ABSTRACT

BADRIKE, KAUSTUBH JAGDISH. QisDAX: An Open Source Bridge from Qiskit to Trapped Ion Quantum Devices. (Under the direction of Frank Mueller).

Quantum computing has become widely available to researchers via cloud-hosted devices with different technologies using a multitude of software development frameworks. The vertical stack behind such solutions typically features quantum language abstraction and high-level translation frameworks that tend to be open source, down to pulse-level programming. However, the lower-level mapping to the control electronics, such as controls for laser and microwave pulse generators, remains closed source for contemporary commercial cloud-hosted quantum devices. One exception is the ARTIQ (Advanced Real-Time Infrastructure for Quantum physics) open-source library for trapped-ion control electronics. This stack has been complemented by the Duke ARTIQ Extensions (DAX) to provide modularity and better abstraction. It, however, remains disconnected from the wealth of features provided by popular quantum computing languages. This paper contributes QisDAX, a bridge between Qiskit and DAX that fills this gap. QisDAX provides interfaces for Python programs written using IBM's Qiskit and transpiles them to the DAX abstraction. This allows users to generically interface to the ARTIQ control systems accessing trapped-ion quantum devices. Consequently, the algorithms expressed in Qiskit become available to an open-source quantum software stack. This provides the first open-source, end-to-end, full-stack pipeline for remote submission of quantum programs for trapped-ion quantum systems in a non-commercial setting.

QisDAX: An Open Source Bridge from Qiskit to Trapped Ion Quantum Devices

by
Kaustubh Jagdish Badrike

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

Huiyang Zhou	Jianqing Liu

Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents Manisha and Jagdish Badrike.

BIOGRAPHY

Kaustubh Badrike has obtained his Bachelor's degree in Engineering from the University of Mumbai.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Quantum Computing

1.1.1 Quantum mechanics

Quantum mechanics is a fundamental theory that describes the behavior of matter and energy at the smallest scales. It refined our understanding of the physical world and led to groundbreaking discoveries in physics, including the theory of relativity and the development of nuclear energy.

One of the most intriguing aspects of quantum mechanics is the concept of superposition, which allows a particle to exist in multiple states simultaneously. Another key feature is entanglement, where two particles become connected in a way that their properties become correlated, even when separated by large distances.

1.1.2 Application in computing

Properties of quantum mechanics have given rise to a new field of technology known as quantum computing. Unlike classical computers, which use bits to store and process information, quantum computers use quantum bits, or qubits, which can exist in multiple states at once.

This property of superposition allows quantum computers to represent and manipulate multiple states at the same time, which means that they can solve certain problems faster than classical computers constrained to a single state representation. For example, an error-free quantum computer can factor large numbers exponentially faster than a classical computer, which has important implications for cryptography.

To date, quantum computers have been considered particularly useful for tasks such as financial modeling, optimization, and simulations of quantum systems. As such, quantum computing has the potential to revolutionize fields ranging from energy to drug discovery to materials science.

However, building a reliable and scalable quantum computer has proven to be a significant challenge due to the sensitivity of qubits to their environment. This has led to the development of NISQ (Noisy Intermediate-Scale Quantum) era devices, which are quantum computers that are small enough to be built with today's technology but have a limited number of qubits and high error rates.

Despite their limitations, NISQ era devices have already been used to solve important problems, such as simulating chemical reactions and optimizing financial portfolios. It is important to note that no error-free quantum computer currently exists. This means that the results produced by these devices must be carefully verified and interpreted. Furthermore, these devices are being used to develop and refine quantum algorithms and error correction techniques, which will be essential for building larger and more reliable quantum computers in the future.

1.1.3 Current scenario

Development of a practical quantum computer demands a well-designed, modular software architecture that considers all layers of a vertical stack, from the programming language to the qubit-specific hardware (Chong and Martonosi 2017; Riesebos et al. 2019; Murali et al. 2019). However, the field is currently dominated by a variety of architectures with domain-specific abstractions, leading to customized software stacks that are not easily compatible.

One source of this incompatibility comes from proprietary hardware components, such as microwave pulse generators and lasers, that are decoupled from the higher-level, hardware-agnostic layers of the software stack. Furthermore, their controls tend to be closed-source. An alternate, *open-source* design can instead be promoted by this type of decoupled stack, which enables better abstraction as well as cross-platform compatibility.

An increasingly popular, open-source quantum control system is ARTIQ (Advanced Real-Time Infrastructure for Quantum physics), a software framework with dedicated, open-source

control hardware (Bourdeauducq et al. 2016; Kasprowicz et al. 2020). The ARTIQ stack is complemented by DAX (Duke ARTIQ extensions) (Riesebos et al. 2022), a device abstraction developed to provide high-level, modular utilities for controlling trapped-ion systems. However, in ARTIQ-based quantum computers, the rich capabilities offered by popular quantum computing languages are not accessible to lower layers of the computing stack.

This work contributes QisDAX to bridge this gap. QisDAX facilitates an interface between DAX and Qiskit (ANIS et al. 2021; Wille et al. 2019), allowing the entire quantum computation workflow for a trapped ion system to be incorporated into a single open-source, full-stack pipeline. A wide variety of Qiskit algorithms and frameworks therefore become accessible to DAX users and can be applied to a new set of backend devices, such as trapped-ion systems and simulators.

By exporting results in Qiskit-compatible objects, QisDAX also facilitates classical analysis of quantum results, including processing in a hybrid environment with repeated quantum kernel invocations.

1.2 Transpilation

Transpilation, also known as source-to-source compilation, is the process of converting code written in one programming language (the source language) into another programming language (the target language). Unlike traditional compilation, which converts code into machine code that can be executed directly by a computer's processor, transpilation generates code that is still in a high-level programming language, but in a different form.

There are several popular open-source transpilation tools available, including:

- Babel (Babel 2014): A JavaScript (Ecma International 2022) transpiler that converts modern ECMAScript 6+ code into compatible code that can run in older browsers or environments that do not yet support the latest language features.

- C2Rust (Immunant and Galois 2018): C (Kernighan and Ritchie 1978) is a widely used language for operating systems, embedded systems, and low-level programming. Rust (RustLang 2015) is a systems programming language designed for performance and safety. The C2Rust transpiler allows developers to write code in C and then generate Rust code that can leverage Rust's enforcements of memory safety with an emphasis on type safety and concurrency.

- TypeScript (Microsoft 2012): A superset of JavaScript that adds optional static typing and other language features. It is transpiled into vanilla JavaScript code that can run natively on any modern browser or server environment.

Transpilation shares some similarities with traditional compilation, such as the ability to convert one language into another and the potential for performance optimizations. However, there are also some important differences and advantages to using transpilation over traditional compilation:

- Reduced Turnaround Time: Since transpilation generates code in the same high-level language as the source code, developers can immediately test and debug their code without waiting for a separate compilation step. This can lead to faster development cycles and more rapid iteration.
- Infrastructure Agnostic Development: Transpilation can help enable infrastructure agnostic development by allowing developers to write code in a high-level language that can be deployed across different environments and platforms without modification.
- More Open Software Stacks: Transpilation can help enable more open software stacks by allowing developers to use the latest language features and libraries without worrying about compatibility issues. This can lead to more innovation and faster progress in the development community.
- Reduced Overhead: Transpilation can reduce the overhead of maintaining separate code bases for different environments, as well as the need for manual code changes to support different platforms or browsers.

However, there are also some disadvantages to transpilation, such as potential performance overhead from the generated code and the need to maintain compatibility with the original language specification.

Overall, transpilation can be a powerful tool for developers to enable faster iteration, infrastructure agnostic development, and more open software stacks. Popular open-source transpilation tools can help simplify and streamline the transpilation process, making it easier for developers to adopt and use in their workflows.

1.3 Thesis Hypothesis

This work hypothesizes that synergy between popular quantum circuit abstractions and device backends can enhance capabilities to run existing quantum circuits on new platforms and, furthermore, benefits from transformations by identifying potential gate parallelism, which can results in faster execution and higher fidelity results.

1.4 Contributions

Python programs written using Qiskit are transpiled via QisDAX for the DAX abstraction, which includes parallelization of gates wherever possible to reduce circuit depth. QisDAX then remotely submits these programs to the respective quantum device or runs them in simulation. One open-source backend accessible through QisDAX is CRYO-STAQ, a DAX-based trapped-ion quantum computing system hosted at Duke University.

The contributions of this work can be summarized as follows:

- We create a software bridge between Qiskit and DAX.
- We develop an algorithm to reduce circuit depth by parallelizing gates within the capabilities of a given backend device.
- We facilitate interactions between quantum and classical processors in order to evaluate results and realize hybrid quantum-classical computing.
- We allow verification of transpiled code by adding simulator backends.
- We evaluate our software stack both with a real quantum device and a simulator backend.
- We demonstrate the capability of the pipeline to remotely execute programs written in Qiskit on an academically-hosted quantum computer.

Overall, we provide an open-source software stack, spanning from quantum languages to low-level devices, that allows remote execution of programs on an academically-hosted quantum computer.

CHAPTER

2

BACKGROUND

2.1 Qiskit

Qiskit is an open-source software development kit that simplifies the ability to compose, run, and analyze quantum circuits and programs (Qiskit contributors 2023; Wille et al. 2019; McKay et al. 2018a). Qiskit is currently the most widely used software stack for quantum cloud computing and is applied to a wide body of commercial and academic research (Griffin and Sampat 2021; Semola et al. 2022). Recent extensions have added abstractions for entire algorithms plus domain-specific APIs (e.g., optimization, finance, machine learning and chemistry) (Egger et al. 2021; developers and contributors 2023), as well as pulse-level programming (Alexander et al. 2020).

The Qiskit software can be used for simulating circuits on classical devices via Qiskit Aer, as well as interfacing to a suite of IBM Quantum devices through the Qiskit Terra library. The Terra library provides transpilers for circuit optimization and translation to suitable data structures and interfaces. QisDAX, the contribution of this work, decouples the Qiskit Terra abstraction from IBM Quantum backends and instead transpiles down to DAX.

2.2 Duke ARTIQ Extensions (DAX) Architecture

DAX builds upon the ARTIQ infrastructure developed by M-Labs and NIST (Kasprowicz et al. 2020). The program flow of ARTIQ and its dedicated FPGA hardware allows for real-time control with nanosecond precision over quantum physics experiments. However, due to ARTIQ's genericity, quantum computing stacks based on this infrastructure often develop quite monolithic and system-specific control software.

DAX is a software framework that can reduce kernel overhead and increase modularity and portability between ARTIQ experiments (Riesebos et al. 2022). DAX also provides high-level utilities and is currently used as the control system framework at Duke University, the University of Waterloo, and the University of Sydney, among others. As a result of the framework's growing popularity in academic institutions, libraries such as QisDAX will make the systems more accessible to users.

In the DAX framework, users build modular control software by grouping system functionality into modules and services. Modules are self-contained and control zero or more related devices to perform basic procedures. For example, a `trap` module may control the voltages applied to ion trap electrodes, in order to change the field shape about the ions. Modules are limited in control to the devices which they contain, i.e., they cannot share devices. Modules are added to a central registry of a system, so they can be found by services.

Services are components that control multiple modules. Any single module may be controlled by multiple services. For example, a service that loads ions into a trap may control the `trap` module to set electrodes so that they form a suitable trapping gradient, an `ablation` module that pulses a laser at an ablation target, and a `cw` module that controls the continuous-wave lasers used to ionize and cool the ablated atoms.

DAX clients further increase code reusability. Clients are generic experiments that, at runtime, combine with system-specific code for execution, allowing for high-level code transfer between systems. One such client is DAX.program, which implements an `Operation Interface` containing functions for common gate-level quantum operations with explicit timing control. The DAX architecture is summarized by Fig. 2.1.

DAX.program-sim is an addition to DAX.program allowing for classical simulation of quantum systems, with its pipeline designed to be identical to the one that runs on quantum hardware (Dalvi et al. 2022). This simulator framework is considered a canonical backend for any program written using DAX.program and provides a reliable test bench for programs converted using QisDAX.

The modular architecture of QisDAX allows it to be re-targeted to any other control system, even from another device architecture.

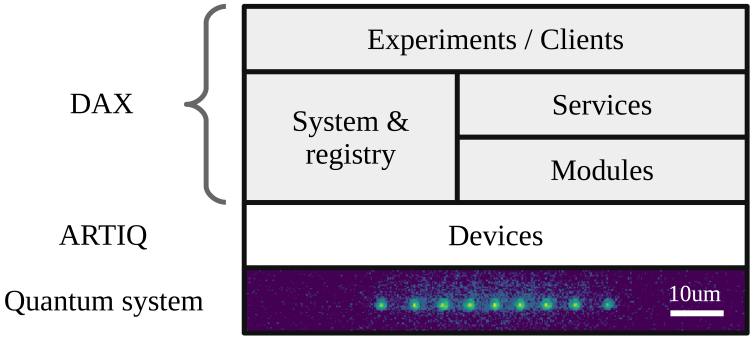

Figure 2.1: Schematic overview of how DAX architecture combines with ARTIQ to control a quantum system, in this case a trapped ion device. This figure has been taken from (Riesebos et al. 2022).

CHAPTER

3

DESIGN

The overall design objective of QisDAX is to provide users the experience of the Qiskit platform, which combines circuit abstractions with result evaluations. This means that the same data structures used by Qiskit should be made available by QisDAX, regardless of the underlying lower levels of the quantum software stack.

3.1 Software Design Challenges

Both Qiskit and DAX are designed as Python libraries, but they differ in their structure. These differences pose the following challenges:

1. A Qiskit program supports heterogeneous backends through providers. DAX, our immediate target, does not package a provider and only targets ARTIQ.

2. Qiskit represents circuits as Directed Acyclic Graphs (DAGs). As these DAGs do not translate directly to the DAX representation, we propose a novel, time-sliced approach. DAX provides various scoping constructs to more explicitly define the ordering of instructions within a circuit, indicating whether they may execute sequentially or in parallel. DAGs may be non-planar, but the scoping constructs expressed within a program are required to be planar.

3. The results of a DAX program are expressed as a vector of measurement values, many of which may be extending across multiple channels simultaneously. For the results to be usable by any workflow that analyzes results from a Qiskit program, this representation must be converted to a `Qiskit Result` object representation.

4. Resource constraints for hardware controls are not explicit for Qiskit, as circuits operate on virtual qubits. In contrast, DAX programs operate on physical qubits with explicit specification of parallel execution of gate sets using shared resources, e.g., see resources in Sect. 4.

While these constraints are specific to DAX and ARTIQ, the aim of QisDAX is to provide a software layer that can be re-targeted to lower levels of other control stacks for ion traps, or even to control stacks using different a quantum device type such as neutral atoms.

3.2 Design Solutions

QisDAX provides the following solutions to the above challenges, while considering the design objectives:

1. We provide a transpilation component from a Qiskit program to a DAX program while considering the available resource types.

2. We provision the required interfaces and objects compatible with Qiskit for heterogeneous quantum/classical processing, namely provider and job abstractions.

3. We instantiate the DAX layer with hardware-specific options compatible to the Qiskit program, where a provider can be chosen from (i) the ion trap device or (ii) a simulator instance.

4. We facilitate the conversion of job results by transforming the DAX execution results through a component to Qiskit compatible objects.

We subsequently verify the correctness of this translation by validating the generated circuits under simulation via the DAX.sim (Riesebos and Brown 2022) and the DAX.program-sim components. We further demonstrate the capability of our approach via circuit execution on trapped-ion quantum computer.

3.3 Circuit representations

Circuit representations must preserved in their semantics through the vertical layers of the software stack as part of the transpilation process, yet they should be compliant with existing Qiskit inspection and visualization capabilities.

3.3.1 Visualizing a circuit

Qiskit provides utility functions to display QuantumCircuit objects rendered as text, matplotlib, or even LaTeX. The rendered circuit consists of a (time) line for every qubit, with gates as blocks spanning the lines for the qubits they operate on and additional representations for operations such as measuring (see Fig. 3.1) and barriers. This representation has to be preserved by lower layers, where virtual qubits can be mapped to physical ones.

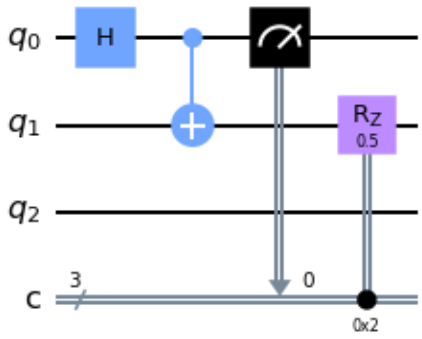

Figure 3.1: A Qiskit circuit rendered using matplotlib renderer showing a Hadamard gate, a Controlled-X gate, a single qubit measurement to a classical register of size 3, and a Z rotation gate in order from left to right.

3.3.2 Qiskit DAG

A circuit can be represented as a DAG consisting of inputs, outputs and operations as nodes and directed edges that correspond to gates and qubits (see Fig. 3.2). The depicted information may be enhanced by device-specific details such as operational parallelism, timing constraints and mappings to physical qubits when generating optimal DAX representations. On top of optimizations specified for the Qiskit transpiler, QisDAX provides only the trivial layout of mapping the i-th virtual qubit to the ith physical qubit. A cost (in terms of gates) and noise-

aware mapping could be integrated as a post-processor at a later time to implement code optimization strategies.

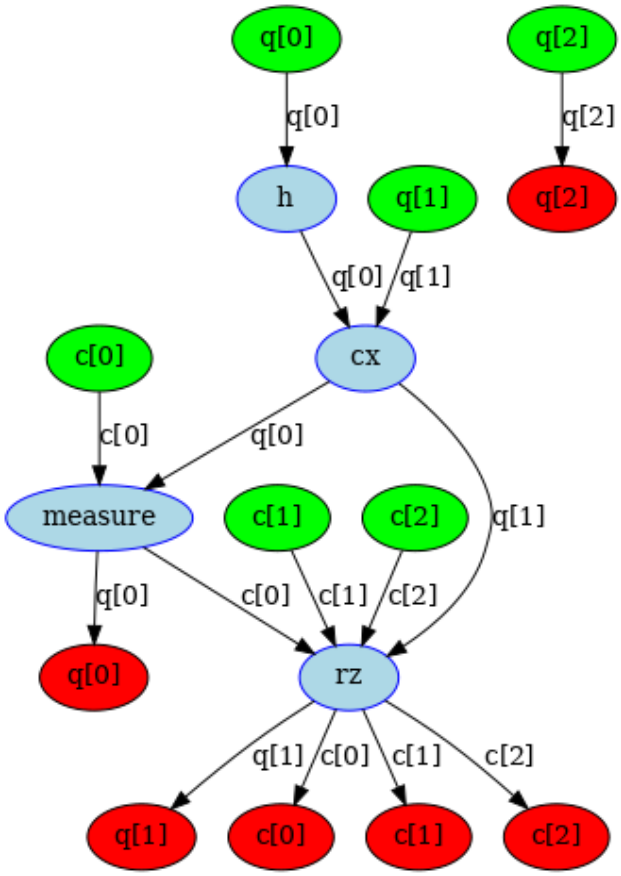

Figure 3.2: Qiskit DAG

3.3.3 DAX

DAX extends the ARTIQ circuit representation as a program, i.e., a sequence of gates using the explicit program scoping constructs `with sequential` and `with parallel` to support the specification of gate parallelism at the level of physical qubits. Each scope may contain multiple instructions or other nested scopes. Instructions and scopes at the root level of a sequential scope are guaranteed to execute in the order they appear. For a parallel scope, the instructions and scopes at the root level may execute in parallel, subject to resource availability. In other words, any subset of gates with logical concurrency in a program can be executed utilizing physical parallelism. In a noise-free environment, this adjustment would always result

in the same quantum state regardless of the amount of actual parallelism (from none to all logically concurrent gates in parallel). However, in noisy environments, the result is a trade-off in which a system may be adversely affected by an increase in parallelism while simultaneously benefiting from lower decoherence due to decreased circuit depth.

CHAPTER

4

IMPLEMENTATION

Details specific to the software packages Qiskit DAX, and, to a lesser extend, ARTIQ, influence implementation choices under the objectives of the QisDAX project.

4.1 Software Stack

QisDAX serves as a bridge between two projects, Qiskit and DAX. Qiskit serves as the input and surrounding driver program. DAX provides a number of utilities that interface with the lower-level ion-trap quantum hardware and simulators.

4.2 QisDAX Components

We ensure maximum interoperability with pre-existing Qiskit programs and minimum refactoring by providing the following utilities:

- DAXProvider: A counterpart to the Qiskit IBM [Q] provider, which ordinarily serves as a reference to access IBM Quantum backends, whereas ours refers to an ion trap device.
- DAXSimulator: An alternate backend that simulates the results of the DAX program execution transpiled down from Qiskit, by utilizing the DAX program simulator.

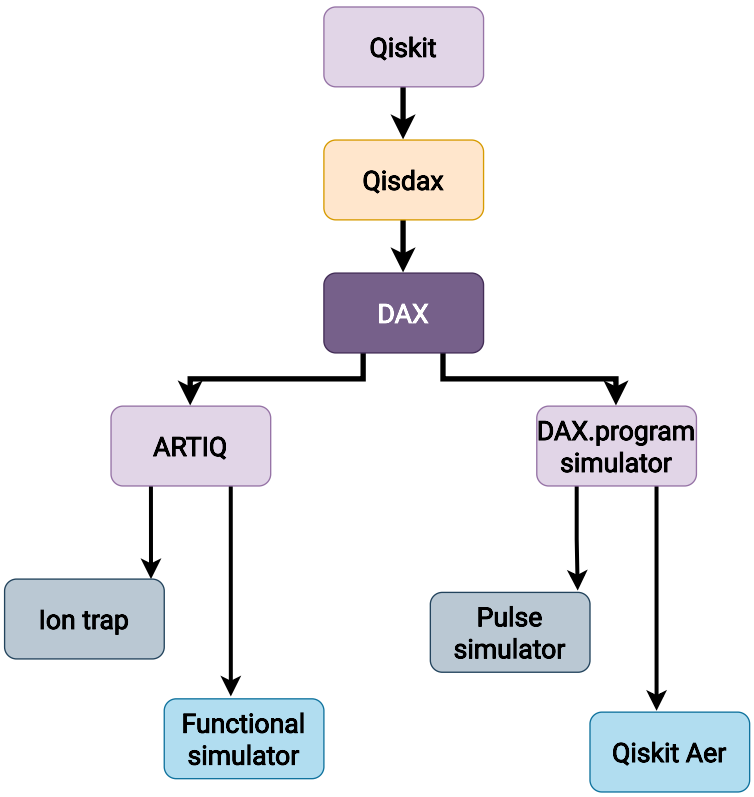

Figure 4.1: Software stack

- DAXPrinter: A backend used for generating the DAX program without executing it. All results are reported in the ground state.
- DAXArtiq: A backend for executing circuits on supported quantum hardware. Backend can be configured through a resource configuration file for the network address and destination filesystem of the device controller.
- DAXJob: The base class for QisDAX jobs specifying the execution pipeline for circuits. It is also responsible for converting results back to Qiskit-compatible objects.
- DAXSimJob: Dispatches circuits to the DAX program simulator. It is derived from DAXJob.
- DAXPrintJob: Displays DAX code to stdout. It is also derived from DAXJob.
- DAXArtiqJob: Dispatches circuits to the configured ARTIQ-compatible backend. Requires network address information for the backend.
- qobj_to_dax: Converts a Qiskit QasmQobj to the equivalent DAX program.

These utility classes are used to dispatch the circuits to the specified backends, including the circuit definition and all subsequent classical computations, which are handled as if processed by the IBM Quantum backend. As the backend is abstracted, both the input to the backend and the subsequent Qiskit result object do not need to be transformed but are fully compatible (see Fig. 4.2 for execution stages).

4.3 Resource Configuration

Resources availability can be configured through a resource specification file, resources.toml, in TOML format. The TOML specification provides an association through key-value pairs. QisDAX supports the following configuration options:

- total_lasers: Indicates the total number of lasers available for the trapped ion device. This number is assumed to account only for the lasers realizing gate operations, but not others used for cooling, measurement, etc. Note that we assume no constraints on the other laser types. This allows flexibility for operations such as measurement, assuming no upper bound for simultaneous measurement operations and future enhancements for handling of mid-circuit measurements or qubit re-initializing.
- total_mirrors: Provides the total number of mirrors available to the trapped ion computer. (Note that mirrors are specific to ion traps utilizing Micro-electromechanical systems (MEMS) technology (Wang et al. 2020)).
- relative_time: Comma-separated list of relative times for executing the n-qubit gate, where n is the 1-based index of the timing value in the list.
- lasers: The number of lasers required to perform the gate on the circuit. Used in a TOML table for a particular gate.
- mirrors: The number of mirrors required to perform the gate on the circuit. Used in a TOML table for a particular gate.

4.4 Converting to DAX

4.4.1 Restructuring Gates

We restructure the linear timeline from Qiskit to a list of layers. Each layer in turn is a list of lists for each qubit. We utilize an approach similar to a breadth-first search over the DAG, adding

parallel blocks to a sequential root block. Instructions are added to the parallel block spanning the entire circuit as individual sequential blocks for every qubit.

The pseudocode for the algorithm is shown in Algorithm 1:

The algorithm uses the following functions:

- get_qb_indices: Returns a priority ordering of the qubit indices, ordered as the qubit with the longest remaining depth first. This also takes in account the relative time comparing single qubit and 2 qubit gates.
- should_add: Returns a tuple of a boolean and integer. If the first value is True, the gate is to be inserted in the current parallel layer. Note that in our implementation, the first value is always True for the first gate for every qubit in the layer. If the gate is not the first, the gate is added if the difference in depth for the particular qubit in the layer and the maximum depth for any qubit in the layer does not increase, ensuring uniform depth distribution across qubits. Adding the first gate for any qubit is the only exception, which ensures that the layer is not empty,
- resource_cnt: Returns a dictionary of the gate resources required to execute a particular layer. Since the gate execution may be concurrent, it is the type-wise (mirrors, lasers, etc.) total of resources across all the gates in a layer.
- resource_check: Returns True if the resources required for the layer may be fulfilled by the available resources.

The algorithm works in a step-by-step manner:

i. Initialize total_gates as the count of all the gates to be scheduled and next_indices as the indices of the gates to be scheduled next.

ii. For generating each layer, keep track of the participation of each qubit (first_gate), whether the next gate has for each qubit has already been considered for the current layer (width_-checked), the max depth for any qubit in the current layer (max_width), and whether the current layer has exhausted all available resources(resource_exhausted).

iii. Prioritize qubits with longer remaining depths to add to the current layer. Add the next gate for the highest priority qubit to the layer if there are enough resources and if it is either the first gate for the qubit in the layer or if the difference between the layer depth and depth of the deepest layer does not change.

iv. Continue adding gates to the layer, keeping track of the depth for each qubit. If the maximum depth for a layer changes, reset the depth tracker for all the layers.

v. Continue adding layers to the root list until all the gates from the circuit have been included.

4.4.2 Handling Multi-qubit Gates

With the restructuring approach discussed above, we assume that each qubit has independent gate sequences. However, there may exist intersections in the form of multi-qubit gates. We mitigate this by splitting a layer into sub-layers, with a multi-qubit gate as the latter boundary of the preceding split.

4.4.3 Serializing to DAX Code

The QisDAX representation is that of a nested list of lists. At each level, we have:

i. A collection of sub-layer organizer lists. The total time for each qubit in the layer is approximately equal to the other qubits, except for the last layer. They are realized as `with_parallel` scopes. The root context is assumed to be sequential.

ii. A collection of sub-layer lists. They are realized as `with_sequential` scopes to ensure the relative order before and after a multi-qubit gate.

iii. A collection of list of gates for each qubit. They are realized as `with_parallel` scopes, as qubits in a sub layer are independent except for the final gate, which may be multi-qubit.

iv. A collection of gates for each qubit. They are wrapped in a `with_sequential` scope, executed in the order they appear in the Qiskit circuit.

The DAX program is rendered through a Jinja template. However, the loops do not exist in the template itself. Instead, they are preprocessed and injected in the template as a string,

4.5 Measurement

DAX supports simultaneous measurement of multiple qubits while maintaining all intermediate measurement results in memory. A measurement extraction from a DAX data context simply consists of a nested list of integer values, one for each qubit channel being measured. When converting to DAX, the register information for the corresponding measurement gates is stored. Each value from the DAX data context can then be mapped to its corresponding Qiskit register in the Qiskit Result object.

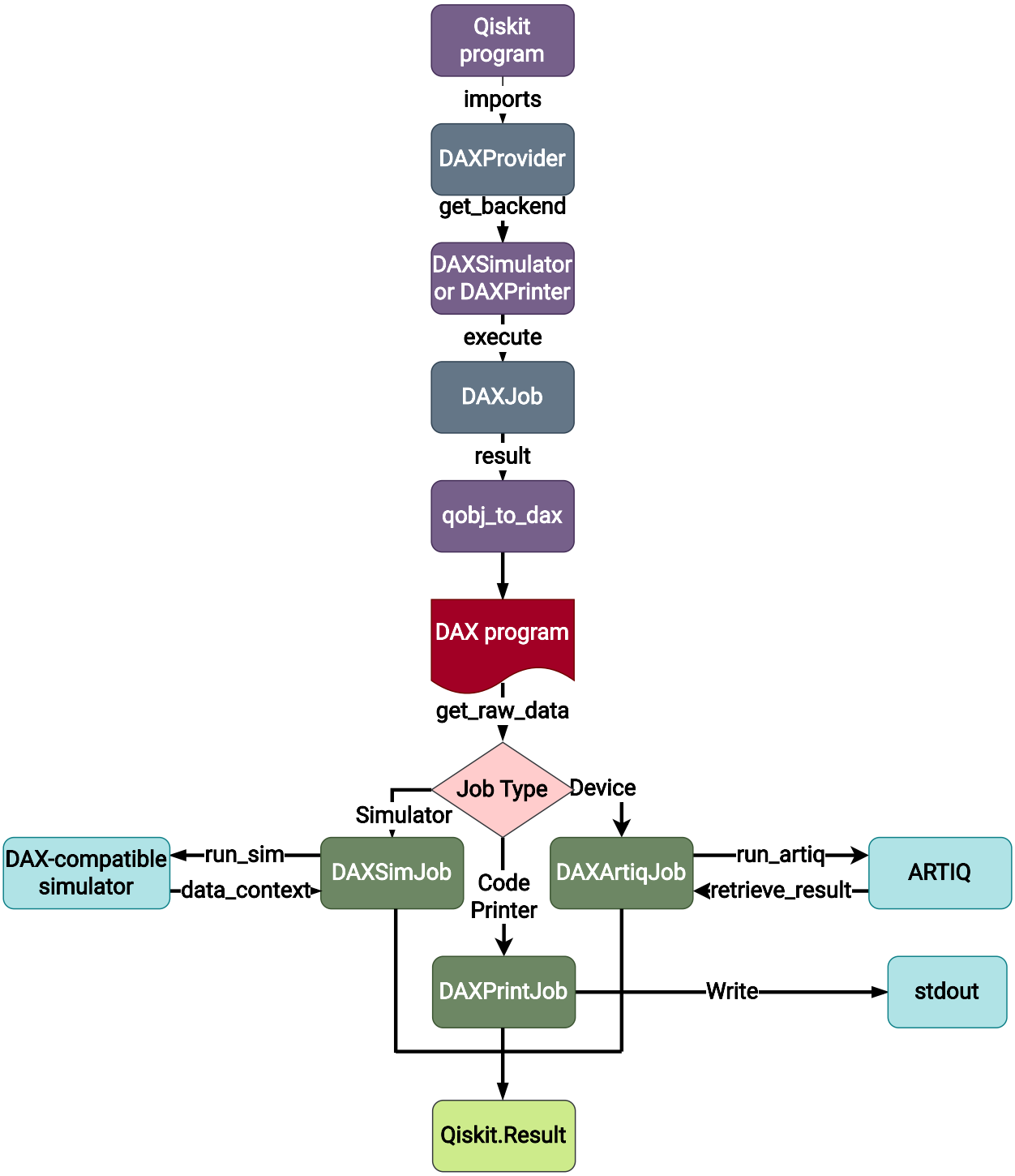

Figure 4.2: QisDAX architecture

Algorithm 1 QisDAX restructuring algorithm

```
1:  procedure GET_PARALLELIZED_LAYERS(instrs, resources)
2:      parallelized_layers ← []
3:      while instrs are unvisited do
4:          while resources available and instr queues for all
5:              qbs have not been marked unavailable do
6:              layer ← get_next_layer(instrs, resources)
7:          parallelized_layers.append(layer)
8:      return parallelized_layers
9:  procedure GET_NEXT_LAYER(instrs, resources)
10:     for qb_index =
11:             get_qb_indices(instrs, next_indices, resources) do
12:         layer ← []
13:         for qb ∈ 1 to qbs do
14:             layer.append([])
15:         sequence ← instrs[qb_index]
16:         if instr is last for qb then
17:             mark instr queue as unavailable
18:             continue
19:         instr ← sequence[next_indices[qb_index]]
20:         flag ← True                          ▷ True if the succeeding loop does not break
21:         for participant ∈ instr.qbs do
22:             if instr != participant.next then
23:                 mark instr queue as unavailable
24:                 flag ← False
25:                 break
26:         if flag then
27:             is_first_gate ← instr = layer.first for all participants
28:             should_add, new_width ←
29:                 should_add(instr, layer, is_first_gate)
30:             if should_add then
31:                 add instr to layer for all participants
32:                 resource_cnt ← resource_cnt(layer, resources)
33:                 resource_check ←
34:                     resource_check(resource_cnt, resources)
35:                 if resource_check then
36:                     for participant ∈ instr.qbs do
37:                         next_indices[participant] += 1
38:                         if new_width > max_width then
39:                             max_width ← new_width
40:                             mark participant layers as unavailable
41:                         else if new_width = max_width then
42:                             mark participant layers as unavailable
43:                 else
44:                     Remove instr from all participants
45:                     resources unavailable
46:                     break
47:             else
48:                 mark participant layers as unavailable
49:
50:     return layer
```

CHAPTER

5

EXAMPLE

5.1 Original circuit

As an example, we choose the Simon's algorithm (Simon 1997) applied to a bitstring of 110 (see Fig. 5.1 for the quantum circuit).

5.2 Circuit after Transpilation

We transpile the above circuit via QisDAX with the resource configuration as specified by Listing 1.

We then obtain a representation for the circuit as seen in Fig. 5.2. Each colored box of gates is a layer executed within a `with_parallel` scope. This representation will then be serialized to DAX.

For the first layer, we schedule the Hadamard gates to be run on the first and second qubits from the top. When executed in parallel, these utilize 4 of the 5 available lasers. We then add the Hadamard on the third qubit in the next layer. Similarly, we obtain layers for the subsequent CNOTs. Barriers always terminate the current layer.

Note that the relative ordering of the gates may change, while preserving logical order. This is a result of the priority ordering of the qubits based on the length of the remaining circuit

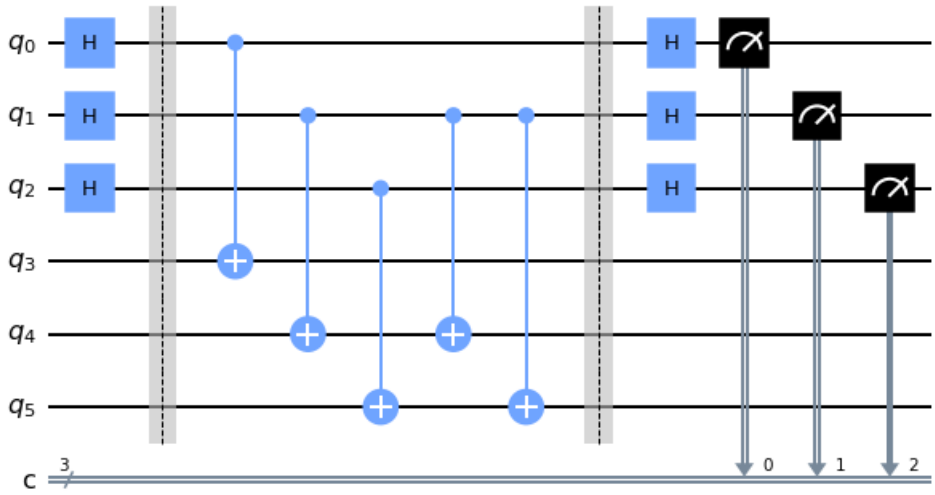

Figure 5.1: Circuit for Simon's algorithm

during processing the subsequent gates. Some of the qubits are non-engaged in some layers, even though gates are assigned to them in the immediately succeeding layer. This is intentional, and inspecting the resource files reveals that the layers where qubits are inactive may have already exhausted the available resources.

5.3 Reshaping Raw Data to a Results Object

The data context stores measurement results as a nested list of values. For our example above, it stores results for each `store_measurement` invocation and for the qubit channel(s) specified by the parameter. For x shots and y measurements, we have $x * y$ measurements returned by the data context. Each measurement result returned by the data context is independent of the specified Qiskit register associated to store the result. Hence, we additionally maintain the order of registers in which measurements are to be reported in the Qiskit context, overwriting registers as necessary.

5.4 Understanding scoping constructs

While the previous example considers a small number of available resources on a simpler circuit, we get optimal parallelization by just scheduling gates using a first-come-first-served policy on each qubit, until we exhaust resources. However, for complicated circuits running on devices with a larger number of available resources, we also consider sub-circuit parallelization.

Listing 1 Sample resource configuration for Simon's algorithm

```
total_lasers = 5
total_mirrors = 5
relative_time = '1,2,4'

[x]
lasers = 1
mirrors = 1

[h]
lasers = 2
mirrors = 2

[cx]
lasers = 2
mirrors = 2
```

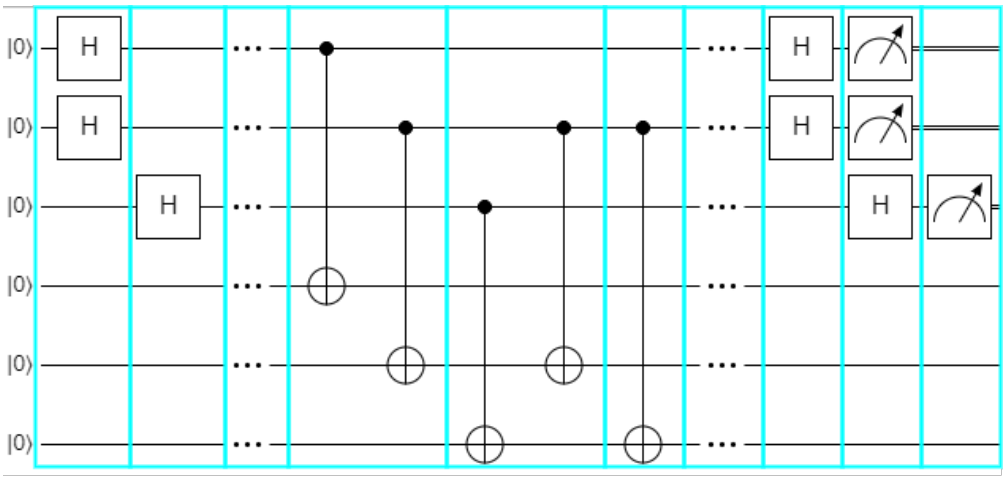

Figure 5.2: DAX representation for Simon's algorithm

Consider the resource configuration as outlined by Listing 2.

This represents a circuit with a possible scoping arrangement overlaid as colored boxes, as depicted in Fig. 5.3.

1. The red boxes denote the parallel context of a layer, in a root sequential context.

2. The blue boxes are sequential contexts to organize the contents of a layer in the parallel context of the layer itself. These contexts are defined groups of qubits that execute completely independently of other sibling contexts in the layer.

3. The green boxes are parallel contexts within the parent sequential context (blue). These are used when the execution timelines of multiple qubits concur to execute a multi-qubit gate, after which they may resume independent execution.

Listing 2 Resource configuration for demonstrating scoping structure

```
1  total_lasers = 20
2  total_mirrors = 20
3  relative_time = '1,4,10'
4
5  [x]
6  lasers = 1
7  mirrors = 1
8
9  [h]
10 lasers = 2
11 mirrors = 2
12
13 [cz]
14 lasers = 4
15 mirrors = 4
16
17 [ccx]
18 lasers = 6
19 mirrors = 6
```

4. The orange boxes are sequential contexts within the parent parallel context (green). These contain the gates in order for a single qubit.

We have a 3 qubit Toffoli on the first 3 qubits executing for 10 time units. We may schedule the Hadamard gate, the X gate, the CZ followed by another Hadamard and X gate on the 4th and 5th qubits, in parallel to the Toffoli. Further, for the 4th and 5th qubits, the X and Hadamard gates on either side of the CZ must be sequential. The maximum resource utilization in such a timeline would be when the Toffoli and both Hadamards on either side of the CZ execute concurrently.

While some boxes in Fig. 5.3 might not demonstrate all child nesting levels (see the top blue box, the horizontally middle green box, or the rightmost red box), it is deliberately drawn for ease of visualization. The implementation will only generate the innermost scope to reduce overhead. This results in the DAX.program circuit as seen in Listing 3.

In Fig 5.3, the first red box from the left corresponds to the context at line 23, while the second red box corresponds to line 41. The top and bottom blue boxes are represented by the contexts at lines 24, and 25, respectively. The green boxes from left to right are represented by the contexts at lines 26, 33 and 34. Finally, each of the orange boxes appear top to bottom in the green boxes in the image exactly as their sequential contexts appear ordered in the generated DAX program.

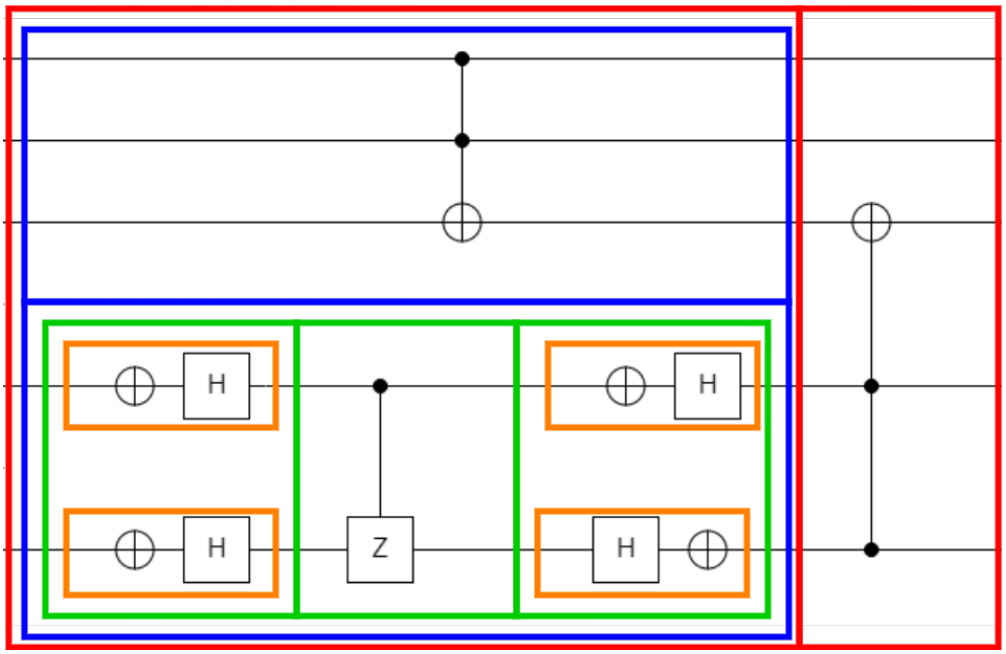

Figure 5.3: Circuit with nested scopes

Listing 3 DAX program to demonstrate scopes

```
from DAX.program import *

class QisDaxProgram(DaxProgram, Experiment):

  def build(self):
    # initialize program information

  def run(self):
    # Run the kernel
    self._run()

  @kernel
  def _run(self):
    self._qiskit_kernel()

  @kernel
  def _qiskit_kernel(self):
    with self.data_context:
      for _ in range(self._num_iterations):
        self.core.reset()
        self.q.prep_0_all()

        with parallel:
          self.q.ccx(0,1,2)
          with sequential:
            with parallel:
              with sequential:
                self.q.x(3)
                self.q.h(3)
              with sequential:
                self.q.x(4)
                self.q.h(4)
            self.q.cz(3,4)
            with parallel:
              with sequential:
                self.q.x(3)
                self.q.h(3)
              with sequential:
                self.q.h(4)
                self.q.x(4)
        self.q.ccx(3,4,2)
    # continues accordingly
```

CHAPTER

6

RESULTS

The QisDAX pipeline was demonstrated on a physical quantum computer and in simulation. The following subsections describe the results from these experiments.

6.1 Hardware Results

QisDAX was demonstrated on the CRYO-STAQ device, an experimental trapped-ion quantum computing system at Duke University (Kim et al. 2020). CRYO-STAQ is designed to be a fully connected 32 qubit system with a cryogenic vacuum chamber. All-to-all connectivity of the qubits is enabled by a multi-channel acousto-optical modulator (AOM). CRYO-STAQ uses a Kasli 2.0, from the ARTIQ hardware ecosystem, as the real-time control hardware solution. DAX is used as the control software solution.

To demonstrate the pipeline on CRYO-STAQ, we used QisDAX to remotely execute a series of single-qubit circuits written in Qiskit. Here, QisDAX was configured to use the `DAX ARTIQ device` backend, which appropriately generates the DAX.program circuit and executes it on a physical system using an ARTIQ based control system. The configuration file associated with this backend allows the user to enter the appropriate credentials required to access the remote system.

We ran a series of single-qubit circuits with increasing number of gates to benchmark the

pipeline overhead with increasing circuit depth. This pipeline overhead captures the time it takes QisDAX to convert a Qiskit circuit to a DAX.program circuit, send the circuit to be remotely run on the physical device, and finally retrieve the results back to be returned to the user. It does not include the configurable wait time for the circuit to be executed on the physical device. The results from this demonstration can be seen in Fig. 6.1.

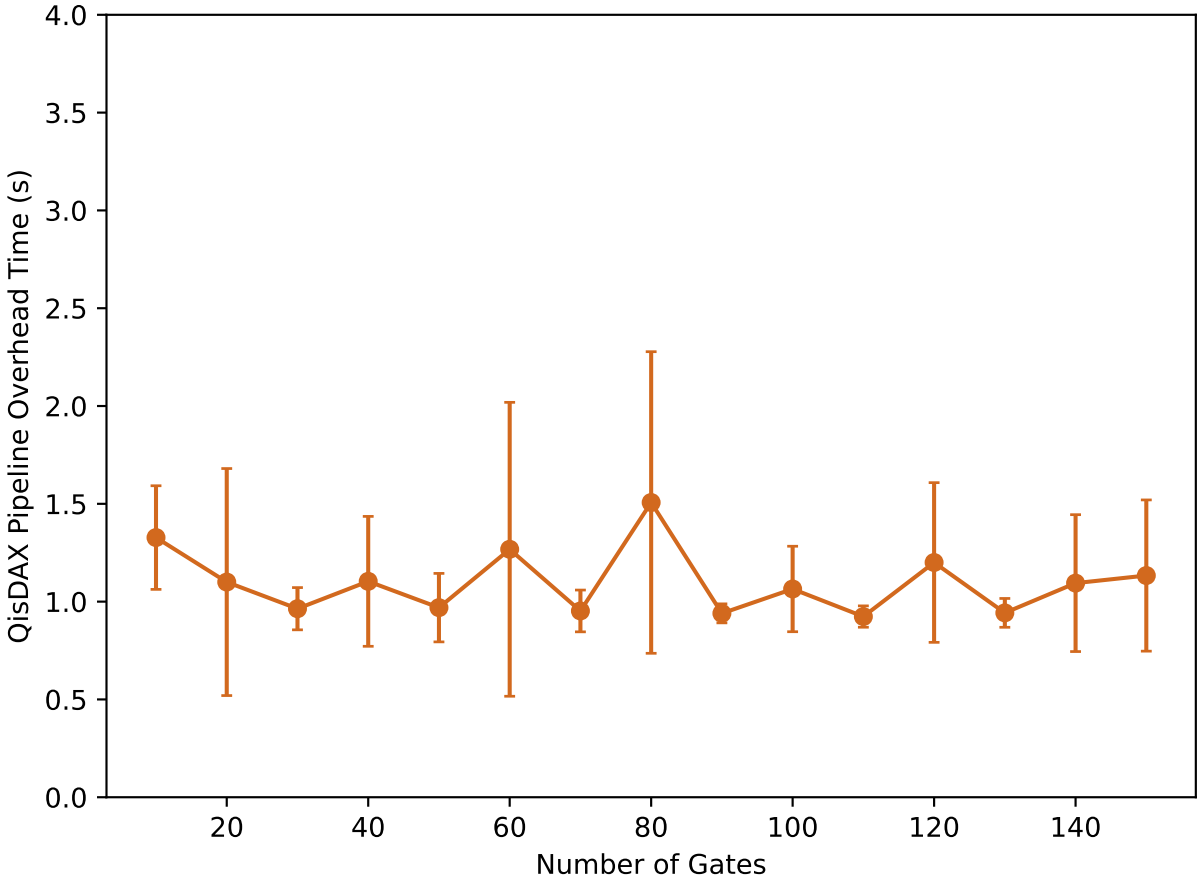

Figure 6.1: QisDAX pipeline overhead. Each of these points indicates the average time taken by QisDAX to convert a Qiskit program into a DAX.program circuit, remotely submit it to CRYO-STAQ, and return the final result. Each data point on the plot was averaged over 10 samples, and the error bars indicate the standard deviation. The data shows a near-constant overhead of ~ 1.1 seconds across increasing circuit depths.

Fig. 6.1 shows the results from this experiment. As we scale the number of gates, and consequently the circuit depth (as it is a single-qubit system), from 10 gates to 150 gates, the overhead time of the pipeline is constant at 1.1 seconds on average, with a standard deviation of

0.16 seconds. This demonstrates the favorable scaling of the pipeline overhead as the number of operations in a system increase.

The QisDAX pipeline also returns metadata about the executed job that the user may be interested in. The default metadata returned is described in Tab. 6.1.

Table 6.1: Default metadata returned by QisDAX

Metadata	Description
RID	Unique ID of the job for lookup post execution
Arguments	Describes runtime arguments like name of generated DAX file
Queue time	Time stamp when job was queued
Run start time	Time stamp of when the circuit began execution
Run end time	Time stamp of when the circuit completed execution

6.2 Simulation Results

A number of benchmark programs were tested to analyze performance. Of these, we feature a subset that includes the Deutsch-Jozsa algorithm (Deutsch and Jozsa 1992), Bernstein-Vazirani algorithm (Bernstein and Vazirani 1997), Simon's algorithm (Simon 1997), Grover's algorithm (Grover 1996) and GHZ state generation (Qiskit contributors 2023). All benchmarks are run using 3 qubits for consistency

6.2.1 Simulation using DAX.program simulator

At the time of writing, CRYO-STAQ was only capable of executing single-qubit circuits. Hence, we complemented these results by running additional benchmarks using the DAX.program-sim simulation (Riesebos and Brown 2022) architecture. This was accomplished programmatically by selecting the `DAX simulator` backend provided by the QisDAX pipeline.

The benchmark setup measures the transpilation and execution time, not including the time to convert the results back to the `Qiskit Result` object. We compare this against a simplistic transpilation with no restructuring, where all instructions are assigned to a single `with_sequential` scope, which results in a longer circuit depth.

These results are shown in Tab. 6.2, measured on a system with the configuration specified by Listing 4. The results depicted in Tab. 6.2 show average compilation of times (in milliseconds) of the benchmark circuits, each with 512 shots and repeated 16 times. The data indicate that

Table 6.2: Mean transpilation time [ms] with and without restructuring

Benchmark	No restructuring	QisDAX	Slowdown
Bernstein-Vazirani algorithm	8104	13106	61.72%
Deutsch-Jozsa algorithm	9649	13917	44.24%
GHZ state	3063	6961	127.25%
Grover's algorithm	17584	21579	22.72%
Simon's algorithm	8400	11989	42.74%
Geometric Mean			50.77%

Listing 4 System configuration for transpilation time analysis

```
CPU Model name: Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz
Operating System: Ubuntu 22.04.2 LTS
GPU: NVIDIA Corporation GK104 [GeForce GTX 660 OEM]
Memory: 7.7GiB
Swap: 2.0GiB
Disk: 2.0TB
```

QisDAX with its circuit transpilation adds an overhead in runtime cost of ≈50% (geometric mean) over the simpler approach of scheduling the instructions sequentially with a standard deviation of 0.4. This cost is independent of the available quantum hardware, and may be minimized with advanced, performant hardware.

6.2.2 Statistical simulation

Table 6.3: Runtime [μs] with and without restructuring, CRYO-STAQ configuration

Benchmark	No restructuring	QisDAX	Speedup
Bernstein-Vazirani algorithm	400	345	13.75%
Deutsch-Jozsa algorithm	585	505	13.68%
GHZ state	360	340	5.56%
Grover's algorithm	1465	1315	10.24%
Simon's algorithm	895	520	41.90%
Geometric Mean			13.5%

We analyze the predicted execution time by considering the resource configuration outlined

Listing 5 Resource configuration for runtime savings analysis

```
total_lasers = 5
total_mirrors = 5
relative_time = '5,150'

[id]
lasers = 1
mirrors = 1

[x]
lasers = 1
mirrors = 1

[z]
lasers = 1
mirrors = 1

[h]
lasers = 2
mirrors = 2

[cx]
lasers = 2
mirrors = 2

[cz]
lasers = 2
mirrors = 2
```

in Listing 5. The configuration option `relative_time` defines estimated execution times for single qubit gate operations as 5 units and two qubit gate operations to be 150 units. The estimated gate times here are based on measurements from the CRYO-STAQ machine, with units as μs. We consider the runtime for a benchmark as the runtime on the simple path that takes the longest time. Here, runtime includes only circuit execution time, but neither device initialization nor measurement costs. These runtime estimates in Tab. 6.3 are for a single shot. QisDAX results in a speedup of 13.5% (geometric mean) over a purely sequential approach.

We also compute benchmark times for a commercially available quantum system, the IonQ Aria (IonQ 2022). For comparability, we keep the available resources the same, only changing the `relative_time` configuration option. Listing 6 highlights the changes made to the CRYO_STAQ configuration in Listing 5 to obtain the configuration for IonQ Aria.

The results in Tab. 6.4 demonstrate a higher relative savings. With single-qubit gates making a higher contribution to the circuit time, costs due to unparallelized two-qubit gates may be offset by parallelizing a larger set of single-qubit gates.

Savings are a function of a number of factors, including gate time, order of gate operations

Listing 6 Changes to Listing 5 to obtain IonQ Aria configuration

```
- relative_time = '5,150'
+ relative_time = '135,600'
```

Table 6.4: Runtime [μs] with and without restructuring, IonQ Aria configuration

Benchmark	No restructuring	QisDAX	Speedup
Bernstein-Vazirani algorithm	3900	2415	38.08%
Deutsch-Jozsa algorithm	5715	3150	44.88%
GHZ state	2820	2280	19.15%
Grover's algorithm	11955	7905	33.88%
Simon's algorithm	6915	3690	46.64%
Geometric Mean			34.89%

in the initial circuit and subcircuit optimizations. The current approach is greedy and does not consider alternative configurations for commutative operations, which may lead to different circuits.

Table 6.5: Circuit depth [gate count on the longest path] with and without restructuring

Benchmark	No restructuring	QisDAX	Savings
Bernstein-Vazirani algorithm	24	13	45.83%
Deutsch-Jozsa algorithm	28	17	39.29%
GHZ state	15	11	26.67%
Grover's algorithm	62	36	41.94%
Simon's algorithm	36	19	47.22%
Geometric Mean			39.41%

Tab. 6.5 shows QisDAX results in an average depth savings of ≈36% over the simple approach with a standard deviation of 0.08. This depth reduction is critical in quantum computing due to decoherence times. Compared to a naïve schedule of gates for a base circuit without restructuring, QisDAX creates 39% shallower circuits under the same fidelity as the base circuit, potentially increasing their fidelity due to lower decoherence.

The absolute savings in gate depth depend on T1/T2 decoherence and gate switch times, which differ significantly depending on device technologies. For example, on an IonQ Aria (21 qubits) IonQ (2022), median T1/T2 times are 1 seconds while 2-qubit gates take 600ns, which

means that the gate depth is 1,666 before reaching a 50% decay from the original state on the decoherence curve. For IBM Kolkata (Falkon r5.11 architecture) IBM (2023), median T1 and T2 are 100us and 50us, respectively, with an average 2-qubit gate time of 0.4us, resulting in a gate depth of 125 for a 50% decay. Notice that worst-case results due to gate and qubit fidelity variations typically reduce the depth to about 30% of these numbers (at least for IBM Kolkata, worst case was not reported for IonQ). Also, to ensure a 0.75 probability for the outcome along the exponential decay curves of T1/T2, i.e., depth may have to be constrained even further. On the plus side, circuits tend to have a mix of 1 and 2 qubit gates, but these estimates are based on 2-qubit gate times since IBM did not report single qubit times. Overall, a depth increase of 39% by QisDAX can make a significant difference in the age of noisy intermediate scale quantum devices.

Table 6.6: Pipeline runtime [ms] with and without restructuring, 10^6 shots, CRYO_STAQ configuration

Benchmark	No restructuring	QisDAX	Speedup
Bernstein-Vazirani algorithm	408104	358106	12.25%
Deutsch-Jozsa algorithm	594649	518917	12.74%
GHZ state	363063	346961	4.44%
Grover's algorithm	1482584	1336579	9.85%
Simon's algorithm	903399	531989	41.11%
Geometric Mean			12.29%

Table 6.7: Pipeline runtime [ms] with and without restructuring, 10^6 shots, IonQ Aria configuration

Benchmark	No restructuring	QisDAX	Speedup
Bernstein-Vazirani algorithm	3908104	2428106	37.87%
Deutsch-Jozsa algorithm	5724649	3163917	44.73%
GHZ state	2823063	2286961	18.99%
Grover's algorithm	11972584	7926579	33.79%
Simon's algorithm	6923399	3701989	46.53%
Geometric Mean			34.74%

Using Tab. 6.2 and Tab. 6.3, the overall runtime for a pipeline leveraging QisDAX is determined, including time for transpilation followed by circuit execution for a million shots. The results in Tab. 6.6 and Tab. 6.7 indicate an overall speedup of ≈12% and ≈35% (geometric mean) with the CRYO_STAQ and IonQ Aria configurations, respectively. With a reduction in circuit depth over a purely sequential approach by ≈39% (geometric mean) with a standard deviation of 0.08, QisDAX provides considerable speedup with efficient utilization. QisDAX facilitates a trade-off between an increase in circuit execution time and decoherence in noisy quantum devices and a one-time transpilation cost. As the transpilation time is constant, the speedup improves proportional to number of shots. For repeated experiments, we may pre-transpile the circuits ahead-of-time.

These benefits also scale with the available resources in a device capable of parallelism. While our results assume a small number of lasers and mirrors for our benchmarks, increasing the available lasers and mirrors would further decrease overall circuit depth by allowing more qubits to be active per layer.

CHAPTER

7

RELATED WORK

As quantum computing workloads expand towards practical applications, we observe the emergence of multiple competing standards for implementation, both at a high abstraction level (Developers 2022; Microsoft 2020; Qiskit contributors 2023) and for low level controls (Kasprowicz et al. 2020). The contexts for higher level quantum programming tools are similar, and efforts to enable interoperability have been forthcoming.

Quantastica (Quantastica 2019), the closest related work, generates higher-level programs adhering to different quantum APIs from simpler circuit descriptions by providing proper API contexts in a template-like manner, similar to QisDAX's translation from QisKit to DAX. However, QisDAX embeds critical circuit analysis within transpilation process to delimit serial and parallel scopes critical for the vertical stack, and enables platform-aware optimizations with ability to execute on available quantum hardware.

Academic efforts towards open quantum computing have leveraged the availability of an open hardware ecosystem (Kasprowicz et al. 2020). Attempts at creating an open, platform agnostic reference standard for quantum information have also been made (Cross et al. 2022). Similar attempts for control hardware exist (McKay et al. 2018b), with cross platform demonstrations (Services 2022). Orchestrating heterogeneous systems as independent components in an application pipeline has introduced a need for platform-agnostic quantum-classical coupling (Mintz et al. 2019) and verification systems (Adams et al. 2021).

Yet, these approaches still lack an open-source transpilation tool. QisDAX provides such capability by transpiling Qiskit programs into DAX code. Together with the underlying ARTIQ low-level controls, QisDAX provides the missing link that allows the wealth of quantum programs available in Qiskit to be automatically translated for non-IBM devices, as is demonstrated for the ion-trap CRYO-STAQ device at Duke University.

CHAPTER

8

CONCLUSION

Interoperability between quantum computing stacks can be facilitated by adopting and integrating modular, open-source components. In this work, we have presented QisDAX, a bridge between two open-source quantum computing frameworks, Qiskit and DAX. QisDAX represents the first open-source, end-to-end, full-stack pipeline for remote submission of quantum programs for trapped ions in an academic setting. Its modular architecture also allows QisDAX to be re-targeted to any other control system, so that in the future it can support a variety of backend implementations, not limited to trapped-ion systems. QisDAX transpilation parallelizes gates wherever possible, maintaining circuit fidelity and result artifacts without developer overhead.

We demonstrate this transpilation procedure using backend implementations for simulators and trapped-ion devices, both local and remote. In doing so, we establish operational capabilities with algorithms from the Qiskit library, which includes parametrized quantum procedures as well as classical result analysis. The modular architecture of QisDAX also allows us to leverage advantages available only to the target system; i.e., we achieved parallel semantics that are trapped-ion specific and not readily available via Qiskit.

Single-qubit timing data from a trapped-ion device shows that the pipeline runtime overhead scales well with increasing circuit depth. A number of benchmark algorithms, run on a functional simulator, allow us to analyze the impact of the parallelization process by measuring

the mean transpilation time and decrease in overall circuit depth. We use these results to benchmark runtimes for two trapped-ion systems, CRYO-STAQ and IonQ Aria. Though we incur a one-time transpilation cost, we calculate speedups of 12% and 34% for CRYO-STAQ and IonQ Aria, respectively, in the overall pipeline runtime due to shorter circuit depth, reducing the impact of decoherence and improving efficiency and throughput.

REFERENCES

Adams, A., Pinto, E., Young, J., Herold, C., McCaskey, A., Dumitrescu, E., and Conte, T. M. (2021). Enabling a programming environment for an experimental ion trap quantum testbed.

Alexander, T., Kanazawa, N., Egger, D. J., Capelluto, L., Wood, C. J., Javadi-Abhari, A., and McKay, D. C. (2020). Qiskit pulse: programming quantum computers through the cloud with pulses. *Quantum Science and Technology*, 5(4):044006.

ANIS, M. S., Abraham, H., AduOffei, et al. (2021). Qiskit: An open-source framework for quantum computing.

Babel (2014). Babel/babel: Babel is a compiler for writing next generation javascript.

Bernstein, E. and Vazirani, U. (1997). Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473.

Bourdeauducq, S., Jördens, R., Zotov, P., Britton, J., Slichter, D., Leibrandt, D., Allcock, D., Hankin, A., Kermarrec, F., Sionneau, Y., Srinivas, R., Tan, T. R., and Bohnet, J. (2016). Artiq 1.0.

Chong, Frederic T., D. F. and Martonosi, M. (2017). Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187.

Cross, A., Javadi-Abhari, A., Alexander, T., Beaudrap, N. D., Bishop, L. S., Heidel, S., Ryan, C. A., Sivarajah, P., Smolin, J., Gambetta, J. M., and Johnson, B. R. (2022). OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50.

Dalvi, A. S., Mazurek, F., Riesebos, L., Whitlow, J., Majumder, S., and Brown, K. R. (2022). Modular architecture for classical simulation of quantum circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 810–812.

Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558.

Developers, C. (2022). Cirq. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

developers, T. Q. N. and contributors (2023). Qiskit nature 0.6.0. Qiskit Nature has some code that is included under other licensing. These files have been removed from the zip repository provided here and are only available via Github. See https://github.com/Qiskit/qiskit-nature#license for more details.

Ecma International (2022). ECMA-262 language specification. Retrieved from `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`.

Egger, D. J., Hincks, I., Landa, H., Malekakhlagh, M., Parr, A., Puzzuoli, D., Rosand, B., Rupesh, R. K., Treinish, M., Ueda, K., and Wood, C. J. (2021). Qiskit dynamics.

Griffin, P. and Sampat, R. (2021). Quantum computing for supply chain finance. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 456–459.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 212–219, New York, New York, USA. ACM.

IBM (2023). Ibm quantum dashboard. `https://quantum-computing.ibm.com/services/resources?tab=systems`.

Immunant and Galois (2018). Immunant/c2rust: Migrate c code to rust.

IonQ (2022). Ionq aria: Practical performance. `https://ionq.com/resources/ionq-aria-practical-performance`.

Kasprowicz, G., Kulik, P., Gaska, M., Przywozki, T., Pozniak, K., Jarosinski, J., Britton, J. W., Harty, T., Balance, C., Zhang, W., et al. (2020). Artiq and sinara: Open software and hardware stacks for quantum physics. In *Quantum 2.0*, pages QTu8B–14. Optica Publishing Group.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall, Inc., USA.

Kim, J., Chen, T., Whitlow, J., Phiri, S., Bondurant, B., Kuzyk, M., Crain, S., Brown, K., and Kim, J. (2020). Hardware design of a trapped-ion quantum computer for software-tailored architecture for quantum co-design (staq) project. In *Quantum 2.0*, pages QM6A–2. Optical Society of America.

McKay, D. C., Alexander, T., Bello, L., Biercuk, M. J., Bishop, L., Chen, J., Chow, J. M., Córcoles, A. D., Egger, D., Filipp, S., Gomez, J., Hush, M., Javadi-Abhari, A., Moreda, D., Nation, P., Paulovicks, B., Winston, E., Wood, C. J., Wootton, J., and Gambetta, J. M. (2018a). Qiskit backend specifications for openqasm and openpulse experiments.

McKay, D. C., Alexander, T., Bello, L., Biercuk, M. J., Bishop, L., Chen, J., Chow, J. M., Córcoles, A. D., Egger, D., Filipp, S., Gomez, J., Hush, M., Javadi-Abhari, A., Moreda, D., Nation, P., Paulovicks, B., Winston, E., Wood, C. J., Wootton, J., and Gambetta, J. M. (2018b). Qiskit backend specifications for OpenQASM and OpenPulse experiments. *preprint arXiv:1809.03452*.

Microsoft (2012). Microsoft/typescript: Typescript is a superset of javascript that compiles to clean javascript output.

Microsoft (2020). *Q# Language Specification*. Microsoft.

Mintz, T. M., Mccaskey, A. J., Dumitrescu, E. F., Moore, S. V., Powers, S., and Lougovski, P. (2019). Qcor: A language extension specification for the heterogeneous quantum-classical model of computation.

Murali, P., Linke, N. M., Martonosi, M., Abhari, A. J., Nguyen, N. H., and Alderete, C. H. (2019). Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 527–540.

Qiskit contributors (2023). Qiskit: An open-source framework for quantum computing.

Quantastica (2019). Quantastica/qconvert-js: Quantum programming language converter.

Riesebos, L., Bondurant, B., Whitlow, J., Kim, J., Kuzyk, M., Chen, T., Phiri, S., Wang, Y., Fang, C., Horn, A. V., Kim, J., and Brown, K. R. (2022). Modular software for real-time quantum control systems. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 545–555.

Riesebos, L. and Brown, K. R. (2022). Functional simulation of real-time quantum control software. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 535–544.

Riesebos, L., Fu, X., Moueddenne, A. A., Lao, L., Varsamopoulos, S., Ashraf, I., Van Someren, J., Khammassi, N., Almudever, C. G., and Bertels, K. (2019). Quantum accelerated computer architectures. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.

Rust-Lang (2015). Rust-lang/rust: Empowering everyone to build reliable and efficient software.

Semola, R., Moro, L., Bacciu, D., and Prati, E. (2022). Deep reinforcement learning quantum control on ibmq platforms and qiskit pulse. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 759–762.

Services, A. W. (2022). Amazon braket python sdk.

Simon, D. R. (1997). On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483.

Wang, Y., Crain, S., Fang, C., Zhang, B., Huang, S., Liang, Q., Leung, P. H., Brown, K. R., and Kim, J. (2020). High-fidelity two-qubit gates using a microelectromechanical-system-based beam steering system for individual qubit addressing. *Physical Review Letters*, 125(15):150505.

Wille, R., Van Meter, R., and Naveh, Y. (2019). Ibm's qiskit tool chain: Working with and developing for real quantum computers. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1234–1240. IEEE.