup

Benchmarking was performed on an Intel Core i7-7700HQ @ 3.5GHz running Windows 10 build 18363. The software packages used include: Python 3.7.6, ProjectQ 0.5.1, Qiskit 0.23.1, LLVM/Clang 10.0.0, and MLIR built from source from the master branch dated 2020/10/25. Program transformation at the MLIR level is handled by an adapted version of the modular MLIR optimizer (mlir-opt). Further tools are used in the translation from MLIR to LLVM IR (mlir-translate), the compilation of LLVM IR by the LLVM static compiler (llc), and the linking phase (clang). Table 4 shows the built-in and custom QIRO passes used in the benchmark. Execution was timed using the `hyperfine`[7] command-line tool, using the median and standard deviation from each sample set.

6.2 Optimization & Resource Estimation Run Time

A comparison of the time it takes to optimize and obtain resource counts for Shor's algorithm is shown in Figure 6. The numbers to factor were chosen as $N = 2^n - 1$, where $n \in \{2, \ldots, 64\}$ is the number of bits. Inputs beyond 64-bit integers are also possible but would require to adapt our implementation to larger fixed-precision or infinite-precision arithmetic. As shown, compilation in QIRO including all implemented optimizations took a mere 551±4 ms, independent of n. Moreover, the run time of the resource estimation executable on the entire input range is situated between 10.2±0.4 ms to 560±20 ms. By contrast, inputs to ProjectQ were limited to $n = 8$ due to execution times (optimization + resource estimation) reaching 162±3 s, an almost 300-fold increase over QIRO's total running time on the same input (0.561±0.004 s). Similarly, Qiskit reaches 74.4±0.1 s on 11 qubits for optimization and resource estimation, a 130-fold increase over QIRO (0.562±0.004 s).

While some of the difference in the measured run times may be attributed to different implementation details (e.g. loop-heavy operations in Python compared to MLIR's C++ implementation), a

[7]https://github.com/sharkdp/hyperfine.

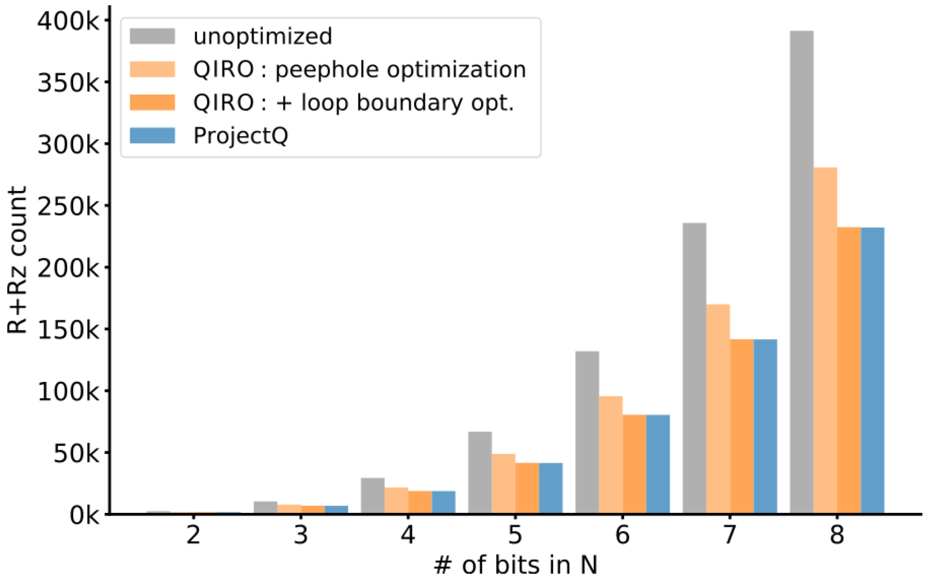


Fig. 7. Fraction of optimization opportunities identified by QIRO at compile time. Rotation gate counts for Shor's algorithm factoring $N = 2^n - 1$, with $n \in [2, 8]$, are compared to those obtained in ProjectQ using run-time optimizations.

remarkable distinction of conceptual nature is the fact that the run time of optimizations in QIRO is independent of program inputs, and thus inherently more efficient. This can be attributed to the fact that QIRO is able to perform optimizations at compile time, while the two other frameworks require program input propagation and control flow resolution before performing optimizations.

We note that true application-scale inputs for breaking RSA are expected to be $n \sim 2000$. Extrapolating our measurements to 2000 bits based on a polynomial of degree 4 (as the number of gates scales quartically), we roughly estimate to be able to process Shor's algorithm in QIRO on the order of a week, rather than upwards of 10^4 years in ProjectQ and 10^3 years in Qiskit. To further speed up resource estimation, we plan to implement custom compiler passes such as the ones proposed by Meuli et al. [23].

6.3 Effectiveness of Static Optimizations

For this benchmark, we implemented a series of simple peephole optimizations and measure their effectiveness in QIRO compared to ProjectQ. Our focus is not on the effectiveness of the optimizations themselves, but rather to investigate the difference in effectiveness when the same optimizations are performed at compile time rather than run time. The implemented optimizations include Hermitian gate cancellation on the native gate set, generalized adjoint cancellation via meta-operations (including on user-defined circuits), successive rotation gate merging, and loop boundary optimization. As shown in Figure 7, practically all ($\sim$99.8% at $n = 8$) optimization opportunities on rotation gates exploited by ProjectQ are also identified by our optimizer at compile time, not least due to our ability to statically optimize across loop boundaries, without which only $\sim 69.4\%$ would be identified. Our approach comes with the additional benefit that the optimizer's execution time is independent of a program's input size, making it especially promising for optimization of very large-scale quantum programs.

7 RELATED WORK

7.1 Intermediate Representations

To the best of our knowledge, the results presented by McCaskey and Nguyen [22] have been the only other effort to leverage the MLIR framework for quantum compilation. We see their work as complementary to ours, as it strictly focuses on translation aspects of compiling quantum programs down from QASM to the LLVM-based QIR, without considering the MLIR-based representation as a platform for quantum program optimization.

The recently introduced **Quantum Intermediate Representation (QIR)** [10] is another IR specifically crafted as a language- and hardware-agnostic intermediate representation for integrated classical-quantum programs. QIR is a set of specifications for representing quantum programs in the LLVM IR, with the goal of leveraging the performant LLVM compiler infrastructure. However, we note that the employed memory-semantics for quantum operations limits reuse of optimization passes that rely on dataflow analysis – a problem that we address by introducing a separate optimization dialect in MLIR.

LLVM IR has also been used in the ScaffCC compiler [18] for the C-based Scaffold programming language [17]. However, Scaffold programs are restricted to descriptions of fully specified circuits, i.e. all classical control flow present in the input must be statically resolvable to produce flattened circuits. This constitutes a limitation that is not present in our work.

Most existing quantum programming languages represent quantum circuits as gate lists or DAGs, e.g. Qiskit [9], ProjectQ [31], pyQuil [30], and Cirq [6]. As a result, static co-optimization of quantum-classical programs with nontrivial control flow is infeasible. As a remedy, these frameworks usually employ run-time optimizations. However, the execution time of optimizations then scales with problem size, as can be seen in Figure 6. This makes these approaches ill-suited for quantum program optimization at application scale.

7.2 Quantum Program Optimization

In addition to quantum analogs of classical optimizations (e.g. constant-folding at different levels of abstraction [15]), there exists a host of quantum-specific optimizations that are mostly targeted at quantum circuits: Circuit synthesis may be employed to re-synthesize small subcircuits [3, 8, 16, 26]. Furthermore, optimization using phase polynomials has proven to be effective [2], especially when combined with other heuristics to tackle larger universal circuits [24]. Moreover, assertion-based optimization has been proposed to optimize quantum programs at higher levels of abstraction [13].

All of these optimization algorithms may be integrated into QIRO as transformation passes. Indeed, we have implemented a generalization of constant-folding and we show that it successfully reduces the resource requirements of our implementation of Shor's algorithm that is based on the work by Beauregard [4]. Further optimizations could be applied in our IR, for example to quantum circuit definitions or to loop bodies that are free of control flow.

To the best of our knowledge, optimizations that explicitly target mixed quantum-classical programs do not exist yet, in part due to the lack of IRs that are capable of representing such programs. Our IR may thus enable such quantum-classical optimizations. For example, assertion-based optimization may be extended to take branching and loop conditions into account [13].

8 CONCLUSION

Our proposed multi-level IR for quantum computing is specifically targeted at quantum-classical co-optimization. In contrast to previous work, it supports carrying out such optimizations at application scale, allowing for optimized resource estimates of large-scale quantum programs.

Moreover, QIRO supports quantum-specific optimization passes that may fully leverage the infrastructure provided by MLIR. Crucially, the employed value-semantics in the optimization dialect directly exposes quantum data dependencies. In addition to reuse of existing components, this may enable future quantum program optimizations that leverage dataflow analysis.

APPENDICES

A MAPPING Q# TO QIRO

In this section, we discuss in detail how to map a quantum program in Q# to our input dialect. We choose Q# as an example front-end for its completeness, most notably with respect to its support for mixed quantum-classical programs.

A.1 Organization

Q# code lives inside (non-nestable) namespaces. We can map these to MLIR modules as they serve a similar purpose. Furthermore, Q# allows to bring symbols defined in other namespaces into the current one with an *open* directive. In this case, the modules should be nested inside the main execution module. When resolving symbols, the front-end should append the corresponding MLIR module identifiers to all references to symbols in those external namespaces, according to the following syntax: `@ModuleName::@SymbolName`. The inside of namespaces is composed of global variable definitions, callable definitions such as $operations$ and $functions$, and invocations of callables. How these constructs map to QIRO is described below.

A.2 Data Types

Immutable let bindings and mutable variable assignments in Q# are treated no different in the IR, both which can be mapped to value definition statements. Note that statically, due to MLIR's SSA structure, every value is already defined precisely once.

A.2.1 Numeric. MLIR provides standard integer and floating point types of arbitrary bit-width. Literals can be passed to operations that accept arguments in the form of *attributes*, if not, they must first be bound to a value with the `constant` op. Arithmetic expressions can be represented with the appropriate operations from the standard dialect.

A.2.2 Boolean. Booleans should be represented by the `i1` type.

A.2.3 Qubit. Q#'s Qubit type directly maps to the one present in QIRO. New qubits are created in Q# with `Qubit()`, which translates to a `%q = q.alloc : !q.qubit` operation in our IR. Once qubit values go out of scope in Q#, they are automatically deallocated. The front-end should insert explicit `q.free %q : !q.qubit` operations at this point to make the qubit resource available again.

A.2.4 Arrays. Note that there is distinction in how Qubit arrays and other arrays are handled. In general, Q# can build immutable arrays out of any valid type. These are null initialized, and support slicing and concatenation, which always creates a new array with element copies. Such classical arrays might be represented by the `memref` type, see the MLIR documentation for more details.

Qubit arrays created with `Qubit[n]` must instead be represented with the Qureg type from the quantum dialect. The allocation looks as follows: `%r = q.allocreg(n) : !q.qureg<n>`, where the size attribute `n` can also be replaced with a dynamic value, in which case the type will not contain a size. Multidimensional qubit arrays are not supported, so they must be unrolled into a 1D array. Static bounds checking is implemented where possible, but for dynamic out-of-bounds accesses a run-time exception is expected. Indexing and slicing is supported at the point of use for quantum registers, as all operations accepting the Qureg type optionally also accept a register access expression for each such argument. It is composed of up to three dynamic or static values, corresponding to *start*, *stop*, *step*. If only *start* is present, a single qubit is accessed. If additionally *stop* is present, the slice [*start*, *stop*) with a step of 1 is accessed. An example operation would look like this: `q.X %r[%a, %b, 2] : !q.qureg<n>`. As with qubits, registers need to be explicitly freed when they go out of scope with `q.freereg %r : !q.qureg<n>`.

A.2.5 Tuples. While a tuple type is available in MLIR, there are no operations in the standard or quantum dialects that take advantage of them. Functions and circuits have no need for tuples as they are capable of accepting and returning multiple values. Thus the argument and return tuples of Q# callables should be deconstructed into their components.

A.2.6 Pauli. The Pauli type in Q# is used to indicate rotation axes and measurement bases. For rotations, the corresponding rotation operation should be used {*Rx*, *Ry*, *Rz*}. Single qubit measurements in bases other than the Z-basis can be simulated by conjugating the measurement with the corresponding unitary. Joint multi-qubit measurements (e.g. ZX, ZZ, ...) are currently not supported but could easily be added as additional native operations in our IR.

A.3 Quantum Gates

Built-in quantum gates (intrinsic operations) in Q# for the most part have a direct analog in QIRO. If that is not the case, the front-end must express such gates in terms of the provided ones, for which standard algorithms exist. Measurement is done with a `q.measure %q : !q.qubit` operation in the computational (Z) basis.

A.3.1 Functors. Functors in Q# are the analog to QIRO's meta-operations. These are "functions" which take in an operation and produce a new one, modifying its behaviour in the process. The default ones, *adjoint* for inverting an operation and *control* for conditioning the execution based the quantum state of control qubits, are both supported by the quantum dialect. To use them, the desired operation to be modified must be constructed without target qubits, which produces a value representing the operation. Meta-operations then accept the operation value and the target qubits as arguments. See below for an example of applying an inverted and a controlled rotation gate:

```
%R = q.R(%pi) -> !q.u1
q.adj %R, %q : !q.u1, !q.qubit
q.ctrl %R, %c, %t : !q.u1, !q.qubit, !q.qubit
```

A.4 Callables

We deal with two types of callables in Q#: *functions* and *operations*. Functions contain purely classical code, and intuitively map to functions in MLIR:

```
    func @name(arg: argT ..) -> resT.. { ... }
    call @name(arg: argT ..) : (argT..) -> resT..
```

In contrast, operations contain quantum code (alongside classical one), and this division is exactly reflected by functions and circuits in QIRO. Circuits are similar to functions except that they also operate on quantum data arguments. They can be defined and called as follows:

```
    q.circ @name(arg: argT ..) -> resT.. { ... }
    q.call @name(arg: argT ..) : argT.. -> resT..
```

In order to support meta-operations on circuits, there is also an indirect call mechanism via the `apply` operation that operates on the `!q.circ` type, or the `!q.cop<n, baseT>` type with `!q.circ` as its base type.

```
      %op = q.getval @name -> !q.circ
      %inv_op = q.adj %op : !q.circ -> !q.circ
      q.apply %inv_op(arg..) : !q.circ(argT..)
```

A.5 Conditionals

Control flow in SSA-based IRs is explicit due to their block structure, which have a single entry and exit point. If/else constructs can easily be represented in MLIR using this block structure and conditional branching.

```
cond_br %cond, ^bb1(arg..), ^bb2(arg..) # if
^bb1(arg..):                            # then
    ...
    br ^bb3(arg..)
^bb2(args..):                           # else
    ...
    br ^bb3(arg..)
```

Here we've shown a simple if-else structure, where %cond is a previously calculated Boolean condition, and `arg..` represents arbitrary block arguments. Arbitrary many else-if sections can be added by inserting more blocks creating conditional branching chains, each with a then block (True) and and a successor block (False) in the chain.

A.6 Loops

Index-based for loops in Q# are well represented by the structured control flow dialect in MLIR. The syntax goes as follows:

```
scf.for %i = <low> to <up> step <step> { ... },
```

where the lower/upper bounds and step operands are SSA values. For-each loops in Q# can be transformed to this form as well by using the length of the array being traversed as an upper bound. This is one reason that qubit register arguments to circuits always need an accompanying size argument if their size is not statically specified in the type.

Q#'s while loops (for classical loop conditions) and repeat-until-success loops (for measurement based conditions) are represented by MLIR's standard control flow via blocks and branches. A `while (i < n) { ...; i++; }` loop would be translated as follows:

```
^bb1(%i: i32):
    ...
    %ip1 = addi %i, %1 : i32
    %cond = cmpi "slt", %ip1, %n : i32
    cond_br %cond, ^bb1(%ip1), ^bb2
```

where `%1` is the result of the `constant` op with the value 1, and `^bb2` is the next block after the loop. The entering condition check is omitted for brevity.

B SHOR'S ALGORITHM IN QIRO

```
func @mod(%a: i64, %N: i64) -> i64 {
    %0 = divi_unsigned %a, %N : i64
    %1 = muli %N, %0 : i64
    %2 = subi %a, %1 : i64
    return %2 : i64
}

func @mod_exp(%b: i64, %e: i64, %N: i64) -> i64 {
    %c0 = constant 0 : i64
    %c1 = constant 1 : i64
    %c2 = constant 2 : i64
    %cond = cmpi "eq", %N, %c1 : i64
    cond_br %cond, ^ret(%c0 : i64), ^reduce

    ^reduce:
        %res = constant 1 : i64
        %base = call @mod(%b, %N) : (i64, i64) -> i64
        %cond2 = cmpi "ugt", %e, %c0 : i64
        cond_br %cond2, ^while(%base, %e, %res : i64, i64, i64), ^ret(%res : i64)

    ^while(%base_0: i64, %exp_0: i64, %res_0: i64):
        %0 = call @mod(%exp_0, %c2) : (i64, i64) -> i64
        %cond3 = cmpi "eq", %0, %c1 : i64
        %res_1 = scf.if %cond3 -> i64 {
            %1 = muli %res_0, %base_0 : i64
            %2 = call @mod(%1, %N) : (i64, i64) -> i64
            scf.yield %2 : i64
        } else {
            scf.yield %res_0 : i64
        }

        %exp_1 = shift_right_unsigned %exp_0, %c1 : i64

        %3 = muli %base_0, %base_0 : i64
        %base_1 = call @mod(%3, %N) : (i64, i64) -> i64

        %cond4 = cmpi "ugt", %exp_1, %c0 : i64
        cond_br %cond4, ^while(%base_1, %exp_1, %res_1 : i64, i64, i64), ^ret(%res_1 : i64)

    ^ret(%r: i64):
        return %r : i64
}

func @mod_inv(%C: i64, %N: i64) -> i64 {
    %c0 = constant 0 : i64
    %c1 = constant 1 : i64
    br ^while(%N, %C, %c0, %c1 : i64, i64, i64, i64)

    ^while(%r_0: i64, %old_r: i64, %s_0: i64, %old_s: i64):
        %q = divi_unsigned %old_r, %r_0 : i64
        %qr = muli %q, %r_0 : i64
        %r_1 = subi %old_r, %qr : i64

        %qs = muli %q, %s_0 : i64
        %s_1 = subi %old_s, %qs : i64

        %cond = cmpi "ne", %r_1, %c0 : i64
        cond_br %cond, ^while(%r_1, %r_0, %s_1, %s_0 : i64, i64, i64, i64), ^ret(%s_0 : i64)

    ^ret(%s: i64):
        %0 = addi %s, %N : i64
        %1 = call @mod(%0, %N) : (i64, i64) -> i64
        return %1 : i64
}

func @calc_qft_angle(%j: index) -> f64 {
    %pi = constant 3.141592653589793238 : f64
    %c1 = constant 1 : index
    %0 = addi %c1, %j : index
    %1 = shift_left %c1, %0 : index
    %2 = index_cast %1 : index to i64
    %3 = uitofp %2 : i64 to f64
    %4 = divf %pi, %3 : f64
    return %4 : f64
}

func @calc_add_angle(%i: index, %j: index) -> f64 {
    %pi = constant 3.141592653589793238 : f64
    %c1 = constant 1 : index
    %0 = subi %i, %j : index
    %1 = shift_left %c1, %0 : index
    %2 = index_cast %1 : index to i64
    %3 = uitofp %2 : i64 to f64
    %4 = divf %pi, %3 : f64
    return %4 : f64
}

func @calc_cur_a(%N: i64, %n: index, %a: i64, %i: index) -> i64 {
    %c1 = constant 1 : i64
    %c2 = constant 2 : i64
    %k = index_cast %i : index to i64
    %nbits = index_cast %n : index to i64

    %0 = muli %nbits, %c2 : i64
    %1 = subi %0, %c1 : i64
    %2 = subi %1, %k : i64
    %3 = shift_left %c1, %2 : i64
    %4 = call @mod_exp(%a, %3, %N) : (i64, i64, i64) -> i64

    return %4 : i64
}

func @calc_shor_angle(%i: index, %j: index) -> f64 {
    %mpi = constant -3.141592653589793238 : f64
    %c1 = constant 1 : index
    %0 = subi %i, %j : index
    %1 = shift_left %c1, %0 : index
    %2 = index_cast %1 : index to i64
    %3 = uitofp %2 : i64 to f64
    %4 = divf %mpi, %3 : f64
    return %4 : f64
}

// quantum fourier transform on register r
q.circ @QFT(%r: !q.qureg<>, %n : index) attributes {no_inline} {
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %c2 = constant 2 : index

    scf.for %i = %c0 to %n step %c1 {
        %0 = addi %i, %c1 : index
        %k = subi %n, %0 : index
        q.H %r[%k] : !q.qureg<>
        scf.for %j = %c0 to %k step %c1 {
            %phi = call @calc_qft_angle(%j) : (index) -> f64
            %R = q.R(%phi: f64) -> !q.u1
            %1 = addi %j, %c1 : index
            %h = subi %k, %1 : index
            q.ctrl %R, %r[%h], %r[%k] : !q.u1, !q.qureg<>, !q.qureg<>
        }
    }
    %nd2 = divi_unsigned %n, %c2 : index
    scf.for %i = %c0 to %nd2 step %c1 {
        %0 = addi %i, %c1 : index
        %j = subi %n, %0 : index
        q.SWAP %r[%i], %r[%j] : !q.qureg<>, !q.qureg<>
    }
}

// add a positive or negative number to register of size n
q.circ @addConstant(%C: i64, %r: !q.qureg<>, %n: index) {
   %c0 = constant 0 : index
   %s1 = constant 1 : index
   %c1 = constant 1 : i64

   // compute
   q.call @QFT(%r, %n) {compute} : !q.qureg<>, index

   scf.for %i = %c0 to %n step %s1 {
      %ip1 = addi %i, %s1 : index
      scf.for %j = %c0 to %ip1 step %s1 {
        %k = subi %i, %j : index
        %0 = index_cast %k : index to i64
        %1 = shift_right_signed %C, %0 : i64
        %2 = and %1, %c1 : i64
        %cond = cmpi "eq", %2, %c1 : i64
        scf.if %cond {
            %phi = call @calc_add_angle(%i, %k) : (index, index) -> f64
            q.R(%phi: f64) %r[%i] : !q.qureg<>
        }
      }
   }

   // uncompute
   %qft = q.getval @QFT -> !q.circ
   %qft_inv = q.adj %qft : !q.circ -> !q.circ
   q.apply %qft_inv(%r, %n) {uncompute} : !q.circ(!q.qureg<>, index)
}

// substract a number from register of size n
q.circ @subConstant(%C: i64, %r: !q.qureg<>, %n: index) {
    %cm1 = constant -1 : i64
    %mC = muli %C, %cm1 : i64
    q.call @addConstant(%mC, %r, %n) : i64, !q.qureg<>, index
}

// add a positive number to register modulo N
q.circ @addCmodN(%C: i64, %N: i64, %r: !q.qureg<>, %n: index) {
    %c1 = constant 1 : index
    %nm1 = subi %n, %c1 : index

    q.call @addConstant(%C, %r, %n) : i64, !q.qureg<>, index

    // compute
    q.call @subConstant(%N, %r, %n) {compute} : i64, !q.qureg<>, index
    %anc = q.alloc -> !q.qubit
    q.CX %r[%nm1], %anc {compute} : !q.qureg<>, !q.qubit
    %addOp = q.getval @addConstant -> !q.circ
    %ctrlAdd = q.ctrl %addOp, %anc : !q.circ, !q.qubit -> !q.cop<1, !q.circ>
    q.apply %ctrlAdd(%N, %r, %n) {compute} : !q.cop<1, !q.circ>(i64, !q.qureg<>, index)

    q.call @subConstant(%C, %r, %n) : i64, !q.qureg<>, index

    // uncompute
    q.X %r[%nm1] {uncompute} : !q.qureg<>
    q.CX %r[%nm1], %anc {uncompute} : !q.qureg<>, !q.qubit
    q.X %r[%nm1] {uncompute} : !q.qureg<>
    q.free %anc : !q.qubit

    q.call @addConstant(%C, %r, %n) : i64, !q.qureg<>, index
}

// subtract a positive number to register modulo N
q.circ @subCmodN(%C: i64, %N: i64, %r: !q.qureg<>, %n: index) {
    %NmC = subi %N, %C : i64
    q.call @addCmodN(%NmC, %N, %r, %n) : i64, i64, !q.qureg<>, index
}

// multiply a positive number by a register modulo N, need gcd(C, N) = 1
q.circ @mulCmodN(%C: i64, %N: i64, %r: !q.qureg<>, %n: index) {
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %np1 = addi %n, %c1 : index
    %anc = q.allocreg(%np1) -> !q.qureg<>
    %Cinv = call @mod_inv(%C, %N) : (i64, i64) -> i64

    scf.for %i = %c0 to %n step %c1 {
        %addOp = q.getval @addCmodN -> !q.circ
        %ctrlAdd = q.ctrl %addOp, %r[%i] : !q.circ, !q.qureg<> -> !q.cop<1, !q.circ>

        %0 = index_cast %i : index to i64
        %1 = shift_left %C, %0 : i64
        %2 = call @mod(%1, %N) : (i64, i64) -> i64
        q.apply %ctrlAdd(%2, %N, %anc, %np1) : !q.cop<1, !q.circ>(i64, i64, !q.qureg<>, index)
    }

    scf.for %i = %c0 to %n step %c1 {
        q.SWAP %anc[%i], %r[%i] : !q.qureg<> , !q.qureg<>
    }
    scf.for %i = %c0 to %n step %c1 {
        %subOp = q.getval @subCmodN -> !q.circ
        %ctrlSub = q.ctrl %subOp, %r[%i] : !q.circ, !q.qureg<> -> !q.cop<1, !q.circ>

        %3 = index_cast %i : index to i64
        %4 = shift_left %Cinv, %3 : i64
        %5 = call @mod(%4, %N) : (i64, i64) -> i64
        q.apply %ctrlSub(%5, %N, %anc, %np1) : !q.cop<1, !q.circ>(i64, i64, !q.qureg<>, index)
    }

    q.freereg %anc : !q.qureg<>
}

q.circ @shor(%N: i64, %a: i64) {
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %c2 = constant 2 : index

    %0 = uitofp %N : i64 to f64
    %1 = log2 %0 : f64
    %2 = ceilf %1 : f64
    %3 = fptoui %2 : f64 to i64
    %n = index_cast %3 : i64 to index
    %n2 = muli %n, %c2 : index

    %m0 = constant 0 : i1
    %meas = alloc(%n2) : memref<?xi1>
    scf.for %i = %c0 to %n2 step %c1 {
        store %m0, %meas[%i] : memref<?xi1>
    }

    %r = q.allocreg(%n) -> !q.qureg<>
    %cqb = q.alloc -> !q.qubit

    q.X %r[0] : !q.qureg<>

    scf.for %i = %c0 to %n2 step %c1 {
        %cur_a = call @calc_cur_a(%N, %n, %a, %i) : (i64, index, i64, index) -> i64

        q.H %cqb : !q.qubit
        %mulOp = q.getval @mulCmodN -> !q.circ
        %ctrlMul = q.ctrl %mulOp, %cqb : !q.circ, !q.qubit -> !q.cop<1, !q.circ>
        q.apply %ctrlMul(%cur_a, %N, %r, %n) : !q.cop<1, !q.circ>(i64, i64, !q.qureg<>, index)

        scf.for %j = %c0 to %i step %c1 {
            %cond = load %meas[%j] : memref<?xi1>
            scf.if %cond {
                %phi = call @calc_shor_angle(%i, %j) : (index, index) -> f64
                q.R(%phi: f64) %cqb : !q.qubit
            }
        }
        q.H %cqb : !q.qubit

        %m = q.meas %cqb : !q.qubit -> i1
        store %m, %meas[%i] : memref<?xi1>
        scf.if %m {
            q.X %cqb : !q.qubit
        }
    }

    %mres = q.meas %r : !q.qureg<> -> memref<?xi1>

    q.free %cqb : !q.qubit   q.freereg %r : !q.qureg<>

    // process result
}

q.circ @mlir_main(%N : i64, %a : i64) attributes {no_inline_target} {
    q.call @shor(%N, %a) : i64, i64
}
```

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley, USA.

[2] M. Amy, D. Maslov, and M. 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), 1476–1489. https://doi.org/10.1109/TCAD.2014.2341953

[3] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (June 2013), 818–830. https://doi.org/10.1109/tcad.2013.2244643

[4] Stephane Beauregard. 2003. Circuit for Shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.* 3, 2 (March 2003), 175–185. https://dl.acm.org/doi/10.5555/2011517.2011525

[5] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007

[6] Cirq Developers. 2020. Cirq. (Oct. 2020). https://doi.org/10.5281/zenodo.4062499

[7] Thomas G. Draper. 2000. Addition on a quantum computer. (2000). arXiv:quant-ph/0008033.

[8] Simon Forest, David Gosset, Vadym Kliuchnikov, and David McKinnon. 2015. Exact synthesis of single-qubit unitaries over Clifford-cyclotomic gate sets. *J. Math. Phys.* 56, 8 (2015), 082201. https://doi.org/10.1063/1.4927100

[9] Jay Gambetta, Diego M. Rodríguez, Salvador de la Puente González, Matthew Treinish, Ali Javadi-Abhari, Paul Kassebaum, Marco Pistoia, Shaohan Hu, tigerjack, Carlos Azaustre, Zlatko Minev, Travis L. Scholten, Steven Oud, Matthieu Dartiailh, Maddy Tod, Juan Cruz-Benito, Christopher J. Wood, and Albert Frisch. 2019. Qiskit: An Open-source Framework for Quantum Computing. (2019). https://doi.org/10.5281/zenodo.2573505

[10] Alan Geller. 2020. Introducing Quantum Intermediate Representation (QIR). (Sep. 2020). https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir.

[11] Craig Gidney and Martin Ekerå. 2019. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. (2019). arXiv:quant-ph/1905.09749.

[12] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177

[13] Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-based optimization of quantum programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 133 (Nov. 2020), 20 pages. https://doi.org/10.1145/3428201

[14] Thomas Häner, Samuel Jaques, Michael Naehrig, Martin Roetteler, and Mathias Soeken. 2020. Improved quantum circuits for elliptic curve discrete logarithms. In *Post-Quantum Cryptography*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer International Publishing, Cham, 425–444. https://doi.org/10.1007/978-3-030-44223-1_23

[15] Thomas Häner, Damian S. Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quantum Science and Technology* 3, 2 (2018), 020501. https://doi.org/10.1088/2058-9565/aaa5cc

[16] Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home, and Matthias Christandl. 2016. Quantum circuits for isometries. *Phys. Rev. A* 93 (Mar. 2016), 032318. Issue 3. https://doi.org/10.1103/PhysRevA.93.032318

[17] Ali JavadiAbhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Fred Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. *Scaffold: Quantum Programming Language.* Technical Report. Princeton University. https://www.cs.princeton.edu/research/techreps/TR-934-12.

[18] 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 *Proceedings of the 11th ACM Conference on Computing Frontiers (CF'14)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2597917.2597939

[19] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. 2019. Strawberry fields: A software platform for photonic quantum computing. *Quantum* 3 (March 2019), 129. https://doi.org/10.22331/q-2019-03-11-129

[20] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[21] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, Korea (South), 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[22] Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. (2021). arXiv:quant-ph/2101.11365.

[23] Giulia Meuli, Mathias Soeken, Martin Roetteler, and Thomas Häner. 2020. Enabling accuracy-aware quantum compilers using symbolic resource estimation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 130 (Nov. 2020), 26 pages. https://doi.org/10.1145/3428198

[24] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *NPJ Quantum Information* 4, 1 (2018), 1–12. https://doi.org/10.1038/s41534-018-0072-4

[25] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, UK. https://doi.org/10.1017/CBO9780511976667

[26] Adam Paetznick and Krysta M. Svore. 2014. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. *Quantum Info. Comput.* 14, 15-16 (Nov. 2014), 1277–1301. https://dl.acm.org/doi/10.5555/2685179.2685181

[27] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

[28] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[29] 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 (2017), 7555–7560. https://doi.org/10.1073/pnas.1619152114

[30] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A Practical Quantum Instruction Set Architecture. (2016). arXiv:quant-ph/1608.03355.

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

[32] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901

[33] Vera von Burg, Guang Hao Low, Thomas Häner, Damian S. Steiger, Markus Reiher, Martin Roetteler, and Matthias Troyer. 2020. Quantum computing enhanced computational catalysis. (2020). arXiv:quant-ph/2007.14460.

Received April 2021; revised July 2021; accepted August 2021