Quingo: A Programming Framework for Heterogeneous Quantum-Classical Computing with NISQ Features

X. FU, Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China
JINTAO YU, State Key Laboratory of Mathematical Engineering and Advanced Computing, China
XING SU, College of Computer, National University of Defense Technology, China
HANRU JIANG, Center for Quantum Computing, Peng Cheng Laboratory, China
HUA WU, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
FUCHENG CHENG, XI DENG, and JINRONG ZHANG, Center for Quantum Computing, Peng Cheng Laboratory, China
LEI JIN, YIHANG YANG, LE XU, and CHUNCHAO HU, School of Information Engineering, Zhengzhou University, China
ANQI HUANG, GUANGYAO HUANG, XIAOGANG QIANG, MINGTANG DENG, PING XU, and WEIXIA XU, Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China
WANWEI LIU, Department of Computing Science, College of Computer, National University of Defense Technology, China
YU ZHANG, School of Computer Science and Technology, University of Science and Technology of China, China

This work is supported by National Natural Science Foundation of China under Grant No. 61902410 (X. Fu), Grant No. 61632021 (J. Wu), Grant No. 62072176 (Y. Deng); and Grant No. 61832015 (Y. Deng); the National Key R&D Program of China under Grant No. 2018YFA0306704 (Y. Feng); and the Australian Research Council under Grant No. DP180100691 (Y. Feng).

Authors' addresses: The Quingo Development Team: X. Fu, A. Huang, G. Huang, X. Qiang, M. Deng, P. Xu, W. Xu, and J. Wu, Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, 410073, China; emails: xiangfu@quanta.org.cn, anqihuang@quanta.org.cn, guangyaohuang@quanta.org.cn, qiangxiaogang@gmail.com, mtdeng@quanta.org.cn, pingxu520@nju.edu.cn, xuweixia@nudt.edu.cn, wwliu@nudt.edu.cn, junjiewu@quanta.org.cn; J. Yu, State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China; email: yjt98765@gmail.com; X. Su, College of Computer, National University of Defense Technology, Changsha, 410073, China; H. Jiang, Yanqi Lake Beijing Institute of Mathematical Sciences and Applications; email: hanru@bimsa.cn; F. Cheng, X. Deng, and J. Zhang, Center for Quantum Computing, Peng Cheng Laboratory, Shenzhen, 518055, China; emails: chengfch@pcl.ac.cn, dengx@pcl.ac.cn, zhangjr@pcl.ac.cn; H. Wu and Y. Deng, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, 200062, China; emails: coconuteva@qq.com, yxdeng@sei.ecnu.edu.cn; L. Jin, Y. Yang, L. Xu, and C. Hu, School of Information Engineering, Zhengzhou University, Zhengzhou, 450001, China; emails: 2513570760@qq.com, 756921388@qq.com, 781288793@qq.com, 18437927318@163.com; W. Liu, Department of Computing Science, College of Computer, National University of Defense Technology, Changsha, 410073, China; email: wwliu@nudt.edu.cn; Y. Zhang, School of Computer Science and Technology, University of Science and Technology of China, Hefei, 230027, China; email: yuzhang@ustc.edu.cn; Y. Feng, Centre for Quantum Software and Information, University of Technology Sydney, Sydney, 2007, Australia; email: yuan.feng@uts.edu.au.

2643-6817/2021/12-ART19
https://doi.org/10.1145/3483528

YUXIN DENG, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
JUNJIE WU, Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China
YUAN FENG, Centre for Quantum Software and Information, University of Technology Sydney, Australia

The increasing control complexity of Noisy Intermediate-Scale Quantum (NISQ) systems underlines the necessity of integrating quantum hardware with quantum software. While mapping heterogeneous quantum-classical computing (HQCC) algorithms to NISQ hardware for execution, we observed a few dissatisfactions in quantum programming languages (QPLs), including difficult mapping to hardware, limited expressiveness, and counter-intuitive code. In addition, noisy qubits require repeatedly performed quantum experiments, which explicitly operate low-level configurations, such as pulses and timing of operations. This requirement is beyond the scope or capability of most existing QPLs.

We summarize three execution models to depict the quantum-classical interaction of existing QPLs. Based on the refined HQCC model, we propose the Quingo framework to integrate and manage quantum-classical software and hardware to provide the programmability over HQCC applications and map them to NISQ hardware. We propose a six-phase quantum program life-cycle model matching the refined HQCC model, which is implemented by a runtime system. We also propose the Quingo programming language, an external domain-specific language highlighting timer-based timing control and opaque operation definition, which can be used to describe quantum experiments. We believe the Quingo framework could contribute to the clarification of key techniques in the design of future HQCC systems.

CCS Concepts: • **Software and its engineering** → **Application specific development environments**; **Domain specific languages**; *Just-in-time compilers*; • **Computer systems organization** → *Quantum computing*; • **Theory of computation** → *Timed and hybrid models;*

Additional Key Words and Phrases: Quantum programming framework, quantum programming language, quantum compilation, NISQ, timing control

ACM Reference format:
X. Fu, Jintao Yu, Xing Su, Hanru Jiang, Hua Wu, Fucheng Cheng, Xi Deng, Jinrong Zhang, Lei Jin, Yihang Yang, Le Xu, Chunchao Hu, Anqi Huang, Guangyao Huang, Xiaogang Qiang, Mingtang Deng, Ping Xu, Weixia Xu, Wanwei Liu, Yu Zhang, Yuxin Deng, Junjie Wu, and Yuan Feng. 2021. Quingo: A Programming Framework for Heterogeneous Quantum-Classical Computing with NISQ Features. *ACM Trans. Quantum Comput.* 2, 4, Article 19 (December 2021), 37 pages.
https://doi.org/10.1145/3483528

1 INTRODUCTION

The potential in solving classically intractable problems such as decryption and quantum chemistry simulation has attracted intensive research on quantum computing. Advancement in quantum information theory, quantum hardware, and quantum control contributed to the arrival of the **Noisy Intermediate-Scale Quantum (NISQ)** [2, 42, 61] era. In the NISQ era, a quantum system can integrate dozens of noisy qubits with limited coherence time and support complex quantum-classical interaction, such as real-time feedback based on the measurement of qubits [9, 12, 47, 48]. With the increased number of qubits and enhanced control capability, the control complexity of executing quantum applications on NISQ systems grows significantly, stressing the importance of high-level **quantum programming languages (QPLs)** [19] in describing various quantum applications in the NISQ era including algorithms as well as experiments. While being connected to NISQ hardware, existing quantum programming frameworks and languages suffer from difficult mapping to today's heterogeneous quantum-classical architectures or tedious and error-prone description of quantum applications, and limited capability in describing NISQ experiments that occupies most of the qubit usage time in the NISQ era.

1.1 Support for Heterogeneous Quantum-Classical Computation

It has been shown that practical quantum computing relies on the synergy between quantum and classical computing resources, and it is a viable way to take quantum computers as coprocessors of classical computing systems in a heterogeneous system.

Since Knill proposed the **Quantum Random Access Machine (QRAM)** model (see Figure 2(a)) in 1996 [30], dozens of QPLs have been proposed to describe **heterogeneous quantum-classical computation (HQCC)**. While mapping quantum algorithms described by these QPLs to NISQ hardware for execution, they are confronted with three kinds of problems:

- The QRAM model is a neat model abstracting away implementation details. QPLs directly based on the QRAM model, such as QCL [38] and Scaffold [22], allow arbitrarily complex classical computation inserted during quantum state evolution subject to applied quantum operations. As NISQ qubits have limited coherence time, it may fail when mapping programs described in these QPLs to hardware for execution (see Section 2.2 for a detailed discussion).
- With the limited qubit coherence time in mind, some QPLs, such as Cirq [16], are designed with strict constraints put on real-time classical computation and quantum-classical interaction. Although these constraints enable a more reliable mapping of quantum algorithms to real hardware, they are over strong and disable describing features that have been demonstrated by real hardware such as real-time feedback based on the measurement of qubits [19].
- Another set of QPLs can support hardware-implementable (real-time) quantum-classical interaction with reasonable constraints. They are implemented as a **domain-specific language (DSL)** embedded in a classical language, such as PyQuil [45] embedded in Python and OpenQL [26] in C++. However, meta-programming techniques need to be used to describe real-time quantum-classical interaction, resulting in counter-intuitive and complicated code, especially when control flow structures are involved. Take the PyQuil code as shown in Code 1 as an example. It repeatedly prepares a qubit to the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and measures it until the measurement result is 0 ($|0\rangle$). To support real-time program flow control based on the measurement of a qubit, it requires the use of a dedicated `while_do` function that takes as parameters a low-level `BIT`-type value and a `Program`-type object composed of sub-circuits. However, it is preferable to implement the same logic with only high-level programming constructs in a much neater way, such as Code 2 (detailed in Section 5.3.2).

Unsatisfactory support for programming HQCC applications targeting execution on NISQ hardware is an issue that should be addressed.

1.2 Support for Quantum Experiments

The noisy nature of qubits makes it essential to repeatedly perform quantum experiments calibrating qubits and tuning quantum operations. Now and in the foreseeable future, quantum experiments would occupy most of the qubit usage time in the NISQ era. While performing quantum experiments, experimentalists have to explicitly operate some low-level configurations, such as applying pulses without well-defined quantum semantics and tuning the timing of pulses. For example, a set of pulses with the same envelope but different amplitudes are used in the Rabi experiment to calibrate x- and/or y-rotations with particular rotation angles [43, 44]. In the experiment measuring the qubit dephasing time (T_2), the intervals between the $X_{\pi/2}$ operations must be changed explicitly.

Most of the existing QPLs aim to provide high-level features to support efficient description, optimization, and verification of quantum algorithms. They intentionally abstract away hardware-dependent constraints. For example, Q# targets large-scale applications on future quantum

Code 1. PyQuil code implementing real-time flow control based on the measurement of a qubit.

```
from pyquil import Program
from pyquil.gates import *

main_prog = Program()  # Initialize the Program
reg_flag  = main_prog.declare('reg_flag', 'BIT')   # declare a 1 bit memory

main_prog += MOVE(reg_flag , 1)          # initial reg_flag to 1

inner_loop = Program()                    # Define the body of the loop
inner_loop += Program(X(0), H(0))
inner_loop += MEASURE(0, reg_flag)

main_prog.while_do(reg_flag , inner_loop) # loop until reg_flag is 0

print(main_prog)
```

Code 2. A neater way to implement real-time flow control based on the measurement of a qubit with only high-level programming constructs.

```
import config.json

operation loop(): unit {
    bool flag = true;
    using (q: qubit) {
        while (flag) {
            RX(q, PI);
            H(q);
            flag = measure(q);
        }
    }
}
```

hardware and is hardware-agnostic [19, 55]. As a result, operating low-level hardware configuration required by quantum experiments is beyond the scope or capability of most existing QPLs. Consequently, dedicated experiment environments in a classical programming language such as QCoDeS [24] or PycQED [46] are used by experimentalists (referred to as *experiment toolchain*). After qubits and quantum operations have been calibrated in this environment, the right hardware configuration and pulse parameters can be determined. They are later used by a hardware-aware conversion layer to convert compiled quantum applications to the control electronics input [8] (referred to as *application toolchain*). In this way, some quantum applications described in high-level QPLs can be executed on physical qubits.

Although workable, there are some drawbacks to using two toolchains to interact with qubits. First, two toolchains can result in an undesired capability mismatch between them. On the one hand, features well supported by the experiment toolchain may not be expressed in the application toolchain. For example, feedback based on measurement results can be supported by many experiment setups but cannot be expressed by some high-level QPLs, such as Cirq [16]. On the other hand, some features can be easily expressed in the application toolchain but are difficult to be converted to the format accepted by the experiment toolchain for hardware execution, *even if the hardware is capable of supporting this feature*. For example, although Scaffold [22] can support program flow control based on the measurement result, a feature also supported by some hardware backends [9, 12, 47, 48], there is difficulty for its compiler ScaffCC [23] in generating the accepted code with real-time control flow, as mentioned in other works [19, 29]. Second, as long as the hardware-aware conversion between the compiler and hardware is still required, quantum compilers can hardly perform thorough platform-specific optimization on all possible degrees of freedom as some low-level details are hidden from the compiler.

Required is a neat quantum programming framework and language with proper assumptions on the quantum-classical interaction, which describe HQCC algorithms and quantum experiments in a format that can be easily mapped to NISQ hardware for execution. Hopefully, this framework can satisfy the requirements of both the algorithm toolchain and experiment toolchain and bridges the gap in between.

1.3 Contribution

In order to address this issue, we propose the Quingo (pronunciation: /ˈkwiŋgo/) framework, a proposal to integrate and manage quantum-classical software and hardware to provide the programmability over HQCC applications and quantum experiments that can be mapped to NISQ hardware.

The main contributions of this article are as follows:

- We summarize three execution models (Section 2.2), namely the QRAM model, the restricted HQCC model, and the refined HQCC model, to depict the quantum-classical interaction observed in existing QPLs. Compared to the other two models, quantum languages based on the refined HQCC model can be expressive and enable a neat description of quantum-classical interaction more suitable to be mapped to the NISQ hardware for execution.
- Aiming to serve quantum computing in the NISQ era based on the refined HQCC model, we propose the Quingo framework at a system level to integrate and manage quantum-classical software and hardware to provide programmability over HQCC applications and experiments and map them to NISQ hardware.
- We propose a six-phase quantum program life-cycle model, which can guide the development, compilation, optimization, and execution of HQCC applications. This model defines how the Quingo framework integrates and manages quantum and classical computing resources.
- The common services of the Quingo framework are provided by the Quingo runtime system according to the six-phase quantum program life-cycle model. We have implemented an open-source prototype of the runtime system in Python, which can orchestrate both quantum and classical software and hardware. It allows components of the framework to focus on their genuine tasks, accordingly achieving a modular programming framework. In particular, we define a protocol for the interaction between the classical host[1] language and the quantum kernel language, which enables connecting an external DSL for HQCC, like Quingo or Q#, to any general-purpose classical programming language, such as Python or C++.
- To support quantum experiments, we propose the Quingo language, which is an external DSL[2] for quantum computing. The Quingo language highlights (1) a flexible, timer-based timing control scheme at the language level with well-defined semantics, and (2) a mechanism for primitive operation definition that enables a flexible binding between operations at the language level and concrete semantics (unitaries or pulses) on the target platform. We have implemented a prototype compiler that can handle quantum-classical interaction and translate the Quingo program into eQASM instructions.

[1] It is worth noting that the term *host* is used with two different meanings throughout this article. First, when we talk about embedded DSLs, the *host* language refers to the general-purpose language that implements the DSL. For example, Python is the host language of Qiskit. Second, when we talk about heterogeneous computing, *host* and *kernel* refer to the language, program, compiler, or hardware corresponding to the general-purpose part and the coprocessor part, respectively. This pair of terms is borrowed from the OpenCL framework. The readers should pay attention to which meaning is used according to the context.

[2] An external DSL is completely designed and implemented from scratch, not relying on any pre-existent language [36].

This article is organized as follows. After illustrating the structure of HQCC algorithms with an example, Section 2 presents and compares the three execution models summarized for HQCC. Section 3 gives an overview of the Quingo framework based on the refined HQCC model, whose key techniques are presented in Section 4. Section 5 introduces the Quingo language for describing quantum kernels of both quantum algorithms and experiments. Quingo-related implementation as well as an example is shown in Section 6. After discussing some design choices and the rationales of the Quingo framework in Section 7, we conclude in Section 8.

2 BACKGROUND

2.1 HQCC Algorithms

Able to dramatically reduce the requirement on the number of qubits and the qubit coherence time, algorithms that utilize both quantum and classical computing are highly promising in the near term with wide applications in quantum simulation and optimization. Quantum phase estimation (QPE) is a key component for a wide range of applications, such as quantum simulation [37, 41] and Shor's factoring [51], and a flavor that utilizes both quantum-classical computation, namely **iterative phase estimation (IPE)**, has been proposed recently. Compared with other flavors of phase estimation such as Kitaev QPE [27], the measurement results help IPE avoid intensive classical computing. We take IPE as an example to illustrate some properties of HQCC algorithms and show their requirement on both the language and the programming framework. This section briefly introduces the principle and procedure of the IPE algorithm, and interested readers are referred to other works [9, 10, 56] for more details.

Suppose the unitary operator U has an eigenstate $|u\rangle$ with the corresponding eigenvalue $e^{i\theta}$—that is,

$$U\,|u\rangle = e^{i\theta}\,|u\rangle\,,$$

and $|u\rangle$ can be provided, the goal of a phase estimation algorithm is to estimate the value of θ. IPE achieves this goal by utilizing only one ancillary qubit with a circuit as shown in Figure 1.

To generate an m-bit estimation of θ, a circuit consisting of m sub-circuits is required with the k-th sub-circuit generating a one-bit measurement result c_k, where $k = \{m, m{-}1, \ldots, 1\}$. These m bits together form an m-bit estimation to the targeting phase with the relationship $\theta = 2\pi \cdot 0.c_1c_2\ldots c_m$, where $0.c_1c_2\ldots c_m$ is binary representation of the value $\Sigma_{i=1}^{m} c_i \cdot 2^{-i}$. As shown in Figure 1, each sub-circuit consists of a z-rotation $[R_z(\theta_k)]$ and a controlled-$U^{2^{k-1}}$ operation sandwiched by two Hadamard operations (H) before the final measurement on the ancillary qubit. Generally, the angle of the z-rotation θ_k in the k-th sub-circuit is determined at real time using classical logic based on

$$\theta_k = (-0.c_{k+1}c_{k+2}\ldots c_m)\cdot\pi, \tag{1}$$

where $k \in \{1, 2, \ldots, m-1\}$ and $\theta_m = 0$. Note that the controlled-$U^{2^{k-1}}$ gate is supposed to be optimized instead of applying the controlled-U gate 2^{k-1} times. Otherwise, the efficiency and fidelity of this algorithm will be reduced dramatically.

The flow chart of the IPE algorithm can be outlined, as illustrated in Figure 1. This algorithm includes the following steps:

(1) *Quantum kernel preparation*: Construct the quantum kernel program required by the following step and convert it into a hardware-readable format.
(2) *Quantum execution*: Performs quantum computing. This step happens on the quantum coprocessor and includes the following sub-steps:
   (a) *Initialization*: Initialize the ancilla qubit into the state $|0\rangle$.
   (b) *Phase information extraction*: Apply the k-th sub-circuit on the qubits and the phase information will be phase-kicked back to the ancilla qubit.

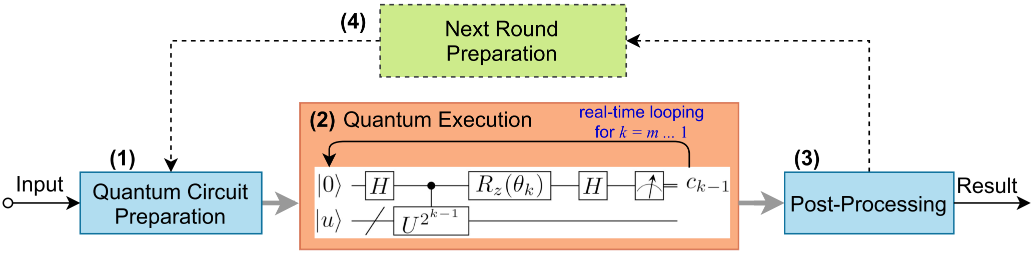


Fig. 1. Flow chart of the IPE algorithm. *Quantum execution* is executed by the quantum device and other steps by the classical computer.

- (c) *Measurement*: Measure the ancilla qubit.
- (d) *Real-time classical computing*: Compute the rotation angle θ_{k-1} using the previous measurement results based on Equation (1) for the next round of iteration.
- (e) *Iteration*: Let $k \longleftarrow k - 1$ and repeat steps (a) through (d).

(3) *Post-processing*: Classically calculate the desired result based on the measurement results.
(4) (Optional) *Search*: Calculate the new parameters based on some classical algorithm, like search.
(5) *Repetition and optimization*: Repeat steps (1) through (4) with the possible new parameters calculated in step (4) in each iteration. The repetition stops after a good enough result is retrieved.

All steps are carried out by the classical computer, except that step *Quantum execution* happens on the quantum coprocessor.

As we can see, the IPE algorithm combines classical and quantum computing in two senses.

First, *Post-processing*, *Search*, and *Quantum kernel preparation* are interleaved with *Quantum execution* where slow interaction is required (indicated by thick grey lines). In other words, the quantum state does not need to be preserved during the period when heavy classical computing happens. In some other cases like the **variational quantum eigensolver (VQE)** algorithm [41], the quantum execution needs to repeat multiple times with different quantum kernel programs, which should be calculated in step (4) (*Search*).

Second, to calculate θ_k at real time and enable efficient looping over different sub-circuits in step (2) (*Quantum execution*), classical instructions must be used, which calls for fast interaction with quantum operations during the quantum execution. To accomplish the phase estimation task with one copy of the eigenstate, the IPE algorithm needs to finish execution before the qubits holding the eigenstate lose the information. Hence, it requires real-time classical instruction to enable fast looping.

A key feature of the IPE algorithm is that not all quantum gates in the circuit can be determined in step (1) (*Quantum kernel preparation*). The z-rotations in the circuit need to be calculated at real time using classical instructions. The feature using classical computation to decide what following operations are applied on the qubits based on previous measurement results was termed *dynamic lifting* by Green et al. [17].

2.2 Execution Models of Quantum Programming Languages

QPLs are designed based on particular execution models, and the execution models largely define the interaction between quantum and classical computing. By analyzing the execution or simulation of most QPLs based on the circuit model, we summarize three execution models that can depict

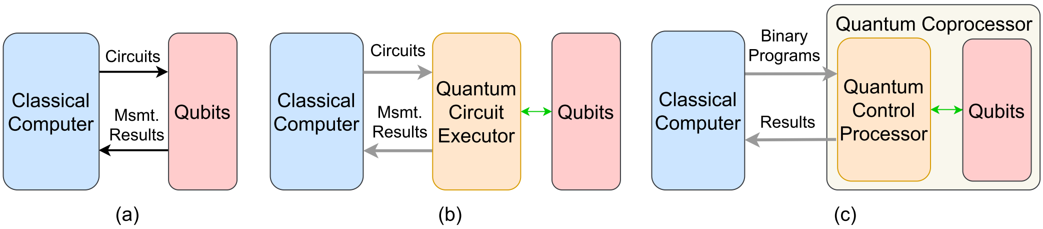


Fig. 2. Various execution models for HQCC. (a) QRAM model. (b) Restricted HQCC model. (c) Refined HQCC model. Thin black lines indicate interaction between corresponding parts abstracting away hardware details, whereas thick gray lines (thin green lines) indicate a slow (fast) interaction between corresponding parts.

possible interactions between classical and quantum computing resources in these languages. As shown in Figure 2, the three models are the QRAM model, the restricted HQCC model, and the refined HQCC model.

2.2.1 The QRAM Model. With the observation that practical quantum computing would take place on a classical machine with access to quantum registers, Knill [30] proposed the QRAM model in 1996, when there was no actual architecture for HQCC. As shown in Figure 2(a), the QRAM model consists of a classical computer and qubits or quantum registers. The classical computer performs classical computation, controls the evolution of qubits by applying operations (or quantum circuits) on the quantum register and can read back measurement results of qubits.

The QRAM model later became a canonical model that guided the design of many QPLs such as QCL [38], QPL [49], Q Language[5], Scaffold [22], LIQ$Ui|\rangle$ [58], and $\mathcal{Q}$WIRE [40].

2.2.2 Restricted HQCC Model. The restricted HQCC model arises from the early experiment setups for quantum computing. Except for flying qubits such as photons, stationary qubits are mostly controlled and measured via complex analog signals such as microwave or laser. Dedicated waveform generators, modulators, and data acquisition boards are used to generate required control signals and discriminate the measurement results of qubits. These devices usually cannot execute classical instructions for classical computation as present in quantum applications.

These analog devices are usually connected to the classical computer via, for example, Peripheral Component Interconnect Express (PCIe) bus or Ethernet. The communication between the classical computer and the analog devices is usually of substantial latency, which is introduced by signal transmission latency in cables, decoding the protocol in both hardware and software, memory access, operating system duties, and so on. As observed in experiments, it could easily take milliseconds to send a command from the classical computer to the analog device, such as Tektronix Arbitrary Waveform Generator 5014 [57], a control device that was widely used in controlling superconducting qubits. Although it is possible to reduce the latency in the communication between the classical computer and the analog devices, the required engineering effort would cost more than reasonable labor and time resources. Moreover, the achievable latency is not guaranteed because the operating system in principle has no guarantee in the response latency to an event. Therefore, the interaction latency between classical operations executed by the classical computer and the quantum operations is very likely larger than the qubit coherence time (e.g., hundreds of microseconds for superconducting qubits [28]).

By abstracting the quantum-classical interaction from this kind of quantum experiment setup, we get the restricted HQCC model, as shown in Figure 2(b). The main difference between this model and the QRAM model is the introduction of the quantum circuit executor, which can only apply a *fixed quantum circuit* on qubits within the qubit coherence time. With little classical computing power, the quantum circuit executor by itself cannot support real-time feedback based on the measurement. It is assumed that the communication between the classical computer and the quantum circuit executor has a latency comparable to or larger than the qubit coherence time. This *slow communication* is indicated by thick grey lines in Figure 2. Hence, quantum-classical interaction can only happen offline—that is, before or after the quantum circuit is applied on qubits.

The restricted HQCC model can depict the quantum-classical interaction of some popular QPLs. For example, Cirq [16] can only generate fixed quantum circuits without real-time feedback. Quipper clearly distinguishes the circuit generation time and circuit execution time. The former takes place on a classical computer and generates a fixed quantum circuit, which is executed at *real time* by the quantum processor in the latter. Although Quipper incorporates a library to support dynamic lifting, this library is built based on the assumption that *the physical quantum device has the ability to preserve qubits in long-term storage between real-time circuit invocations* [17]. However, such kind of long-term storage is beyond the capability of NISQ technology.

2.2.3 Refined HQCC Model. Addressing the issue of not supporting programmable real-time feedback based on qubit measurements, dedicated quantum control architectures [9, 12, 13, 47] containing a control processor capable of executing interleaved quantum and classical instructions have been proposed. The interleaved execution of quantum and classical instructions enables fast interaction between classical and quantum operations. Hence, it can support real-time feedback as well as structural program description based on, for example, loop and selection. We call their execution model the *refined HQCC model*, as shown in Figure 2(c).

Compared to the restricted HQCC model, the refined HQCC model inserts a control processor between the quantum circuit executor and the classical computer. The quantum control processor can execute quantum instructions that control the quantum circuit executor to apply quantum operations on qubits, and auxiliary classical instructions that update classical registers and direct the program flow. Although the control processor can execute classical instructions, it can carry out very limited real-time classical computation, as NISQ qubits have limited coherence time resulting in a very limited quantum execution time.

In the refined HQCC model, the classical computer invokes the quantum coprocessor by sending binary executables instead of quantum circuits. With similar reasons to the restricted HQCC model, the interaction between the classical computer and the quantum control processor is assumed to be slow.

This model has been adopted by heterogeneous quantum-classical programming languages like ProjectQ [52], Qiskit [21], PyQuil [45], Q# [55], OpenQL [26], and sqir [20].[3] ProjectQ and Qiskit support this model because of their support for binary control (the `Control` meta-instruction in ProjectQ and the `c_if` construct in Qiskit). Without binary control, both ProjectQ and Qiskit can only produce fixed quantum circuits in each run and could be classified into the restricted HQCC model.

sqir [20] can support programmable real-time feedback based on the qubit measurements. Targeting at serving program reasoning, sqir is too simple to support some necessary real-time classical logic constructs and cannot support describing complex classical logic happening on the classical computer and its interaction with the quantum logic, such as the *Search* step as mentioned in Section 2.1. Hence, the execution model of sqir can be summarized as the refined HQCC model without slow communication between the classical computer and the quantum control processor.

[3] At the time of writing, these languages have the following version numbers (if they have one): ProjectQ v0.5, Qiskit v0.27, PyQuil v2.28, OpenQL v0.9.

2.2.4 Comparison and Discussion. The QRAM model assumes ideal classical and quantum hardware resulting in a neat model, which enables the programmer to focus on the computational logic without worrying about implementation details. It is possible for the classical computer to decide what following operations are applied on the qubits based on previous measurement results of qubits (i.e., to implement dynamic lifting). However, the QRAM model may be not suitable for the NISQ technology, as no constraint at all is put on the quantum-classical interaction, which is very likely beyond the capability of NISQ hardware.

The restricted HQCC model respects the latency between classical computers and qubits, which results in quantum code more suitable for execution with NISQ hardware. However, as the model does not support real-time feedback or dynamic lifting, the applications that can be described in languages based on this model are significantly limited, including the IPE example as shown in Section 2.1.

In contrast, the refined HQCC model can support dynamic lifting without introducing unrealistic assumptions. In the refined HQCC model, the program using dynamic lifting is compiled into instructions, and sub-circuits within each branch form an individual code block. During quantum execution, the classical instructions can direct the control flow to the corresponding code block for execution based on the qubit measurement results that are fetched at runtime.

Although a quantum language based on a simple model like the QRAM model is more appealing to the programmer for describing quantum algorithms, the refined HQCC model can enable an easier compilation from quantum algorithms described in a high-level language to NISQ hardware for execution. We take the refined HQCC model as the underlying execution model to design the Quingo framework and the Quingo language. After the hardware and the compilation techniques get better developed in the future, we may shift from the refined HQCC model back to the QRAM model for quantum programming.

3 OVERVIEW OF THE QUINGO FRAMEWORK

Practical quantum computing relies on the synergy between quantum and classical computing resources involving various hardware and software. To this end, multiple quantum systems with different capabilities to support HQCC have been proposed. However, only part of these designs or implementations is presented publicly, a big picture of integrating the heterogeneous quantum-classical software and hardware is still missing. This requirement triggers the design of the Quingo framework.

3.1 Design Principles

The Quingo framework adopts the following design principles:

(1) *Matching NISQ technology*: To ensure the framework itself and quantum applications described based on this framework can be implemented with NISQ technology, the framework should be constructed based only on technologies that have been demonstrated.
(2) *Supporting the refined HQCC model*: The Quingo framework should enable describing quantum applications in a way that naturally follows the refined HQCC model. Classical operations that require slow communication and fast interaction with quantum operations should be easily mapped to the classical host and the control processor for execution, respectively.
(3) *Modular system with a natural integration*: The framework needs to be a modular system in which individual components, such as the host program, the kernel, and the compiler, are only responsible for their genuine tasks. In addition, various components could be naturally integrated by some managers through clearly defined interfaces.

(4) *Quantum experiment support*: The framework should support describing quantum experiments and hopefully improves the experiment efficiency.
(5) *Optimization support*: The framework should reserve as much information as possible for the compiler to perform optimization over quantum kernels with techniques such as partial execution [23].

3.2 Design Overview

Inspired by OpenCL [53], an industrial standard for parallel heterogeneous computing, and guided by the design principles, we propose the Quingo framework, a proposal at the system level to integrate and manage quantum-classical software and hardware to provide the programmability over HQCC applications and experiments and map them to NISQ hardware, as shown in Figure 3.

The Quingo framework aims to serve quantum computing in the NISQ era based on the refined HQCC model. It defines a minimum set of requirements on the hardware to support the refined HQCC model. These requirements can be satisfied by today's hardware and have been demonstrated by previous experiments (cf. *Principle 1*), and clarifies required software components with their responsibility boundary and interaction interface in the entire system. Relying on the necessary software infrastructure, the framework integrates and manages the software and hardware for HQCC, helping programmers focus on describing computational logic and map the quantum applications to the NISQ hardware. The Quingo framework comprises the following key components:

- *Quantum programs*: The quantum program is described in two parts: the classical host program and the quantum kernel.
- *Compilers*: At least two compilers should be used, including a classical compiler or interpreter, and a quantum compiler. Besides that, a pulse generator is required to generate pulses for customized operations in quantum experiments or quantum optimal control [33, 50, 59].
- *Hardware*: The Quingo hardware platform includes a classical host, a quantum coprocessor, and a shared memory accessible to both the host and the coprocessor. The quantum coprocessor includes a control processor capable of executing interleaved quantum-classical instructions to control qubits. Note that the quantum coprocessor can be simulated using simulators such as CACTUS [15] with QuantumSim [35] or QICircuit [18].
- *Runtime system*: A runtime system serving as the infrastructure of this framework that integrates and manages various quantum and classical software and hardware components.

To match the HQCC model, quantum applications based on the Quingo framework are described in two parts: the host program described in a classical language such as Python or C, and the quantum kernel described in Quingo. For example, the IPE algorithm as shown in Section 2.1 is described jointly by a Python host program (Code 3) and a Quingo kernel (Code 4) (Section 4.3 and Section 5 detail the host and kernel languages, respectively). The programmer is responsible for putting all classical computation that requires fast interaction with qubits in the kernel and the other classical tasks in the classical host program. Since the host program is fully fledged, any classical computing that does not require fast interaction is suggested to be offloaded to the host program in a classical language, which can significantly ease the burden in developing classical libraries in Quingo. This follows *Principle 2*.

Ideally, the host program should focus on describing the classical computing task and invoking the quantum kernel; the quantum kernel focuses on describing the computing task running on the quantum coprocessor; and the (quantum) compiler focuses on compiling the given Quingo program with given parameters. To get a seamlessly workable system, the framework should (i) provide an interface for the host language to call the quantum kernel with parameters,

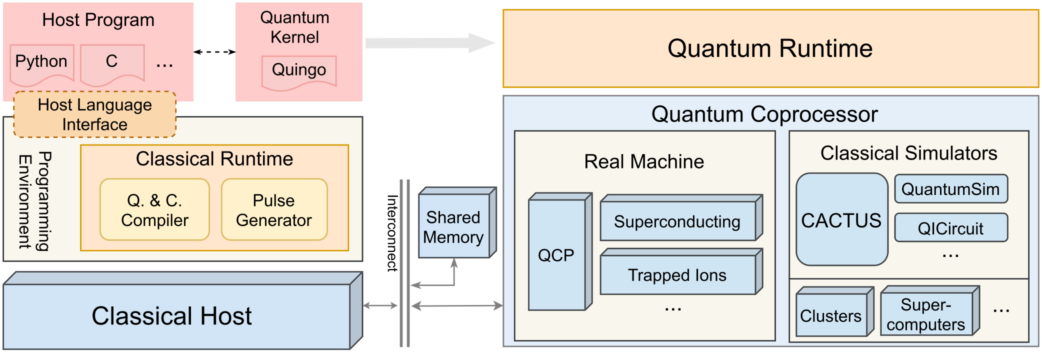


Fig. 3. Overview of the Quingo framework. Except that the interconnect is indicated by a pair of lines, other hardware is illustrated as cubes; software modules are presented as rounded rectangles; source files are as rectangles with a waving bottom line. Most components in the Quingo framework work on the classical host, except that the quantum kernel runs on the quantum coprocessor. The wide grey arrow indicates that the quantum kernel is compiled and uploaded to the quantum coprocessor or simulators for execution by the classical processor.

(ii) trigger the quantum compiler to compile the kernel, (iii) load the quantum binary to the quantum coprocessor and trigger its execution, and (iv) read the kernel execution result and return it to the host language. To this end, the Quingo runtime system is introduced. Besides that, the selection and configuration of the target backend are also done by the runtime system, instead of by the host program. In this way, the same quantum program can be executed by real hardware or simulators without any modification in the source files of the quantum program (cf. *Principle 3*). The runtime system is introduced in Section 4.2.

A key component of the Quingo framework is the language used to describe the quantum kernel. The kernel language should support the usage of opaque operations and controlling the timing control to describe quantum experiments (cf. *Principle 4*). As describing real-time classical logic using embedded DSL would result in complicated code, as illustrated in Code 1, the quantum language is suggested to be an external DSL (cf. *Principle 2*). We have designed the Quingo QPL to satisfy these requirements, which is introduced in Section 5.

Previous work has shown that quantum algorithms can be significantly optimized with partial execution [17, 23] after recognizing the different phases of a quantum program. Based on the refined HQCC model, the Quingo framework refines previous life-cycle models into a six-phase one. Considering the capability of the control processor to execute classical instructions, Quingo delays the generation of quantum code until the point immediately before quantum state evolution starts to provide as much information as possible to the quantum compiler (cf. *Principle 5*).

In the rest of the article, key techniques in the Quingo framework are introduced in the following section and the Quingo language in Section 5.

4 KEY TECHNIQUES IN THE QUINGO FRAMEWORK

We first introduce the quantum program life-cycle model in Section 4.1, which defines the routine of how the entire framework works based on the refined HQCC model. It divides the entire life of a quantum program into six phases and defines the responsibility of each component or the programmer in each phase. The software system of the framework, which is embodied in the runtime system, is introduced in Section 4.2. Section 4.3 presents the requirements and constraints that the Quingo framework put on the host language and how the host language interacts with the quantum kernel via the provided programming interface. The Quingo framework also puts some requirements on the compilation of the quantum kernel, to maximally utilize the optimization space as reserved by the six-phase life-cycle model. This is introduced in Section 4.4. Last but not least, data exchange among different components in the framework should be supported by both the software and hardware. We delay the introduction of the mechanism to exchange data after we introduce the Quingo language in Section 5.5.

4.1 Quantum Program Life Cycle

A quantum program life-cycle model can help clarify what task should be carried out by which component in what order, which can help the programmer to understand the implication of the code they write, and also guide a seamless integration of various quantum and classical components into a working system. The design of the quantum program life-cycle model aims to reserve as big optimization space over quantum programs as possible and support quantum-classical interaction with timing constraints satisfied. Based on the refined HQCC model, we propose a six-phase quantum program life-cycle model, of which the feasibility has been demonstrated by the common workflow as observed in today's quantum experiments and quantum algorithms. It includes six phases:

(1) *Editing*: Quantum programmers describe the quantum application using a classical programming language (e.g., Python or C) for the host program and Quingo for the quantum kernel. The required configuration (e.g., what primitive operations can be used by the hardware) is also provided at this stage.
(2) *Classical compilation (optional)*: A conventional compiler, such as GCC, compiles the host program and outputs a classical binary. Note that this phase may not be needed for interpreted languages such as Python.
(3) *Classical pre-execution*: The classical host executes the classical binary up to the moment calling the quantum function defined in the kernel. At this moment, all parameters used to invoke the quantum function (called the *kernel interface parameters*) have been determined.
(4) *Quantum compilation*: The quantum compiler compiles the kernel into a quantum binary consisting of quantum-classical mixed instructions with possible extra data. At this step, the quantum compiler can use the kernel interface parameters passed in to perform optimization by, for example, partial execution.
(5) *Quantum execution*: The control processor loads and executes the quantum binary. According to the executed instructions, the control processor updates classical registers, performs flow control, and applies corresponding quantum operations over qubits. In this way, the quantum state evolves under program control, accomplishing the kernel computational task. Then, the computation result is sent to the classical host by writing to the shared memory between the quantum coprocessor and the host.
(6) *Classical post-execution*: The classical binary continues execution, reads the quantum computation result, and performs possible post-processing.

When required as in algorithms like VQE, phases (3) through (6) (from classical pre-execution to classical post-execution) could be repeated multiple times to reach a good enough result.

Note that in some scenarios, the quantum compilation phase or part of it can be brought forward to happen at the same time as the classical compilation, such as when the execution of the quantum kernel does not depend on the classical parameters, and deep optimization over the quantum algorithm is not so critical compared to the requirement to execute the quantum circuit as soon as possible, such as programs used for quantum communication like teleportation [4]. Naturally, the semi-static compilation, as described in Section 4.4, might be disabled, and the quantum

Code 3. Python host of the IPE algorithm.

```python
# host.py: the host of the iterative phase estimation algorithm
from qgrtsys import if_quingo
''' Call the Quingo kernel to estimate the oracle phase.
    Repeat $n$ times, each time receive $m$ bits.
'''
def ipe(m: int, n: int) -> float:
    res = 0
    for i in range(n):
        if not if_quingo.call_quingo("ipe.qu", "ipe", m):
            raise SystemError("The execution of the quantum kernel fails.")
        res += if_quingo.read_result()
    return res / n
```

Code 4. Quingo kernel of the IPE algorithm.

```
import operations.*
import config.json.*

// kernel.qu: Iterative phase estimation algorithm.
// Input: the number of bits of the estimation
// Output: the estimation result
operation ipe(m: int) : double {
    double theta = 0.0;      // = theta_k / PI

    using (ancilla: qubit, eigenstate: qubit) { // Allocate two qubits
        if (!measure(eigenstate)) {             // prepare the eigenstate |1>
            X(eigenstate, PI);
        }
        for (int k = m - 1; k >= 0; k -= 1) {
            init(ancilla);                      // reset ancilla to |0>
            H(ancilla);

            control(ancilla, oracle(eigenstate, k));  // controlled-U^(2^i)
            Z(ancilla, -PI * theta);
            H(ancilla);

            if (measure(ancilla)) {             // Update the estimated phase
                theta = theta / 2.0 + 0.5;
            } else {
                theta /= 2.0;
            }
        }
    }
    return PI * theta;
}
```

binary generated without much optimization is loaded to the quantum control processor awaiting execution.

We take the IPE and VQE algorithm as examples to illustrate the match between the six-phase life-cycle model and quantum algorithms. The IPE algorithm is first described in a classical language and a quantum language (see Code 3 and Code 4)—this corresponds to phase (1) (*editing*). Being an interpreted language program, the Python host in Code 3 can start execution without compilation (omitting the optional phase (2): *classical compilation*). The classical host program starts execution in step (1) (*quantum kernel preparation*) and determines all parameters required by the quantum kernel (phase (3): *classical pre-execution*). Thereafter, the quantum compiler compiles the quantum kernel during which the kernel can be optimized (phase (4): *quantum compilation*). The quantum kernel starts execution utilizing quantum and classical operations with fast interaction in between in step (2) (*quantum execution*) and finishes with the measurement results generated (phase (5): *quantum execution*). The host program fetches the measured data and calculates the required result.in step (3) (*post-processing*). For VQE-like algorithms, multiple iterations of the quantum execution with different parameters are required. Based on the previously calculated result, the classical computer searches a new set of parameters in step (4) (*search*). Steps (3) and (4) (*post-processing* and *search*) together form phase (6) (*classical post-execution*). The newly generated parameters are then employed to prepare the quantum kernel to be used in the next round, which restarts the routine from phase (3) (*classical pre-execution*).

4.2 Runtime System

Since the host program, the kernel program, the compiler, and the quantum control processor in the framework are expected to be only responsible for their own tasks, they cannot directly work in collaboration due to the interaction among them. The following problems need to be resolved:

- Since the host program is not responsible for compiling the quantum program, when and which component should trigger the quantum compiler to compile the quantum program?
- The kernel interface parameters provided by the host program reside in the memory space corresponding to the process of the host program on the host machine and cannot be directly read by the quantum compiler working in another process on the host machine. How to pass the kernel interface parameters to the quantum compiler?
- Which component should upload the quantum code to the quantum control processor and trigger the execution after its generation?
- How to pass the kernel execution result back to the host program?

To get a full HQCC system with components working together seamlessly, we propose the Quingo runtime system as the supportive environment of the Quingo framework, as shown in Figure 4. The Quingo runtime system is designed as a library or daemon running on the classical host machine, which provides an **application programming interface (API)** to the classical host language and manages the interaction among the classical host and the quantum kernel at both software and hardware level. It mainly consists of five parts:

(1) A system configurator, which is in charge of configuring the execution environment for the quantum program.
(2) A host language interface, which enables the host program to call quantum kernels to utilize the quantum coprocessor to solve problems and read the result from the quantum kernel.
(3) An interface to call various quantum backends to execute the quantum code and enable them to return the computation results. This interface is implemented as various quantum backend drivers.
(4) A parameter converter and kernel result decoder, which are responsible for enabling the communication between the host and the kernel.
(5) A phase manager, which is responsible for triggering corresponding activities at different phases of the program life-cycle model.

The runtime system supports the execution of quantum programs as described in the following. Before executing the quantum program, the programmer configures the execution environment through the configurator. For example, selecting a real machine or a simulator to execute the quantum program is done at this stage. When the host program calls the quantum kernel through the host language interface (see Section 4.3), the runtime system is activated, which delivers the control to the phase manager. This moment corresponds to a time point in phase (3) (*classical pre-execution*). The phase manager then passes the parameter to the Quingo compiler (see Section 5.5.1), and triggers the compiler and the pulse generator (phase (4): *quantum compilation*). After the compiler returns the quantum code, the phase manager uploads the quantum code to the shared memory and triggers the quantum coprocessor to execute it (phase (5): (*quantum*

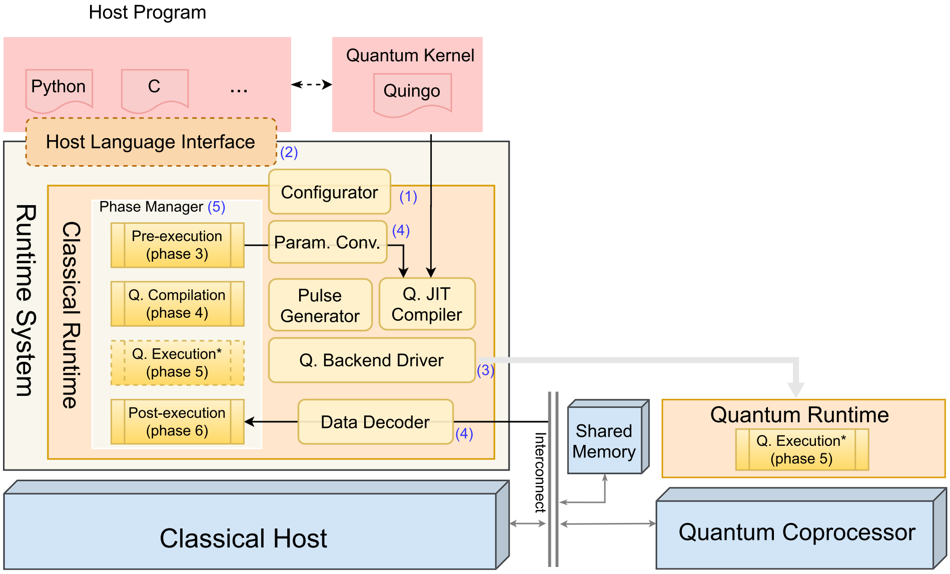


Fig. 4. Quingo runtime system supporting the six-phase quantum program life-cycle model. The number of each part of the runtime system is labeled in purple on the right side of the corresponding module. The *Q. Execution* phase in the classical runtime is framed with dashed lines, which means that the work is actually done by the coprocessor in the quantum runtime.

execution)). The phase manager then waits for the quantum kernel to return. The quantum kernel writes the execution result to a piece of the shared memory with a starting address previously defined. Thereafter, the phase manager triggers the data decoder to decode the returned result in the shared memory and returns it to the host program in a format readable for the host language for post-execution (phase (6): *classical post-execution*). The data decoder is introduced in Section 5.5. After that, the phase manager exits and returns the control to the host program.

All work done by the runtime system is transparent to the host program, the Quingo kernel, and the Quingo compiler. As a result, they can focus on their genuine roles without worrying about other details. For example, the host program only needs to describe the classical logic processing the parameters sent to and the results received from the Quingo kernel. The host program needs not learn about the concrete hardware architecture or the control methods over the quantum coprocessor. In addition, both the host program and the Quingo kernel can directly use the parameters or results passing in between without touching the communication details. We believe that this design helps ease the development of HQCC applications.

4.3 Requirements on the Host Program

In the editing phase, the programmer is responsible for describing not only the kernel in Quingo but also the host program. The Quingo framework allows any general-purpose classical language to be used as the host language, such as Python. The Quingo runtime system provides a set of APIs, currently in Python, for calling the Quingo kernel and configuring the execution environment. A classical language implementing the APIs provided by the runtime system can be an eligible host language in the Quingo framework. It offers the freedom for the programmer to choose a suitable classical language according to his/her requirements.

The host program of the IPE algorithm is shown as an example in Code 3. With the interface provided by the Quingo runtime system (`qgrtsys`), the host program calls the quantum kernel through the interface function `if_quingo.call_quingo` at line 8. The quantum function `ipe` defined in the file *kernel.qu* is called with two kernel interface parameters `m` and `n`, which are both integers in this algorithm. After the kernel finishes execution successfully, the host retrieves the kernel execution result using the function `if_quingo.read_result` for post-classical processing. lines 8 through 11 in Code 3 repeatedly call the Quingo kernel for `n` times, which is followed by a simple averaging at line 12. The actions behind this function correspond to phases (3) through (6) (*classical pre-execution* to *classical post-execution*) in the six-phase life-cycle model.

Although the host program is allowed to call quantum kernels any number of times at any point, the HQCC model assumes the quantum state will *not* be preserved between two consecutive calls to the quantum kernel. If some classical computation is required to interact with quantum operations in real time, the programmer should put these classical operations in the kernel. In this way, the timing of quantum-classical interaction can be assured.

Note that except for the methods and APIs defined by the runtime system used to call the Quingo kernel and retrieve the quantum kernel result, there are no other constraints on the host program written in the classical language. Hence, the host program can import and use any packages or libraries available in the classical language. Since classical operations requiring fast interaction with quantum operations are limited, and classical computation that only requires slow communication with qubits can be offloaded to the host program and described using fully fledged classical programming languages, the Quingo language is not required to develop libraries for complex classical computations to enhance its expressiveness for quantum applications.

4.4 Requirements on the Quantum Compiler

Each time the host program calls the quantum kernel with a set of determined parameters, the quantum compiler is called subsequently to translate and optimize the quantum kernel and generate the quantum code for execution. The quantum compiler also forms a critical component of the Quingo framework. This section introduces the requirements for the compiler of the Quingo framework. We will not discuss a concrete quantum compiler design, as it is not part of the framework.

4.4.1 Intermediate Representation. Since the refined HQCC model allows the execution of interleaved quantum and classical instructions, the quantum compiler should be capable of representing both quantum and classical computational logic to generate quantum-classical mixed binaries. Many existing quantum compilers adopt some format of quantum circuits as the **intermediate representation (IR)** of quantum applications. For example, the IR of Qiskit is a directed acyclic graph with quantum-operation nodes representing a quantum circuit. Quantum-circuit-based IR can be cumbersome when used to represent classical constructs in quantum algorithms.

Taking statements as the basic element of IR can enable a flexible description of HQCC applications with rich quantum-classical interaction [55]. As a consequence, the quantum compiler is best constructed with an IR based on statements or a hierarchical structure such as **multi-level intermediate representation (MLIR)** [32], instead of quantum circuits, to enable easy manipulation. In this way, the classical constructs or operations in the quantum kernel can be translated into classical instructions mixed with quantum instructions, which are eventually executed by the control processor.

Since manipulating the timing of operations is crucial for quantum experiments and presents increasing importance in optimization over NISQ applications [9], the IR should also support representation and manipulation of the timing of operations.

4.4.2 Semi-Static Optimization. Although it is possible to describe classical logic in the QPL, it does not mean that all classical logics described in the kernel should be translated into instructions executed on the control processor.

More information provided to the compiler usually helps with better optimization. As the kernel interface parameters are fully determined before calling the kernel, it can also enlarge the optimization space for the quantum compiler. Although not necessary, the Quingo framework suggests that the quantum compiler highly utilizes classical optimization techniques, such as procedure cloning, constant propagation, and dead code elimination [23]. In addition, partial execution is also an important technique that should be harnessed to resolve and eliminate classical operations in the quantum program.

For example, if the variable `x` in the branch statement `if(x) {X(q);} else {Y(q);}` is a kernel interface parameter, the compiler can know the actual value of `x` and then perform partial execution during compilation. Hence, the branch structure can be eliminated and the compiled result of this branch statement would be a simple quantum assembly instruction `X q` or `Y q`. By doing so, the classical instructions can be further reduced, which better matches the NISQ hardware where the classical computing power on the control processor is limited.

4.4.3 Rethinking About Quantum Compilation. In the Quingo framework, the quantum code is not generated when the host program starts execution. Quantum compilation happens after the classical host calls the quantum kernel, which is the compile time for the quantum kernel but the runtime for the classical host. We call quantum compilation happening at this stage *semi-static compilation*. By delaying the generation of quantum code to the last moment (i.e., just before its execution), the Quingo framework maximizes the optimization space reserved for the quantum compiler. To make it clear, we term phases (3) through (6) as the classical runtime and phase (5) as the quantum runtime. Quantum state evolution only happens during the quantum runtime.

Note that when applicable, optimization techniques in classical Just-In-Time (JIT) compilers could also be adopted by the quantum semi-static compiler, such as collecting statistics about how the program is actually running to rearrange and recompile for an optimal performance.

5 THE QUINGO LANGUAGE

The Quingo framework suggests an external DSL used to describe the quantum kernel. In this section, we present the Quingo language, which is designed following the requirements of the Quingo framework based on the refined HQCC model. The design of the Quingo language is guided by the following principles:

(1) *Intuitive and concise syntax*: The syntax should be intuitive to reduce the barrier for new users in learning this language. It should have native comprehensive support for structured programming and other commonly used program patterns to enable a concise description of HQCC algorithms.
(2) *Native support for fast interaction*: Classical operations and program flow control on the control processor should be naturally described without relying on dedicated variables or program structures.
(3) *Minimal design concepts*: The Quingo language aims at providing core features to support HQCC with a minimal set of concepts. Features that can be implemented using the core features will be provided by libraries. The current design of Quingo should not impede its extension to support high-level programming features in the future, such as adding control qubits to quantum operations or inverting operations.

The Quingo language adopts a two-level design to define the language to ensure the extensibility of the language and enable a practical engineering implementation. At the core level, the syntax is defined with a minimal set of concepts, data types, and rules that aim to be expressive and comprehensive enough. At the user level, syntactic sugar is constructed based on the core syntax to improve programming efficiency. The benefit of this method is double-folded. First, the core syntax being minimal enables a relatively easy and formal definition of this language and simplifies the compiler design and formalization. Second, syntactic sugar can be added by modifying the compiler frontend without affecting the core syntax and IR, which enhances the extensibility of the language while at the same time keeps the language steady. This section gives a detailed description of the Quingo language. We focus on the core syntax since the user-level syntax is still under development.

The Quingo language supports standard statements and expressions available in conventional imperative languages, as well as special syntax for describing quantum algorithms and experiments. The file syntax_core.md[4] presents the core syntax of this language in the Backus-Naur Form (BNF) format. The `import` and `package` statements make up a simple module system (Section 5.1). The Quingo language has a strong static type system where each variable is explicitly defined with a type (Section 5.2). The `using` statement is for the allocation and de-allocation of qubits (Section 5.2.2). General control flow structures are supported, including `if-else`, `while`, `break`, `continue`, and `return`. Functions in the Quingo language are called *operations*. There are two kinds of operations, `opaque` and `operation` (Section 5.3). An operation call can be associated with timing constraints to control the execution time of the operation (Section 5.4).

For demonstration purposes, we use the program shown in Code 4 as a running example throughout this section. Code 4 implements the IPE kernel to estimate the eigenvalue $e^{i\theta/2}$ of the given oracle $R_z(\theta)$ where θ is unknown. As the oracle operates only one qubit, the IPE kernel can be implemented using only two qubits.

5.1 Module System

A module system is used to organize program code in the Quingo language. Multiple operations (explained later in Section 5.3) related to a common topic can be collected into a *package* to enable separate compilation, avoid name conflict, and ease code distribution. A package is declared with the `package` statement at the top of the source file, indicating that all operations defined in this file belong to this package. Code 5 shows a code package named `operations`, in which a bunch of opaque operations are defined. Operations inside a package can be imported by other files with the `import` statement, as shown in 1 and 2 in Code 4.

Code 5. A package containing opaque operations imported by the VQE kernel.

```
package operations

opaque I(q:qubit): unit;
opaque H(q:qubit): unit;
opaque X(q: qubit, angle: double): unit;
opaque Y(q: qubit, angle: double): unit;
opaque Z(q: qubit, angle: double): unit;
opaque CNOT(ctrl: qubit, target: qubit): unit;
opaque measure(c:qubit): bool;
```

5.2 Type System

The Quingo language has a strong static type system. Types in the Quingo language include primitive types for classical and quantum data, operation types, and composite types. Besides these, the Quingo language has two special types dedicated to timing control discussed in Section 5.4.

[4] Available at https://github.com/quingo/compiler_xtext/blob/master/docs/syntax_core.md.

5.2.1 Primitive Classical Types. Considering the limited classical computational power of the control processor, the Quingo language only allows four primitive classical types: `bool`, `int`, `double`, and `unit`. The `unit` type is merely used to describe the return type of an operation that has no return value. For other classical types, basic arithmetic and logical operations are supported.

5.2.2 Primitive Quantum Type. The Quingo language defines `qubit` type as the only primitive type for quantum data. One or more qubits can be allocated from a pool with the `using` statement (e.g., line 10 in Code 4) and can later be referenced using `qubit`-type variables (e.g., the variables `ancilla` and `eigenstate` in line 10 in Code 4). Scope of a `qubit`-type variable is within the block (between the curly braces) corresponding to the `using` statement. Qubits are allocated at the beginning of the `using` block and automatically de-allocated (i.e., returned to the pool) when exiting this block. The automatic de-allocation semantics avoids leakage of the qubit resource. Qubits can only be manipulated by quantum operations (lines 11–21 in Code 4). The only way to read out information from a qubit is using a measurement operation, which projects the qubit to the computational basis state and returns the measurement result as a `bool` value (lines 11 and 21 in Code 4).

Unlike other QPLs, the Quingo language assumes the qubit is in an *unknown* state instead of $|0\rangle$ upon allocation, and the programmer is responsible for initializing it. This assumption is necessary to quantum experiments where the initialization has to be explicit.

5.2.3 Operation Type. The type of an operation consists of a parameter type and a return type. For instance, `qubit->unit` is the type of an operation on a single qubit that returns nothing. Parameter type of a multi-parameter operation would be a tuple—for example, the type of the `oracle` operation (line 18 in Code 4) is denoted as `(qubit, int)->unit`.

5.2.4 Composite Types. The Quingo language supports using *array* and *tuple* to define data collections. Any valid Quingo type can be the element type of *array* and *tuple*.

An array is an ordered sequence of elements of the same type (i.e., *array* is a homogeneous collection type). Arrays can be modified dynamically by inserting, deleting, or replacing values. Quingo arrays are jagged arrays (i.e., sub-arrays can have different lengths).

A tuple is an ordered, heterogeneous collection of elements. In the Quingo language, tuples are immutable and mostly used to pass parameters to and return values from operation calls.

5.3 Operations

The Quingo language supports functions for structured programming. Functions are called *operations* to emphasize that they are processes performing quantum operations on qubits for a particular purpose. There are two kinds of operations in the Quingo language—opaque operations and user-defined operations—defined with keywords **`opaque`** and **`operation`**, respectively.

5.3.1 Opaque Operation. In theory, there exists a set of universal quantum gates that can approximate any other quantum gates with arbitrary precision albeit at the cost of longer operation sequences using some decomposition techniques, such as repeat-until-success [39]. The universal gate set is not unique, and different quantum technologies may utilize a different primitive gate set considering the implementation difficulty. The Quingo language does not define any built-in quantum primitives. Instead, a mechanism is provided to declare platform-dependent primitive operations. The mechanism consists of two parts: (1) a platform-dependent configuration file that describes the available primitive operations and (2) opaque operation declaration statements defining the interface for these primitive operations. The second part has already been shown in Code 5. Code 6 shows part of the configuration file imported by the IPE kernel (Code 4).

The format of the configuration file is quite similar to JSON, with a leading `package` statement declaring the package name. The configuration file consists of two sections, a platform definition section (lines 3–11) and an operation definition section (lines 12–53). The former describes features of the target architecture, such as the number of available qubits, and the latter provides information about primitive operations on that architecture.

Code 6. Platform-dependent configuration file for Quingo.

```
1  package config.json
2
3  platform_def = {
4      "num_qubits": 5,
5      "single_qubit_gate_fidelity": {
6          "xy_rotations": [0.997, 0.992, 0.994, 0.996, 0.995]
7          "z_rotation": [0.985, 0.989, 0.990, 0.984, 0.977]
8      },
9      "qubit_coupling" : [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]],
10     "two_qubit_gate_fidelity": [0.985, 0.989, 0.990, 0.984, 0.977]
11 }
12 op_def = {
13     "X": {
14         "duration": 20e-9,    "num_qubits": 1,
15         "params": [{"name": "theta", "type": "double"}],
16         "semantics": {
17             "type": "rotation",
18             "rot_axis": [1, 0, 0],  # x-axis
19             "rot_angle": "theta"
20         }
21     },
22     "Y": {
23         "duration": 20e-9,    "num_qubits": 1,
24         "semantics": {
25             "type": "pulse",
26             "assembly": {"type": "eqasm", "name": "y"},
27             "pusle": {
28                 "pulse_name": "gaussian",
29                 "params": {
30                     "amplitude": 0.36,
31                     "sigma": 20e-9,
32                     "length": 4,
33                     "sample_rate": 1e9,
34                     "phase": PI/2
35                 }
36             }
37         }
38     },
39     "H": {
40         "duration": 40e-9,    "num_qubits": 1,
41         "semantics": {
42             "type": "matrix",
43             "matrix": [ [[0.707107,0.0], [0.707107,0.0]],
44                         [0.707107,0.0], [-0.707107,0.0]] ]
45         }
46     },
47     "measure": {
48         "duration": 600e-9,
49         "semantics": { "type": "measure", "assembly":"MeasZ", "return": bool }
50     }
51     # more operation declarations
52     # ......
53 }
```

An operation is defined with a few properties. The *duration* and *num_qubits* properties represent the duration the operation lasts and the number of target qubits, respectively. The Quingo language supports the definition of parametric operations. For instance, the `X` operation in Code 6 is defined as rotation along the x-axis with the rotation angle specified by the parameter `theta` (lines 16–20). The *semantics* property describes the operation's inherent quantum semantics—that is, how this operation transforms the target qubit(s) state. Code 6 presents four different ways to define the *semantics* property:

(1) A *rotation* along a specific axis by a particular angle, such as the `X` operation at 16 through 20.
(2) A *pulse* with the corresponding assembly code, such as the `Y` operation at lines 24 through 37.
(3) A unitary *matrix* describing the transformation on the state vector when applying this operation on the target qubit(s), such as the `H` operation at lines 41 through 45.
(4) The *measure* semantics dedicated to the `measure` operation (lines 47–50).

To support features exposed by the Quingo language, the IR of the Quingo compiler should be capable of representing the *matrix*, *rotation*, and *measure* semantics, and can perform code analysis and transformation based on this information. Operations with *pulse* semantics are treated as black boxes during the analysis and transformation phases. The associated information is only used by the compiler backend for code generation.

5.3.2 User-Defined Operation. User-defined operations begin with the `operation` keyword, followed by the operation name, parameter list, return type, and, finally, the operation body. The operation body is constructed in an imperative style with the basic element to be statements. With statements, it allows to freely mix quantum operations with classical operations to enable the execution of quantum operations controlled by classical program flow, such as measurement-based feedback [55].

For simplicity, the Quingo language supports structured programming with four basic but comprehensive structures [49, 60] at the core level: sequence, selection (`if-else`), loop (`while`), and recursion. Other high-level structures, such as the `for` loop, can be easily constructed as syntactic sugar based on the `while` loop. Note that classical operations in the kernel will be finally translated into classical instructions executed by the control processor, which has a fast interaction with quantum operations. Hence, conditions in the selection depending on measurement results are generated into interleaved classical and quantum instructions that are executable on today's hardware.

With the presented structures, the Quingo language provides more compact and readable code than some other languages also supporting the refined HQCC model. For example, Code 2 is the Quingo description that implements the same functionality using dynamic lifting as Code 1 in PyQuil. Since the Quingo language only describes tasks running on the quantum coprocessor, Code 2 will be compiled into interleaved quantum and classical instructions that will be executed by the control processor to implement dynamic lifting. No low-level data type and operations like `BIT` and `MOVE` and extra data structure like `Program` are required in Quingo to describe the kernel program.

5.3.3 Operation Modifier. Adding control qubits to operations and inverting operations can significantly improve the expressiveness of a QPL. They have applications in uncomputation and many oracle-based algorithms such as phase estimation. The Quingo language takes two keywords—`control` and `invert`—to support these features. We refer to them as *operation modifiers*, and they work in a similar way as higher-order functions. The `control` modifier takes a list of qubits and an operation as parameters and returns the controlled version of this input operation with the qubits being the input qubit list. The `invert` modifier simply inverts the given operation. For example, line 18 of Code 4 generates the controlled oracle using the `control` keyword. Note

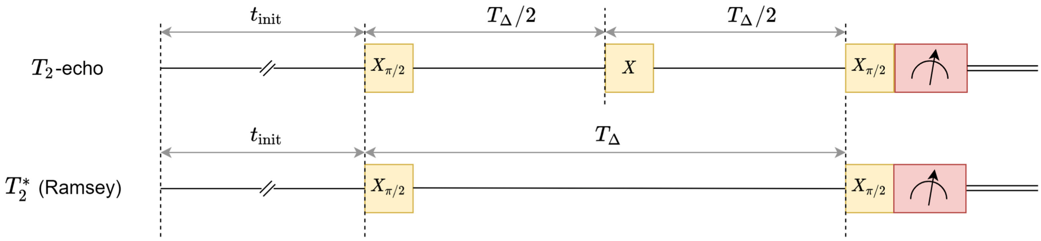


Fig. 5. Timing of operations in the T_2 experiment.

that the oracle in the `control` is written in such a way that it is like being called, which results in rather intuitive code.

5.4 Timing Control

As described in Section 2, experimentalists need to explicitly control the timing of operations in many quantum experiments. As experimental quantum systems have been scaled up to support more quantum algorithms, which are described at a rather low-level by the experimentalists, the boundary between quantum algorithms and experiments blurs, which calls for describing the timing of operations in algorithms. We use the T_2 experiment of which the operations with timing are presented in Figure 5 to demonstrate the timing control mechanism of the Quingo language. In the T_2-echo experiment, three x-rotations ($X_{\pi/2}$, X, and $X_{\pi/2}$) separated by the same interval followed by a measurement are applied to the target qubit after initialization. The intervals will be set by the experimentalist according to some properties of the calibrated qubit(s), such as the dephasing time. The T_2^* (Ramsey) experiment differs from the T_2-echo experiment only by the absence of the middle X gate. Code 7 presents the Quingo code that describes the measurement of T_2-echo or T_2^* selected by the Boolean variable `echo`.

5.4.1 Timer-Based Mechanism. Quingo introduces a timer-based scheme for controlling the timing of quantum operations. Two special types—`time` and `timer`—are added to the type system.

A `time` variable consists of a `double`-type value and a time unit such as *ns* (nanosecond), which can be used to describe a timespan. There are three ways to obtain a `time`-type value: (1) a `time` literal, (2) adding or subtracting two `time` values, and (3) scaling a `time` value by an `int` or `double` value. Line 13 in Code 7 shows an example of (3).

A timer, defined with the `timer` keyword (line 6 in Code 7), is a clock starting at a particular point. Quingo timers can be viewed as a special kind of external resource to the program. They advance automatically at the same pace regardless of the program's execution path. Timers can only be read or reset. Timers can be compared against `time` values to form timing constraints. Multiple timing constraints can be combined with logical operators `&&`. Timers are implicitly reset upon definition.

Timing of quantum operations are specified using the following syntax structure:

```
<operation> @{<timing-constraints>} !{<timer-list>}.
```

The semantics of this structure is that the operation starts its execution at a time point at which the given timing constraints enclosed in `@{...}` are satisfied, and at exactly the same moment, all timers enclosed in `!{...}` get reset. `<timing-constraint>` and `<timer-list>` are both optional in a statement.

Code 7. Quingo code for the T_2-echo and T_2^* (Ramsey) experiment.

```
1  import operations.*
2  import config.json
3
4  operation t2(intervals: time[], echo: bool) : bool[] {
5      bool[intervals.length] results;
6      timer tmr;
7      using (q : qubit) {
8          for (int i = 0; i < intervals.length; i += 1) {
9              init(q);
10             X(q, PI/2) !{tmr};
11             // Conditionally apply an X gate if this is the T2-echo experiment
12             if (echo) {
13                 X(q, PI) @{tmr == intervals[i]/2};
14             }
15             // The X_pi/2 operation below gets executed at the same time point
16             // no matter if the X_pi operation is executed or not.
17             X(q, PI/2) @{tmr == intervals[i]} !{tmr};
18
19             results[i] = measure(q) @{tmr == duration(X)};
20         }
21     }
22     return results;
23 }
```

In Code 7, the `X(q, PI/2)` operation in line 10 resets the timer `tmr`. If the experiment measures T_2-echo, the `X(q, PI)` operation in line 13 starts execution when the timer `tmr` reaches the time specified by `intervals[i]/2`. Hence, an interval of `intervals[i]/2` is inserted between the previous $X_{\pi/2}$ and this X operation. In the same way, the operation $X_{\pi/2}$ specified by line 17 starts at the time `(interval[i])` after the first $X_{\pi/2}$. The `duration` function is a Quingo intrinsic retrieving the duration of the given opaque operation. With the assistance of `duration`, operations executed back-to-back can be specified. For example, the measurement operation in line 19 starts right after the `X(q, PI/2)` operation in line 17 finishes. The same timer can appear in both the `<timing-constraints>` and `<timer-list>` parts (e.g., `tmr` in line 18), which means this timer gets reset immediately after it reaches a time point when all listed timing constraints are satisfied. Note that timing constraints can only be associated with quantum operations.

5.4.2 Solving the Timing Constraint. Through the timer-based scheme, the relative distance of different quantum operations is specified, either as a concrete value (e.g., `tmr==interval [i]/2`) or a range (e.g., `tmr>interval`). However, the underlying quantum architecture (e.g., eQASM [12]) requires a determined schedule in which every operation is associated with an absolute execution time. The compiler is responsible for choosing a schedule where all timing constraints in the quantum kernel are satisfied. Since the timing constraint can be not so strict, the same quantum operation is allowed to be applied at one of multiple timing points, and the Quingo language turns to be a non-deterministic language.

This scheduling problem can be solved by an augmented version of the list-scheduling algorithm [3] commonly used by classical compilers. The `time` and `timer` variables involved in the `<timing-constraints>` are modeled as read dependencies. Timers listed in the `<timer-list>` are modeled as write dependencies. The timing constraints are modeled as extra latency in addition to the operation duration. By adding these dependencies to the dependency graph, the list-scheduling algorithm will determine the related order of the operations, as well as compute the execution time for each quantum operation. Since timing constraints can be non-deterministic, there could be more than one legal schedule, and heuristics can be used to make the decision. If no feasible schedule can be found, the compiler will raise an error asking the programmer to modify the timing constraints in the program.

Code 8. *main.qu* generated by the runtime system when Quingo kernel ipe is called.

```
operation main(): double {
   return ipe(5);
}
```

5.5 Data Exchange

Data exchange is required between the host program and Quingo kernel. According to the six-phase quantum program life-cycle model, the host program is required to pass parameters to the Quingo kernel at the language level, and the kernel execution result should be returned to the host program. The Quingo framework defines a protocol to enable such data exchange with today's available technology. Since the quantum coprocessor can only return binary data via the shared memory to the host language, a binary format should be defined for the quantum computation result. With the well-defined data exchange protocol between the classical host and the quantum kernel, it opens the door for replacing the classical host language or the quantum kernel language, and the compiler with other classical or quantum language or compiler by supporting the communication protocol.

5.5.1 Host Program to Quingo Kernel. To decouple the data passing process from the compiler implementation, we propose a Quingo-source-file-based method, which contains two parts. First, the runtime system provides a set of interfaces in the host language, which can encode kernel interface parameters to a format defined by the data converter. The programmer is responsible for preparing the kernel interface parameters into the format by calling functions in the interface, as is done in Q# [55]. However, if the host language supports type inference, these functions are not required to be exposed to the programmer, and the underlying implementation of the `call_quingo` function can perform the conversion automatically, which simplifies the programming in the host language. The host program may pass invalid parameters (e.g., a list of elements with different types), which is unsupported by Quingo where only homogeneous arrays are supported. In this case, errors are raised during the conversion.

The converted data is written into a generated Quingo file. This file is added to the compilation and hence can be read by the compiler. For example, assume the host program calls kernel `iqe` with a parameter 5 in Code 3. The runtime system generates file *main.qu* as shown in Code 8. This file contains a `main` operation that has zero parameters. Its return type is the same as the called Quingo kernel `iqe`. Then, the compiler can read this file and retrieve the kernel interface parameters to compile the quantum kernel. The advantage of this method is to simplify the design of the Quingo compiler. Its inputs are all Quingo source files, and it does not need to provide a dedicated interface for the host program (cf. *Principle 3*).

5.5.2 Quingo Kernel to the Host Program. After the execution of a quantum program, the results are transferred back to the host by the runtime system. A shared memory space that can be accessed by both the control processor and the host is used for this data transfer. First, the Quingo kernel writes the return values to the shared memory. Next, the runtime system copies the data to the host machine. Finally, the runtime system interprets the data as the types of the host language and passes them to the host program.

The process of converting an object into a stream is normally referred to as *serialization* [54]. The byte stream that the Quingo kernel writes to the shared memory can be seen as the serialization of the return data. We propose a set of rules to define the format of the serialized data:

- For data of a primitive type, write the data in a little-endian style (i.e., the least significant bits are stored in the lowest address). The `bool` values `true` and `false` are represented by 1 and 0, respectively. The `double` numbers are serialized following the IEEE-745 single-precision floating-point standard.
- For tuple types, their elements are serialized individually and then the results are combined.
- An array is serialized into an integer value that is the offset from the current address to the actual storing region. The actual storing region starts with an integer value indicating the number of the elements in the array. The serialization result of the elements is placed after the integer.

Note that the data type is not stored in the serialization result because the runtime system can fetch the data type from the Quingo kernel source program.

This data serialization format has the advantage that the content is independent of absolute memory addresses. Note that the actual array storage location is accessed with a relative address offset. This format allows the serialized byte stream to be copied to any absolute address and maintains the same semantics. This feature is essential for the HQCC model, as the host and the control processor have different memory spaces, and the serialization result needs to be transferred from the control processor to the host.

6 IMPLEMENTATION AND EXAMPLES

6.1 Implementation

The runtime system embodies the Quingo framework. We have implemented[5] most of the runtime system as an open-source library with ~1,600 lines of Python code, including all modules except the pulse generator in the *Runtime System* block in Figure 4. Pulse generators and hardware drivers for real quantum devices are available in prevailing open-source quantum control environments, such as PycQED [46]. They can be integrated into the runtime system using the reserved interface when the Quingo framework is connected to actual quantum setups.

An open-source prototype of the Quingo compiler[6] has been implemented using the Xtext [11] framework with ~200 lines of Xtext code, ~700 lines of Xsemantics [6] code, and ~6,000 lines of Xtend code. The prototype compiler comprises a frontend and a code generator. The frontend is automatically generated from a description of the Quingo syntax in Xtext and the Quingo semantics in Xsemantics [6]. The output of the front end is an abstract syntax tree of the Quingo program, which forms the input to the code generator. While traversing the nodes of the abstract syntax tree, the code generator partially executes the quantum kernel using techniques such as constant propagation and dead code elimination. This step can simplify the abstract syntax tree of the quantum kernel by removing nodes whose values are known at static time. For nodes corresponding to quantum operations or classical operations whose operands are unknown at compile time (e.g., the value comes from a measurement of a qubit), the code generator emits corresponding quantum or classical eQASM instructions, which will be eventually executed by the control processor.

eQASM programs can be simulated by the quantum control architecture simulator CACTUS [15] connected to a quantum state simulator, like QuantumSim [35]. Since CACTUS is a cycle-accurate simulator, of which the simulation speed is relatively low, we developed a functional simulator, PyCACTUS[7] for eQASM. PyCACTUS is implemented using ~2,000 lines of Python code.

[5] The code of the runtime system can be found at https://github.com/quingo/runtime_system.

[6] The code of the prototype compiler can be found at https://github.com/quingo/compiler_xtext.

[7] The code of PyCACTUS can be found at https://github.com/gtaifu/PyCACTUS.

We have also implemented tens of examples of Quingo applications with the host language being Python including some algorithms and quantum experiments.[8]

Due to a lack of a linear IR like LLVM [31] in the prototype compiler, it is difficult to perform global analysis (including timing analysis of quantum operations) or transformation on the program. Hence, we postpone solving the timing constraint of Quingo programs in a new compiler based on MLIR [32], which is currently under development. Adding control qubits to or inverting a quantum operation would significantly improve the expressiveness of the Quingo language. This functionality has not been implemented due to limited manpower and will be implemented once we reach a stable enough compiler that can process the quantum-classical interaction with optimization.

6.2 Example

We take an example to illustrate how components in the Quingo framework work together seamlessly. The kernel and host program of the example are shown in Code 9 and Code 10, respectively. The kernel operation `sum_random` takes a list of at least two integers (`arr`) and a Boolean value (`r`) as parameters. It returns two values. The first one is the sum of integers in the list `arr`. If `r` is **`false`**, the second one is 0. Otherwise, the second one is randomly chosen from `arr[0]` or `arr[1]`, with the random flag generated from measuring an equally superposed qubit (lines 13–15).

After the host program calls the kernel with parameters (line 6 of Code 10), the runtime system converts the given parameters into the corresponding data types in Quingo, and generate a `main` operation that actually calls `sum_random` (Code 11). The phase manager then triggers the compiler to compile related Quingo files and configuration files. During compilation, the compiler performs optimization over the quantum kernel and generates a corresponding eQASM file (Code 12) for execution. Since `r` is **`false`**, the `if` body can be directly eliminated via partial execution at compile time. Together with constant propagation, other classical operations in the kernel can also be eliminated. As a result, there are neither classical or quantum instructions generated from `sum_random`. Note that when `r` is true, corresponding quantum and classical instructions will be generated to perform dynamic selection (see Code 13).

After the quantum coprocessor finishes computation and gets the result, it should serialize and store the result into the shared memory with a starting address of 0, of which the task is fulfilled by the last instructions of the eQASM program (lines 12–16 of Code 12). These special instructions are generated by the compiler when compiling the return statement of the main operation. Later, when the host language tries to retrieve the kernel computation result using `if_quingo.read_result()` (line 9 of Code 10), the runtime system retrieves the leading bytes in the shared memory (result block), decodes them into the data types of the host language, and returns them to the host program. Then, the host program can process the kernel computation result as required, such as print it (line 10 of Code 10).

7 DISCUSSION

7.1 Quantum Experiment Support of Quingo Versus Other QPLs

The core goal of the Quingo language is to support quantum experiments and assist NISQ algorithms, which differentiates Quingo from other QPLs or programming frameworks. To this end, the Quingo language introduces mechanisms to support interacting with low-level details, including the usage of opaque operations and the timing control of operations. As a result, the Quingo language becomes a QPL that is not so high level compared with many other QPLs.

[8] The Quingo examples can be found at https://github.com/quingo/quingo_examples.

Code 9. Quingo kernel implementing the accumulator and random number selector.

```
// kernel.qu
import config.json.*
import operations.*

operation sum_random(arr: int[], r: bool): (int, int) {
    int sum = 0, i = 0, random = 0;
    while (i < arr.length) {
        sum += arr[i];
        i += 1;
    }

    if (r) {
        using(q: qubit) {
            init(q);  H(q);
            if (measure(q)) { random = arr[0]; }
            else { random = arr[1]; }
        }
    }
    return (sum, random);
}
```

Code 10. Python host calling the quantum accumulator and random number selector.

```
# host.py
from qgrtsys import if_quingo
from pathlib import Path

kernel_file = Path(__file__).parent / "kernel.qu"
if not if_quingo.call_quingo(kernel_file, 'sum_random', [2, 6, 8], False):
    raise Error("failed to call the quantum kernel.")

res = if_quingo.read_result()
print("result of add example is:", res)
```

Code 11. Main operation generated by the runtime system.

```
operation main(): (int, int) {
    int[] var0_arr = {2, 6, 8};
    bool var1_bool = false;
    return sum_random(var0_arr,var1_bool);
}
```

Code 12. Generated eQASM code when the condition is false.

```
XOR r0, r0, r0
ADDI r1, r0, 1
LDI r2, 0x20000
LDUI r2, r2, 0x1
SW r0, 0x10000(r0)
FCVT.S.W f0, r0
# start of sum_random
# end of sum_random
ADDI r6, r0, 0      # a 'stack' pointer to the shared memory, used for exporting
ADDI r7, r0, 0      # a 'heap' pointer to the shared memory, used for exporting
# start exporting: (int,int)
ADDI r8, r0, 0      # load sharedAddr to the base register: r8
ADDI r9, r0, 16     # exporting: int (sum = 16)
SW r9, 0x0(r8)
ADDI r10, r0, 0     # exporting: int (random = 0)
SW r10, 0x4(r8)
STOP
```

Timing control plays a key role in quantum experiments. Nevertheless, almost all existing QPLs actively neglect the requirement on timing control since it is low-level hardware detail. An exception is OpenQL, which supports controlling the timing of quantum operations using the wait statements. However, wait statements can be cumbersome when used to describe the timing of multiple qubits if program flow controls such as loops are required. Although being straightforward for a small number of qubits, the complete semantics of wait statements is difficult to comprehend, which may result in unexpected compilation results. The Quingo language proposed a timer-based timing control scheme at the language level, which is more flexible than the wait-statement-based timing control. The underlying model is the timed automata proposed by Alur and Dill [1], of which the semantics related to timing control is well defined and easy to understand, even with the presence of classical constructs. As a result, the program with complex timing control could also be easier to write.

The semantics of quantum operations in a quantum program is assumed to be well defined in most QPLs, although their implementation might be opaque. The Quingo language supports the usage of opaque operations without well-defined quantum semantics and treats them as pulse(s) applying on one or multiple qubits with a certain duration. A dedicated configuration system is coupled with the Quingo language to bind opaque operations to concrete quantum semantics or particular pulses, and opaque operations will be treated as black boxes during compilation if no quantum semantics is provided.

7.2 Embedded Versus External DSL for HQCC

Although most QPLs are implemented as eDSLs, such as Quipper, ProjectQ, Qiskit, PyQuil, Cirq, and OpenQL, the Quingo framework advocates external DSLs for quantum computing. The rationale is the intrinsic disadvantage of using eDSLs to describe real-time quantum-classical interaction, which deeply roots in the life cycle of an eDSL-based quantum program.

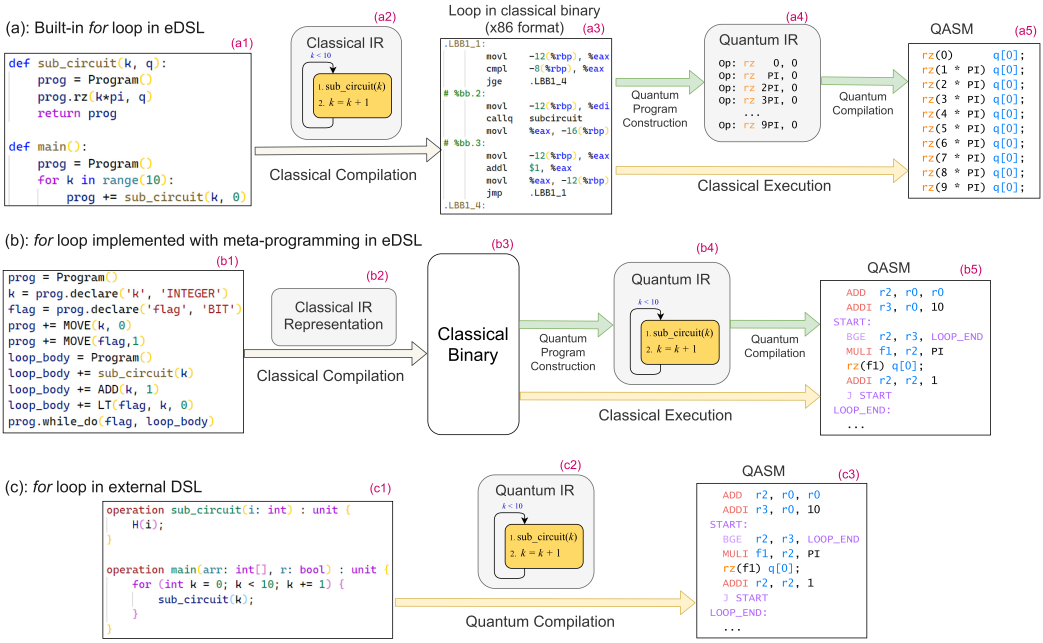
By analyzing existing eDSLs for quantum computing, we observe that eDSLs usually provide a library in the host language with an interface of several methods including

(i) instantiating objects representing quantum circuits or programs,
(ii) adding operations or sub-circuits to construct the quantum program, and
(iii) compiling and executing the quantum program.

After being written with these methods, an eDSL-based quantum program is first compiled into a classical binary by a classical compiler and then executed by the classical computer.[9] The classical execution stage starts with some classical pre-processing followed by the construction of the kernel program. Thereafter, the quantum compiler is triggered to optimize the kernel program. A quantum binary forms the final output of both the quantum compilation stage and classical execution stage, which is sent to the quantum coprocessor for execution. The life cycle of an eDSL-based quantum program is shown in Figure 6(a) (the classical pre-processing is not present in the classical execution stage for simplicity).

However, built-in classical operations and control flow structures provided by the host language, such as the addition operation and loops, *cannot* be translated into classical instructions in the quantum binary. Take the loop shown in Figure 6(a) as an example. The loop structure in the host program (a1) will be parsed into a classical IR format (a2) by the classical compiler and translated into classical instructions (a3) including `cmpl`, `jge`, `jmp`, and so on. These classical instructions are executed by the classical computer, resulting in multiple occurrences of the sub-circuit with

[9] For the sake of simplicity, we blur the difference between compiled languages and interpreted languages here and in Figure 6. The difference in these two kinds of languages does not affect the conclusion of this section.



```
def sub_circuit(k, q):
    prog = Program()
    prog.rz(k*pi, q)
    return prog

def main():
    prog = Program()
    for k in range(10):
        prog += sub_circuit(k, 0)
```

```
.LBB1_1:
        movl    -12(%rbp), %eax
        cmpl    -8(%rbp), %eax
        jge     .LBB1_4
# %bb.2:
        movl    -12(%rbp), %edi
        callq   subcircuit
        movl    %eax, -16(%rbp)
# %bb.3:
        movl    -12(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rbp)
        jmp     .LBB1_1
.LBB1_4:
```

```
Op: rz   0, 0
Op: rz  PI, 0
Op: rz 2PI, 0
Op: rz 3PI, 0
     ...
Op: rz 9PI, 0
```

```
rz(0)       q[0];
rz(1 * PI) q[0];
rz(2 * PI) q[0];
rz(3 * PI) q[0];
rz(4 * PI) q[0];
rz(5 * PI) q[0];
rz(6 * PI) q[0];
rz(7 * PI) q[0];
rz(8 * PI) q[0];
rz(9 * PI) q[0];
```

```
prog = Program()
k = prog.declare('k', 'INTEGER')
flag = prog.declare('flag', 'BIT')
prog += MOVE(k, 0)
prog += MOVE(flag,1)
loop_body = Program()
loop_body += sub_circuit(k)
loop_body += ADD(k, 1)
loop_body += LT(flag, k, 0)
prog.while_do(flag, loop_body)
```

```
  ADD  r2, r0, r0
  ADDI r3, r0, 10
START:
  BGE  r2, r3, LOOP_END
  MULI f1, r2, PI
  rz(f1) q[0];
  ADDI r2, r2, 1
  J START
LOOP_END:
  ...
```

```
operation sub_circuit(i: int) : unit {
    H(i);
}

operation main(arr: int[], r: bool) : unit {
    for (int k = 0; k < 10; k += 1) {
        sub_circuit(k);
    }
}
```

```
  ADD  r2, r0, r0
  ADDI r3, r0, 10
START:
  BGE  r2, r3, LOOP_END
  MULI f1, r2, PI
  rz(f1) q[0];
  ADDI r2, r2, 1
  J START
LOOP_END:
  ...
```

Fig. 6. Compilation process of quantum programs with various loop implementations in an eDSL or external DSL. (a) Loop implemented directly using built-in flow control structures in the host language of an eDSL. (b) Loop implemented using meta-programming techniques in an eDSL. (c) Loop implemented using flow control structures in an external DSL.

corresponding parameters in the quantum IR (a4), and hence multiple rz gates in the quantum binary (a5). That no classical instructions can be directly generated from the host language primitives forms an obstacle in describing real-time quantum-classical interaction when using eDSLs for quantum computing.

The quantum-IR-format classical operations or control flow structures are the keys to generate classical instructions in the quantum binary. eDSLs like PyQuil introduces meta-programming techniques such as the BIT data type and while_do function to build classical operations or control flow structures in the quantum IR. As shown in Figure 6(b), the eDSL source code utilizing meta-programming (b1) is parsed into classical IR (b2) and translated into classical binary (b3). Classical operations and control flow structure introduced by meta-programming will only be constructed using dedicated data structure after the classical binary is executed. Hence, these constructs can go through the classical binary layer and appear in the quantum IR (b4), which are finally translated into classical instructions in the quantum binary (b5). However, this inevitably increases the code complexity and reduces its readability. Such inefficiency is clearly demonstrated by the PyQuil implementation (Code 14) of the same IPE algorithm as the Quingo description (Code 4), where the former costs 44 lines of code, whereas the latter only costs 24 lines. In addition, describing structured programs with meta-programming in eDSLs is much less readable than external DSLs. A reader can hardly recognize the loop structure with its various components within lines 26 through 40 of Code 14 at first glance.

To enable fast quantum-classical interaction and avoid complex source code, the Quingo framework advocates using a classical language for the host program and an external DSL merely for the quantum kernel. As NISQ qubits have very short coherence time, it is unlikely to interpret and execute a quantum program on NISQ hardware. Hence, it is natural for QPLs to be compiled languages other than interpreted languages. As shown in Figure 6(c), the quantum compiler can parse the quantum kernel described in an external DSL into the quantum IR directly. In this way, classical operations and control flow structures can be kept in the quantum IR and finally translated into classical instructions in the quantum binary. Both the Q# language and Quingo language adopt this design.

Another advantage of external DSL is that it is not restricted by the syntax of an existing language. For example, the `!` operator in the Quingo language is used to denote resetting timers in the following braces. This freedom is hard, if not impossible, to achieve when using an eDSL.

7.3 HQCC Support of Quingo Versus Q#

The Q# language is also an external DSL and adopts a model similar to the refined HQCC model including three levels: the offline classical computation, quantum computation, and real-time classical computation [55]. Since Q# aims to target large-scale quantum computation for future hardware, there is no assumption about any hardware details, such as the latency difference between two kinds of quantum-classical interaction, and the capability of the control processor. Hence, programmers are not explicitly guided to put heavy classical tasks without real-time interaction with quantum operation in the classical host when programming with Q#. Take as an example the function `EstimatePhase`, which estimates the phase of an oracle iteratively [34]. The only line of code in this function for quantum computation is calling the function `ApplyIterativePhaseEstimationStep`, which is sandwiched between classical tasks including integration and array manipulation that does not require real-time interaction with quantum operations. In contrast, the control processor in the refined HQCC model would have limited classical power. Correspondingly, the Quingo language is designed with moderate classical processing capability. Hence, it can form an implicit guide for the programmer to put classical tasks not requiring fast interaction with qubits to the classical computer.

7.4 Program Life Cycle of Quingo Versus Quipper

Based on the restricted HQCC model, the Quipper program life cycle is divided into three phases, including the compile time, circuit generation time, and circuit execution time (see section 4.3.1 in the work of Green et al. [17]). Classical operations in Quipper can be only performed by the classical computer at circuit generation time, which cannot have fast interactions with quantum operations that are executed at circuit execution time. To support dynamic lifting, it needs to assume that there is long-term storage that can preserve qubits during the alternation between the circuit generation time and circuit execution time. This life cycle is not suitable for programs based on the refined HQCC model where fast interaction between classical and quantum operations is available. By utilizing the fast interaction between the control processor and qubits, the six-phase quantum program life-cycle model can naturally support dynamic lifting without assuming the long-term storage for qubits. As a result, the six-phase quantum program life-cycle model can naturally support dynamic lifting with today's available hardware.

7.5 Optimization via Semi-Static Compilation

Semi-static compilation for quantum kernels can further exploit the potential of HQCC architectures to support HQCC. Semi-static compilation utilizes classical computation power to optimize the quantum kernel, which can help improve the fidelity of quantum algorithms.

Semi-static compilation offloads the classical instructions that should be executed by the less powerful control processor to the more powerful classical host. For example, the compiler running on a classical computer performs aggressive optimization over the `sum_random` operation, resulting in zero instructions that should be executed on the quantum coprocessor for this operation, as shown by lines 7 and 8 in Code 12. Fewer operations executed by the control processor can lead to two advantages in the NISQ era. First, with a possibly shorter quantum execution time, which is highly dependent on the control processor execution time, the fidelity of the quantum program could be improved. Second, the control processor has limited processing power and memory space due to resource constraints. For example, most of today's control processors or control devices serving as a control processor are implemented with dedicated soft cores on FPGA [7, 12–14, 25, 47, 62]. The semi-static compilation can offload some classical computation to the classical host via partial execution, which enables the programmer to describe the quantum kernel with more classical computation power than what the control processor can offer.

The idea of performing aggressive optimization over the quantum program based on static analysis and partial execution was first proposed by JavadiAbhari et al. [23] with an implementation in ScaffCC based on LLVM [31]. Since its design bears no control processor in mind, loop unrolling and procedure cloning are extensively used during the compilation. The compiler cannot generate reliable code describing quantum-classical interaction with timing constraints satisfied to support flow control in real time, and the size of the generated quantum code can be big.

8 CONCLUSION AND FUTURE WORK

This article summarizes three kinds of execution models that can depict quantum-classical interaction in most QPLs. By analyzing the difference among these three models, we found the refined HQCC model can best suit NISQ computing system implementation. The refined HQCC model clarifies the difference between two kinds of quantum-classical interaction: the slow interaction between the classical host and quantum coprocessor, and the fast interaction between classical operations and quantum operations in the quantum coprocessor. To integrate and manage quantum and classical computing resources, *a framework more than a language for HQCC is required.*

This work proposes the Quingo framework at a system level based on the refined HQCC model. The Quingo framework enables the programmer to code HQCC applications in a neat programming model. With higher confidence, quantum programs described in this model can be mapped to a real quantum computing system for execution with timing constraints satisfied. By introducing a novel, six-phase quantum program life-cycle model based on the refined HQCC model, optimization space of the quantum kernel is reserved for semi-static compilation, which could lay a foundation for co-optimization of mixed quantum and classical computation.

With flexible timing control at the language level and a mechanism for primitive operation definitions, the Quingo language can be used to describe a wide range of quantum experiments. Compared to other QPLs with many high-level features, the Quingo language is a relatively low-level QPL. The Quingo language is expected to bridge the gap between various quantum software and quantum experiments, which implies a closer description of quantum algorithms to real quantum machines.

The next steps for Quingo include introducing more high-level features to Quingo to ease the programming of complex quantum algorithms, such as controlled quantum gate generation and automatic uncomputation; developing a more powerful compiler for the Quingo language based on some compilation framework, such as LLVM [31] or MLIR [32]; integrating the Quingo framework with quantum control environments, such as PycQED [46]; and demonstrating its application in quantum experiments with real hardware.

Hopefully, the Quingo framework could guide the design of an HQCC system enabling seamless collaboration between quantum and classical software and hardware in the future.

9 EXAMPLE QUINGO CODE

Code 13. Generated eQASM code when the condition is true.

```
XOR r0, r0, r0
ADDI r1, r0, 1
LDI r2, 0x20000
LDUI r2, r2, 0x1
SW r0, 0x10000(r0)
FCVT.S.W f0, r0
# start of sum_random
SMIS s0, {0}
MEASZ s0             # start of init
FMR r3, q0
ADD r4, r3, r0
BNE r4, r1, if_0_end
rx180 s0
if_0_end:            # end of init
H s0                 # H followed by msmt
MEASZ s0
FMR r5, q0           # r5 gets the random msmt result
ADDI r6, r0, 3
SW r6, 0x10022(r0)
BNE r5, r1, if_1_end
ADDI r7, r0, 5
SW r7, 0x10022(r0)
if_1_end:
LW r8, 0x10022(r0)
ADD r9, r8, r0
# end of sum_random
ADDI r10, r0, 0      # a 'stack' pointer to the shared memory, used for exporting
ADDI r11, r0, 0      # a 'heap' pointer to the shared memory, used for exporting
# start exporting: (int,int)
ADDI r12, r0, 0      # load sharedAddr to the base register: r12
ADDI r13, r0, 8      # exporting: int (c = 8)
SW r13, 0x0(r12)
SW r9, 0x4(r12)      # exporting: int (random)
STOP
```

Code 14. PyQuil implementation of IPE.

```python
def pyquil_ipe(prog: Program, controlled_oracle, m):
    eigenstate = 1
    ancilla = 0

    power = prog.declare('power', 'REAL')
    theta = prog.declare('theta', 'REAL')
    prog += MOVE(theta, 0.0)
    prog += MOVE(power, float(2 ** (m - 1)))

    ro_es = prog.declare('ro_es', 'BIT')  # prepare the eigenstate |1>
    prog += MEASURE(eigenstate, ro_es)
    prog += NOT(ro_es)
    prog.if_then(ro_es, Program(X(eigenstate)))

    ro_ancilla = prog.declare('ro_ancilla', 'BIT')
    prog += MEASURE(ancilla, ro_ancilla)

    k = prog.declare('k', 'INTEGER')  # start of for Loop
    prog += MOVE(k, m)
    flag = prog.declare('flag', 'BIT')
    prog += GT(flag, k, 0)

    loop_body = Program()
    prog.if_then(ro_ancilla, Program(X(ancilla)))  # reset ancilla to |0>

    loop_body += H(ancilla)
    loop_body += controlled_oracle(ancilla, eigenstate, power)
    loop_body += PHASE(theta, ancilla)  # Z(theta_k)
    loop_body += H(ancilla)
    loop_body += MEASURE(ancilla, ro_ancilla)

    loop_body += DIV(theta, 2)
    if_branch = Program(ADD(theta, -0.5 * pi))
    loop_body.if_then(ro_ancilla, if_branch)

    loop_body += DIV(power, 2)

    loop_body += SUB(k, 1)
    loop_body += GE(flag, k, 0)
    prog.while_do(flag, loop_body)  # end of for loop

    ro = prog.declare('ro', 'REAL')  # try to return the estimated phase
    prog += MUL(theta, -1)
    prog += MOVE(ro, theta)
```

ACKNOWLEDGMENTS

We would like to thank Center for Quantum Computing, Peng Cheng Laboratory, for their generous support. We thank Yizhi Wang and Yujie Liu for discussing NISQ algorithms (https://git.pcl.ac.cn) and Y. Yu for hosting Quingo-related code with help.

REFERENCES

[1] Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183–235.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5

[3] Kenneth R. Baker. 1974. *Introduction to Sequencing and Scheduling*. John Wiley & Sons.

[4] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters* 70, 13 (1993), 1895.

[5] Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. 2003. Toward an architecture for quantum programming. *European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics* 25 (2003), 181–200.

[6] Lorenzo Bettini. 2016. Implementing type systems for the IDE with Xsemantics. *Journal of Logical & Algebraic Methods in Programming* 85, 5pt.1 (2016), 655–680.

[7] S. Bourdeauducq, whitequark, R. Jordens, Y. Sionneau, enjoy-digital, cjbe, D. Nadlinger, et al. 2018. m-labs/artiq: 4.0. Retrieved October 14, 2021 from https://doi.org/10.5281/zenodo.1492176

[8] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature* 549 (2017), 180.

[9] Antonio D. Córcoles, Maika Takita, Ken Inoue, Scott Lekuch, Zlatko Minev, Jerry M. Chow, and Jay M. Gambetta. 2021. Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits. arXiv:2102.01682 (2021).

[10] Miroslav Dobšíček, Göran Johansson, Vitaly Shumeiko, and Göran Wendin. 2007. Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark. *Physical Review A* 76, 3 (2007), 030306.

[11] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*. ACM, New York, NY, 307–309.

[12] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, et al. 2019. eQASM: An executable quantum instruction set architecture. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, Los Alamitos, CA, 224–237.

[13] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, et al. 2017. An experimental microarchitecture for a superconducting quantum processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE, Los Alamitos, CA, 813–825.

[14] X Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, et al. 2018. A microarchitecture for a superconducting quantum processor. *IEEE Micro* 38 (2018), 40–47.

[15] Xiang Fu, Mengyu Zhang, and Peng Zhou. 2020. CACTUS: A Control Architecture Simulator. Retrieved October 14, 2021 from https://github.com/gtaifu/CACTUS.

[16] Google. 2018. Cirq: A Python Library for Writing, Manipulating, and Optimizing Quantum Circuits and Running Them Against Quantum Computers and Simulators. Retrieved October 14, 2021 from https://github.com/quantumlib/Cirq.

[17] 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)*. 333–342.

[18] Chu Guo, Yong Liu, Min Xiong, Shichuan Xue, Xiang Fu, Anqi Huang, Xiaogang Qiang, et al. 2019. General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. *Physical Review Letters* 123, 19 (Nov. 2019), 190501.

[19] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. Quantum programming languages. *Nature Reviews Physics* 2 (2020), 1–14.

[20] Kesha Hietala, Robert Rand, Shih Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[21] IBM. 2020. Qiskit: An Open-Source SDK for Working with Quantum Computers at the Level of Pulses, Circuits, and Algorithms. Retrieved October 14, 2021 from https://github.com/QISKit.

[22] Ali JavadiAbhari, Arvin Faruque, Mohammad J. Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum Programming Language.* Technical Report. Princeton University.

[23] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Computing* 45 (2015), 2–17.

[24] A. Johnson and G. Ungaretti. 2019. QCoDeS: Modular data acquisition framework. Retrieved October 14, 2021 from https://github.com/QCoDeS/Qcodes.

[25] Keysight. 2017. M3202A PXIe Arbitrary Waveform Generator, 1 GSa/s, 14 bit, 400 MHz. Retrieved October 14, 2021 from http://www.keysight.com/en/pd-2747446-pn-M3202A/pxie-arbitrary-waveform-generator-1-gs-s-14-bit-400-mhz?cc=US&lc=eng.

[26] Nader Khammassi, Imran Ashraf, J. V. Someren, Razvan Nane, A. M. Krol, M. Adriaan Rol, L. Lao, Koen Bertels, and Carmen G. Almudever. 2020. OpenQL: A portable quantum programming framework for quantum accelerators. arXiv:2005.13283 (2020).

[27] A. Yu Kitaev. 1995. Quantum measurements and the Abelian stabilizer problem. *arXiv quant-ph/9511026* (1995).

[28] Morten Kjaergaard, Mollie E. Schwartz, Jochen Braumuller, Philip Krantz, and William D. Oliver. 2020. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics* 11, 1 (2020), 369–395.

[29] V. Kliuchnikov. 2018. Wrong QASM output for teleportation circuit. Retrieved October 14, 2021 from https://github.com/epiqc/ScaffCC/issues/28.

[30] E. Knill. 1996. *Conventions for Quantum Pseudocode.* Technical Report. Los Alamos National Laboratory.

[31] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, Los Alamitos, CA, 75–86.

[32] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. arXiv: 2002.11054 (2020).

[33] Nelson Leung, Mohamed Abdelhafez, Jens Koch, and David Schuster. 2017. Speedup for quantum optimal control from automatic differentiation based on graphics processing units. *Physical Review A* 95 (2017), 042318.

[34] Microsoft. 2018. Bayesian Phase Estimation Implementation in Q#. Retrieved October 14, 2021 from https://github.com/microsoft/Quantum/blob/main/samples/characterization/phase-estimation/BayesianPhaseEstimation.qs.

[35] T. E. O'Brien, B. Tarasinski, and L. DiCarlo. 2017. Density-matrix simulation of small surface codes under current and projected experimental noise. *npj Quantum Information* 3, 1 (2017), 39.

[36] Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. 2009. Domain-specific languages—A theoretical Survey. In *Proceedings of the 3rd Conference on Compilers, Programming Languages, Related Technologies, and Applications (CoRTA'09)*. ACM, New York, NY, 35–46.

[37] P. J. J. Omalley, Ryan Babbush, Ian D. Kivlichan, Jonathan Romero, Jarrod McClean, R. Barends, J. Kelly, et al. 2016. Scalable quantum simulation of molecular energies. *Physical Review X* 6, 3 (2016), 031007.

[38] Bernhard Omer. 2003. *Structured Quantum Programming*. Institute of Information Systems, Technical University of Vienna.

[39] Adam Paetznick and Krysta M. Svore. 2014. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation* 14 (2014), 1277–1301.

[40] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A core language for quantum circuits. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. 846–858.

[41] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5 (2014), 4213.

[42] John Preskill. 2012. Quantum computing and the entanglement frontier. *arXiv:1203.5813* (2012).

[43] Isidor Isaac Rabi. 1937. Space quantization in a gyrating magnetic field. *Physical Review* 51, 8 (1937), 652.

[44] Matthew David Reed. 2013. *Entanglement and Quantum Error Correction with Superconducting Qubits*. Ph.D. Dissertation. Yale University.

[45] Rigetti. 2017. A Python library for quantum programming using Quil. Retrieved October 14, 2021 from https://github.com/rigetti/pyquil.

[46] M. A. Rol, C. Dickel, S. Asaad, C. C. Bultink, R. Sagastizabal, N. K. L. Langford, G. de Lange, et al. 2019. DiCarloLab PycQED_py3. Retrieved October 14, 2021 from https://github.com/DiCarloLab-Delft/PycQED_py3.

[47] Colm A. Ryan, Blake R. Johnson, Diego Ristè, Brian Donovan, and Thomas A. Ohki. 2017. Hardware for dynamic quantum computing. *arXiv:1704.08314* (2017).

[48] Yves Salathé, Philipp Kurpiers, Thomas Karg, Christian Lang, Christian Kraglund Andersen, Abdulkadir Akin, Sebastian Krinner, Christopher Eichler, and Andreas Wallraff. 2018. Low-latency digital signal processing for feedback and feedforward in quantum computing and communication. *Physical Review Applied* 9, 3 (2018), 034011.

[49] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14 (2004), 527–586.

[50] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. 2019. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 1031–1044.

[51] Peter W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. 124–134. https://doi.org/10.1109/SFCS.1994.365700

[52] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2, 49 (2018), 49.

[53] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73.

[54] Audie Sumaray and S. Kami Makki. 2012. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International conference on Ubiquitous Information Management and Communication (ICUIMC'12)*. ACM, New York, NY, Article 48, 6 pages.

[55] 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*. ACM, New York, NY, 1–10.

[56] Krysta M. Svore, Matthew B. Hastings, and Michael Freedman. 2013. Faster phase estimation. arXiv preprint arXiv:1304.0741 (2013).

[57] Tektronix. 2018. Arbitrary Waveform Generator AWG5000 Series. Retrieved October 14, 2021 from https://www.tek.com/datasheet/awg5000-series.

[58] Dave Wecker and Krysta M. Svore. 2014. LIQUi|⟩: A software design architecture and domain-specific language for quantum computing. arXiv:1402.4467 (2014).

[59] J. Werschnik and E. K. U. Gross. 2007. Quantum optimal control theory. *Journal of Physics B: Atomic, Molecular and Optical Physics* 40 (2007), R175.

[60] Mingsheng Ying. 2016. *Foundations of Quantum Programming*. Morgan Kaufmann.

[61] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, et al. 2020. Quantum computational advantage using photons. *Science* 370, 6523 (2020), 1460–1463.

[62] Zurich Instruments. 2020. Quantum Computing Control System. Retrieved October 14, 2021 from https://www.zhinst.com/europe/quantum-computing-control-system-qccs.

Received August 2020; revised March 2021; accepted August 2021