

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-2020

Quantum Transpiler Optimization: On the Development, Implementation, and Use of a Quantum Research Testbed

Brandon K. Kamaka

Follow this and additional works at: <https://scholar.afit.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Kamaka, Brandon K., "Quantum Transpiler Optimization: On the Development, Implementation, and Use of a Quantum Research Testbed" (2020). *Theses and Dissertations*. 3590.

<https://scholar.afit.edu/etd/3590>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.



**Quantum Transpiler Optimization:
On the Development, Implementation, and Use
of a Quantum Research Testbed**

THESIS

Brandon K Kamaka
AFIT-ENG-MS-20-M-029

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-20-M-029

QUANTUM TRANSPILER OPTIMIZATION:
ON THE DEVELOPMENT, IMPLEMENTATION, AND USE OF A QUANTUM
RESEARCH TESTBED

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Cyber Operations

Brandon K Kamaka, B.Sc.

March 26, 2020

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-20-M-029

QUANTUM TRANSPILER OPTIMIZATION:
ON THE DEVELOPMENT, IMPLEMENTATION, AND USE OF A QUANTUM
RESEARCH TESTBED
THESIS

Brandon K Kamaka, B.Sc.

Committee Membership:

Laurence D Merkle, Ph.D
Chair

David E Weeks, Ph.D
Member

Lt Col Patrick J Sweeney, Ph.D
Member

Abstract

Quantum computing research is at the cusp of a paradigm shift. As the complexity of quantum systems increases, so does the complexity of research procedures for creating and testing layers of the quantum software stack. However, the tools used to perform these tasks have not experienced the increase in capability required to effectively handle the development burdens involved. This case is made particularly clear in the context of IBM QX Transpiler optimization algorithms and functions. IBM QX systems use the Qiskit library to create, transform, and execute quantum circuits. As coherence times and hardware qubit counts increase and qubit topologies become more complex, so does orchestration of qubit mapping and qubit state movement across these topologies. The transpiler framework used to create and test improved algorithms has not kept pace. A testbed is proposed to provide abstractions to create and test transpiler routines. The development process is analyzed and implemented, from design principles through requirements analysis and verification testing. Additionally, limitations of existing transpiler algorithms are identified and initial results are provided that suggest more effective algorithms for qubit mapping and state movement.

Acknowledgements

I would first like to offer my everlasting gratitude to my advisor, Dr. Laurence D Merkle for his time, guidance, knowledge and most of all for his faith in my ability to succeed. His unflappable calm in the face of chaos was equal parts frustrating, comforting, and empowering. I'd also like to thank my committee, Dr. David Weeks and Lt Col Patrick Sweeney, for their time and expertise. Whatever success may come from my time and work at AFIT, they will forever be a part of it.

I'd also like to thank my family. I could not have understood before I began this journey the time required to get here, and without my family's boundless support and accommodation I could never have succeeded.

Finally, I'd like to thank my fellow students and researchers for their support, assistance, and advice. Lt Marvin Newlin, Lt Chris Dukarm, Ms. Jessica Switzler, and Capt Leleia Hsia were instrumental to this achievement. Time with them did not always raise my grades, but always raised my spirits and especially when I most needed it.

Brandon K Kamaka

Contents

	Page
Abstract	iv
Acknowledgements	v
List of Figures	viii
List of Tables	ix
I. Introduction	1
1.1 Motivation	1
1.2 Problem Background	2
1.3 Research Objectives	5
1.4 Limitations	6
1.5 Document Overview	8
II. Background and Literature Review	9
2.1 Overview	9
2.2 Quantum Computation Model	9
2.2.1 Hilbert Spaces	10
2.2.2 Qubits	12
2.2.3 Unitary operators	13
2.2.4 Measurement Operators	14
2.2.5 Quantum Algorithms	16
2.2.6 Quantum Annealing	16
2.2.7 Quantum Gate Model	17
2.3 Qiskit and IBM QX Architecture	18
2.3.1 Terra	19
2.3.2 Aer	22
2.3.3 IBM Quantum Hardware	25
2.4 IBM QX Transpilation	29
2.5 Quantum Layout Problem	30
2.5.1 Optimization Techniques (Previous Work)	32
2.6 Summary	38
III. Methodology	39
3.1 Overview	39
3.2 Approach	39
3.3 Test Bed Design Principles and Goals	39
3.3.1 Design Principles	40
3.3.2 Requirements Analysis	43

	Page
3.4 Algorithm Overview	45
3.4.1 Initialization and Pre-processing	47
3.4.2 Connectivity Component	49
3.4.3 Distance Component	50
3.5 Benchmarks and Evaluation	54
3.5.1 Evaluation Circuits	55
3.5.2 Evaluation Transpiler Configurations	57
3.6 Summary	62
IV. Results and Analysis	63
4.1 Overview	63
4.2 Quantum Layout Problem Testbed (QLP-TB) Design Implementation	63
4.2.1 Design Principle Analysis and Results	64
4.2.2 QLP-TB Requirements Results	67
4.2.3 Verification Results	73
4.2.4 QLP-TB Results Summary	76
4.3 Quantum Layout Solver (QLS) Heuristic Results	82
4.4 Summary	83
V. Conclusions	85
5.1 Overview	85
5.2 Contribution	85
5.3 Future work on the QLP-TB	86
5.4 Future Work on the Quantum Layout Problem (QLP)	87
5.4.1 Global Improvements	88
5.4.2 Connectivity Improvements	90
5.4.3 Distance Improvements	90
5.5 Concluding Remarks	92
Appendix A. Quantum Circuits Provided by the QLP-TB	94
Appendix B. Experiment Script for the QLP-TB	98
Appendix C. Code Snippets	103
Bibliography	117
Acronyms	125

List of Figures

Figure		Page
1	A schematic view of the circuit model constructed by Qiskit Terra.....	20
2	Simple quantum program in Qiskit and its associated circuit.....	23
3	Example topologies available on IBM QX QCs.....	26
4	An example simple circuit and an associated, fully satisfactory layout	30
5	An example complex circuit and an associated, partially satisfactory layout	31
6	Siraichi SWAP minimization algorithm	35
7	A circuit with its locally-optimal and globally preferred layouts	46
8	Grover’s Algorithm test circuit	57
9	Full pass sets for the IBM pre-populated PassManager configurations	61
10	Increase in SWAP count after transpilation.....	77
11	Increase in SWAP count after transpilation, relative to the starting size of the circuit	78
12	Euclidean distance between the results distributions from real and ideal executions	79
13	Jensen-Shannon distance between the measurement distributions of noiseless circuit simulation and actual execution on quantum hardware	80
14	Transpilation time (ms) of each configuration and circuit.....	81

List of Tables

Table		Page
1	Summary of commonly used gates and their matrix representations	28
2	Summary of test set circuits for evaluating QLS performance	58
3	Summary of test set circuits available in the QLP-TB	97

QUANTUM TRANSPILER OPTIMIZATION:
ON THE DEVELOPMENT, IMPLEMENTATION, AND USE OF A QUANTUM
RESEARCH TESTBED

I. Introduction

1.1 Motivation

Quantum computing is a rapidly growing field with imminent potential to dramatically increase the tractability of numerous problems of concern to the Department of Defense, including resource allocation, cryptanalysis and cryptography, and advanced materials engineering.

In pursuit of such capabilities, successful implementation of advanced compiler optimization techniques for generic quantum algorithms is a key milestone in achieving quantum supremacy on Noisy Intermediate-Scale Quantum (NISQ) systems. As NISQ systems are likely to be the dominant paradigm of quantum computers for the near and mid-term in quantum computation, the Air Force mission to maintain information dominance rests greatly on such quantum capabilities.

However, previous work has been highly varied in both techniques used and in choices of parameters to optimize in compilation schemes. Although algorithms and tools exist to improve qubit lifetime heuristics, SWAP minimization heuristics, and gate optimization procedures, there is little integration research designed to enable rapid prototyping of algorithms and optimization methodologies, nor are there effective or efficient benchmarking tools to generate or analyze the results of applying such optimizations.

1.2 Problem Background

Quantum computing shows significant promise for enhancing the future capabilities of the military and civilian organizations that have the expertise, engineering, and resources to construct and use them. However, quantum computing is in its infancy, and most notably no known quantum service provider has yet solved the physical and engineering challenges necessary to enable the level of abstraction end-users of classical computers systems are accustomed to. In particular, traditional computer technology is sufficiently matured such that developers and users generally are not required to understand the physical workings and constraints of an executing machine, instead multiple layers of interfaces permit developers to perform operations like data storage and retrieval, networking, operation scheduling, or multi-processing without engaging directly with the hardware that enables such operations; in short, the average developer does not need to know, for example, the voltage or timing of the transistors they are making use of.

Quantum computers, by comparison, require significant investment by developers and users in understanding and engaging directly with the constraints of the underlying hardware. There are multiple, substantial consequences of the current paradigm: first is that this limits the pool of available users and developers by restricting the operation of quantum computers to multi-disciplinarians comfortable with more than the usual programming principles; second, there is a significant workload associated with each new task executed on quantum hardware—including experimental tasks to resolve the existing abstraction issue—and users must develop solutions, often replicating the work of others, to solve the constraints presented and perform basic tasks like operation scheduling. The goals of providing comprehensive layers of abstraction are to identify and consistently use optimal solutions to simple, recurring, low-level tasks, as well as to enable end-users to expend their time and expertise

solving higher-order problems in a more portable manner.

In classical computer systems, compilers and operating systems transparently manage memory access. Developers identify symbols that map to virtual memory locations and data to associate with them [35], but these virtual memory locations exist only within the scope of the program being executed; the compiler and operating system map the virtual memory locations to physical memory, and also ensure that the same physical memory is not unintentionally mapped to multiple symbols. This allows developers to allocate and use memory without concern for the underlying memory architecture. This abstraction also permits arbitrary memory access, regardless of location; that is, two integers can be allocated, stored, and later, e.g., summed without reference to the physical location of the data in the memory chips.

Quantum compilers currently lack this abstraction layer. All memory—physical qubits storing some quantum state—must be allocated manually, bit-by-bit, and retrieved by explicit reference to the physical qubit’s address in the architecture [25]. If a quantum developer wishes to have data from two qubits interact, then an entanglement operation must be performed on both qubits simultaneously. The constraints of all existing transmon architectures, including that used by IBM, are such that entanglement operations can only be performed on qubits that are *physically* adjacent and share a single, microwave-pulse wave-guide. One goal of quantum compilers, or in IBM’s nomenclature the quantum transpiler, is to orchestrate remapping of symbols and their associated data present in a quantum program to different physical qubits, such that when program execution requires an entanglement, the qubits being entangled are physically co-located as required.

Additional characteristics of the IBM quantum architecture also affect the remapping process. Although all operations on classical computers have some non-zero probability of failure, such probabilities are individually extremely low [52] and the

large number of bits available for program execution makes error correction procedures easy to implement and effective. Conversely, quantum computers have relatively high error rates associated with data storage, manipulation, movement, and retrieval, varying between approximately 0.5 and 7% [5] depending on the specific qubit and operation. Additionally, the *no-cloning theorem* of quantum mechanics identifies a critical constraint on all quantum systems without exception: quantum state cannot be copied between quantum objects [51].

These characteristics mean that mapping and remapping operations on qubits are fraught with difficulties. First, is that the initial mapping of virtual to physical qubits should be done in a way that maximizes the reliability of operations performed; all else being equal, it is better to select qubits with superior associated reliabilities. Second, since qubits cannot be cloned, then Virtual-to-Physical (V2P) mappings should take into account which qubits need to be near one another throughout the execution of the quantum program. The relatively high failure rate of qubit operations compared to bit-wise operations means that probabilities of failure when moving a virtual qubit between physical locations are a dominant concern.

Taken together, these goals, constraints, and methodologies for accommodating them are known as the *circuit mapping problem* [32], the *Quantum Layout Problem (QLP)* [37], or the *qubit allocation problem* [47], and will be referred to as the QLP throughout this paper.

Compounding the difficulties associated with the lack of abstraction layers on existing IBM QX architecture is the fact that testing and experimenting on proposed policies, routines, or algorithms intended to provide such abstractions is itself a manual procedure requiring explicit engagement with complicated, low-level code constructs. Further, development in this environment is plagued by the necessity of making a multitude of small, arbitrary decisions. Although such decisions are a nat-

ural component of many development tasks, they are rarely commented on, justified, or given visibility. This leads to a multitude of researchers using slightly different techniques, algorithms, or data structures to accomplish fundamentally similar tasks, which both hampers collaboration and makes replication or extension of existing results a frustrating and difficult task. This lack of wrappers, utility functions, and structured access to system internals is an issue whose resolution necessarily precedes future work on solving the QLP and similar issues.

1.3 Research Objectives

Little research has been done to optimize the compiler operations that orchestrate the V2P mapping and movement of qubit state among hardware qubits. This research is intended to facilitate comprehensive testing of qubit allocation and mapping algorithms and to introduce and analyze new methods of determining optimal qubit orchestration. In particular, there are two primary questions and an additional question to guide future research in consideration of the primary research goals:

1. What are the design principles and requirements of an effective testbed for proposed QLP solvers?
2. What tradeoffs should be made among various software engineering principles in a testbed implementation satisfying those requirements?
3. Can a method be devised that mitigates the limitations to effectiveness and efficiency that exist with current QLP solutions?

The first two research questions capture an essential problem for quantum computing research previously expounded on in Section 1.2. If toolsets and workflows are to enable research and the practical application of quantum systems, then software engineering principles and best practices must be applied to the issue. This is

a necessary component to have quantum systems leave the laboratory and enter the office. Towards that end, the often antagonistic concerns of functionality, adherence to standards, and accommodation for the scientific computing environment must be evaluated against one another.

The third, aspirational question concerns limitations of existing QLP solutions. Although there are known methods for determining optimal qubit allocations, the general problem is known to be \mathcal{NP} -hard [47], and existing methods require impractically large number of operations even for small numbers of virtual and hardware qubits. As such, quantum researchers have focused on finding heuristic solutions that are executable in reasonable time. Some existing techniques have emphasized finding sub-graphs of hardware qubits that exhibit desirable reliability traits for single-qubit and entanglement operations. Although ideally a quantum program could be wholly executed on a single sub-graph as a static V2P mapping, this is rarely possible for any non-trivial quantum circuit. As such, it is often true that a series of maps must be identified, each of which identifies some sub-graph of hardware qubits that meet the entanglement constraints of the executing quantum algorithm at a specific moment in time. Thus, other existing techniques have emphasized finding paths to efficiently move virtual qubit states such that these entanglement constraints can be met with a minimum of reliability cost. In contrast to both types of existing techniques, this research assesses the viability of a function that constitutes a tradeoff between these dual concerns—a weighted heuristic that takes into account both sub-graph and pathing optimizations.

1.4 Limitations

This research is intended to advance the field of quantum computation optimization on IBM (transmon) architecture. In particular, by first enabling more efficient

and effective research methods on transpiler optimization, and second by exploring potential avenues for improvement to existing algorithms for solving the QLP. There are numerous avenues beyond this area that also demonstrate some potential for improvement, including work on gate scheduling, novel methods for partitioning circuits beyond the layering method described in Section 2.4, including single-gate errors in optimization decision-making, and developing and using more advanced noise models. These methods, though possibly fruitful, are beyond the scope of the optimization research described in this work. Other potential avenues including circuit characterization schemes and distinct metrics for assessing sub-graph quality are also beyond the scope of this work and are discussed in more detail in Section 5.4.

Moreover, distinct quantum computing architectures, like topological qubits or those exploiting quantum annealing, have distinct concerns and programming models to which this research does not apply. Constraint topologies may not be present or defined in the same form as on transmon architecture—specifically in that other architectures do not always require physical adjacency to perform entanglements or may use a distinct quantum operation to entangle their quantum state.

Finally, the developed quantum testbed is intended to work with IBM’s Qiskit programming library. This means that functionality is not guaranteed—or even intended—in environments that do not have Python and Qiskit installed. Nor is the testbed intended to be used in the same manner as the transpiler routines implemented by [33, 50] or others. Their methods use external applications that Qiskit assembly code are exported to, transformed, and then re-imported into Qiskit, while the intention of the testbed is to be fully integrated into a Qiskit workflow. Although there may be some value in increasing the portability of the testbed or by improving its efficiency through the use of more efficient, lower-level languages like C++ or C#, such work is also beyond the scope of this research.

1.5 Document Overview

Chapter II provides background information on quantum computing, the IBM QX architecture, and previous research on the QLP. Chapter III defines the methodology used to implement and evaluate the proposed Quantum Layout Solver (QLS). Chapter IV presents the results and a comparative analysis against existing methods. And, finally, Chapter V concludes and offers areas of potentially fruitful future work to extend on the initial results provided here.

II. Background and Literature Review

2.1 Overview

This chapter covers information relevant to quantum computing the Quantum Layout Problem (QLP). Section 2.2 begins with a general understanding of quantum computation and its mathematical and operation models. Following this Section 2.3 provides specific background on the IBM QX project which provides public and institutional access to quantum hardware for research purposes. This section also includes information about the Qiskit library and its organization, used in Python to access quantum services. Subsequently, the specific Qiskit Transpiler procedures are described in Section 2.4, and finally the Transpiler discussion is extended into a description of the QLP and previous work that defines it and heuristic solutions proposed to solve it in Section 2.5.

2.2 Quantum Computation Model

Quantum computing can be understood by analogy to classical computation. Just as classical computers exploit the physical properties of an artificially constructed system to model a mathematical operation as a series of physically-instantiated state changes - e.g. using the magnetic properties of platters to allow data storage and retrieval, or manipulating the voltage present in a circuit in a manner that corresponds to a bitwise operation on memory locations that themselves map to variables in the mathematical algorithm being performed – so too can quantum computing be understood as exploiting the quantum properties of an artificially constructed physical system to model a sequence of abstract, mathematical operations [24]. And just as understanding Boolean algebra, operations in the finite field \mathbb{F}_2 , and binary arithmetic are critical to crafting and comprehending the execution of algorithms on classical

computers, quantum computation requires understanding the underlying mathematical model of the space the input and output information occupies. This understanding enables the user to semantically map problems to inputs in the computational space and to, similarly, map outputs in such a space to solutions comprehensible in the semantics of the problem [46].

2.2.1 Hilbert Spaces

The, imperfect, quantum analogue to \mathbb{F}_2 is a Hilbert space \mathcal{H}^N . Hilbert spaces are tuples, $\mathcal{H} = (\mathcal{V}_{\mathbb{C}}^N, F)$ composed of an N -dimensional vector field \mathcal{V} whose component scalar field is \mathbb{C} and an associated dot product $F : \mathbf{V}_i \times \mathbf{V}_j \rightarrow \mathbb{C} : \langle \cdot, \cdot \rangle = \langle \cdot | \cdot \rangle$ is defined for all $\mathbf{V}_n \in \mathcal{H}$ [9]. Each

$$\mathbf{V}_i \in \mathcal{H} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = |\Psi\rangle \quad (1)$$

represents some quantum state $|\Psi\rangle$. In a quantum computing context, \mathcal{H} is given characteristics that make it convenient for representing data in standard, binary format and for easing elements of the calculation in involved. In particular, in the single qubit space \mathcal{H}^2 , \mathbf{V}_i represents a state within a *two-level* system with the basis vectors $\{\mathbf{e}_0, \mathbf{e}_1\}$ defined in the normal way:

$$\begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (2)$$

Again for convenience of notation and semantic mapping, the basis states are most commonly labeled by analogy to classical computer logic: $\mathbf{e}_0 = |0\rangle$ and $\mathbf{e}_1 = |1\rangle$, such

that \mathbf{V}_i can be represented as $\mathbf{V}_i = v_0 |0\rangle + v_1 |1\rangle = \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}$. Additionally, all vectors are normalized to have unit length, such that $\forall \mathbf{V}_i \in \mathcal{H} : |v_0|^2 + |v_1|^2 = 1$.

The total dimensionality, N , of a Hilbert space modeling an n -qubit, two-level system is $N = 2^n$. This is best understood by reference to the fact that the basis of some \mathcal{H}^N is exactly the basis induced by the tensor product on all the component, single-qubit systems:

$$\mathcal{H}^N = \bigotimes_n \mathcal{H}^2 \quad (3)$$

and this basis is generally referred to as the *computational basis* of \mathcal{H} .

For example, if we consider the Hilbert space modeling a two-level, two-qubit system, \mathcal{H}^4 , it is known that each component qubit can be represented in \mathcal{H}^2 as a linear combination of two basis states: $|\phi\rangle, |\theta\rangle = \alpha |0\rangle + \beta |1\rangle$. By linearity, for a state vector $|\Psi\rangle \in \mathcal{H}^4$ composed by $|\phi\rangle |\theta\rangle$, we have:

$$|\Psi\rangle = |\phi\rangle |\theta\rangle \quad (4)$$

$$= (\alpha_\phi |0\rangle + \beta_\phi |1\rangle)(\alpha_\theta |0\rangle + \beta_\theta |1\rangle) \quad (5)$$

$$= \alpha_\phi \alpha_\theta |00\rangle + \alpha_\phi \beta_\theta |01\rangle + \beta_\phi \alpha_\theta |10\rangle + \beta_\phi \beta_\theta |11\rangle \quad (6)$$

Although the reverse mapping of a given $|\Psi\rangle$ to individual vectors in \mathcal{H}^2 is not guaranteed – and in fact fails exactly when $|\Psi\rangle$ represents an entangled state – the example serves well to demonstrate that the computational basis of \mathcal{H}^N system can be naturally interpreted as all possible bitstrings of length N . This also provides a convenient mapping from basis states (usually returned by measurement operators) to integer solutions to a problem encoded in the quantum computation; it also provides a natural order on the bases. As such, it is common to render the basis states of \mathcal{H}^N as $\{|0\rangle, |1\rangle, |2\rangle, \dots, |N-1\rangle\}$ [34].

2.2.2 Qubits

Qubits are the quantum analogue to classical ‘bits’ in computing; they both represent a two-level system and act as carriers of information for the computation, although they are significantly different in a variety of ways that are key to understanding quantum advantage in certain computational tasks. While a classical bit is a scalar that takes exactly the values $\llbracket 0, 1 \rrbracket$ - and therefore store exactly one bit of information - qubits are multi-dimensional vectors and take continuous, complex arguments in each level. In particular, let q be some isolated qubit with access to two relevant energy levels (in general, there are more than two energy levels available to the qubit, but higher-order levels are not used and a qubit’s excitation into higher states is frequently pathological behavior [44]); we denote these energy levels with the symbols $|0\rangle$ and $|1\rangle$, respectively. Then the qubit can be represented by the state vector $|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle$, with $\alpha, \beta \in \mathbb{C}$. In this context, α and β are the *amplitudes* of the state $|\Psi\rangle$ in the eigenbasis of the model of the single-qubit system, and if α and β are each non-zero, then q is in a *superposition* of the states $|0\rangle$ and $|1\rangle$.

The ability of a qubit to exist in a superposition of its basis states is a significant feature in quantum advantage, for example in enabling quantum parallelism (see Section 2.2.5).

In addition to amplitudes, qubits have a *phase*, which characterizes their rotation in the complex plane [34]. If a qubit has a phase, then this is represented by the inclusion of an additional factor: $|\Psi\rangle = e^{k\pi i} \alpha |0\rangle + e^{m\pi i} \beta |1\rangle$ for $k, m \in [0, 2)$. Although a qubit can have a phase in either component, it is often convenient to factor this into a global phase and a relative phase solely on the $|1\rangle$ component, since global phase factors are physically immeasurable and can be discarded [34]. “[T]wo amplitudes a

and b differ by a relative phase if there is a real θ such that $a = \exp(i\theta b)$ [34].

$$|\Psi\rangle = e^{k\pi i}\alpha|0\rangle + e^{n\pi i}\beta|0\rangle \quad (7)$$

$$|\Psi\rangle = e^{k\pi i}[\alpha|0\rangle + e^{(n-k)\pi i}\beta|0\rangle] = e^{k\pi i}|\Psi'\rangle \quad (8)$$

Let \hat{M} be a projective measurement operator, then:

$$\hat{M}|\Psi\rangle = \langle\Psi'|\hat{M}|\Psi\rangle\hat{M}|\Psi'\rangle \quad (9)$$

$$= \hat{M}|\Psi'\rangle \quad (10)$$

$|\Psi\rangle$ can now be interpreted as a state with a (discardable) global phase $e^{k\pi i}$ and a relative phase $e^{m\pi i}$.

2.2.3 Unitary operators

Physical constraints on the evolution of quantum mechanical systems determine the set of algebraic operations permitted on \mathcal{H} . Most important, because all non-measurement quantum operations must be conceptually reversible, then they must be norm-preserving [41]; that is, given an operator \hat{A} that represents such a transform on \mathcal{H} and $\mathbf{V}_i, \mathbf{V}_j \in \mathcal{H}$:

$$\langle\hat{A}\mathbf{V}_i, \hat{A}\mathbf{V}_j\rangle = \langle\mathbf{V}_i | \hat{A}^\dagger \hat{A} | \mathbf{V}_j\rangle = \langle\mathbf{V}_i|\mathbf{V}_j\rangle \quad (11)$$

And it thus true that $\hat{\mathbf{A}}$ is unitary $\iff \hat{A}^\dagger \hat{A} = \mathbb{I}$ [41]. We can then conceptualize state changes on this space as rigid rotations of the underlying Hilbert space. This view continues to hold for spaces representing systems with greater numbers of qubits.

Quantum computation on transmon architectures, like classical computation, takes place on a machine that implements a sequence of instructions that alter the

machine state. Some subset of the instructions a machine is capable of interpreting are fundamental in the sense that they are directly, atomically implemented at the hardware-level of the machine; the remainder of the instructions are then aliases for some sequential composition of this fundamental set. True quantum computation, like its classical counterpart, then requires that this fundamental subset of instructions be capable of composing arbitrary algorithms [29]; if it does so, we refer to it as the set of *basis gates* for the quantum architecture, and all gates that are implementable on the architecture but that are not basis gates are composed of sequences of basis gates. The basis gates relevant to this work are discussed in Section 2.3.

2.2.4 Measurement Operators

As referenced prior, measurement operators compose the sole class of non-unitary operations allowable on a quantum system, because measurement is not a reversible process [34]. Instead, as projectors measurement operators must be Hermitian - i.e. given an operator $\hat{\mathbf{H}}$, $\hat{\mathbf{H}}$ is Hermitian $\iff \hat{\mathbf{H}} = \hat{\mathbf{H}}^\dagger$.

It is not possible to measure the exact quantum state, $|\Psi\rangle$ of a given quantum system at a given point in time [51]. Instead, the application of a measurement operator to $|\Psi\rangle$ returns exactly one basis vector of \mathcal{H}^N , with an associated probability proportional to the original state's amplitude in that basis. Given $\text{Basis}(\mathcal{H}^N) = \{|\omega_0\rangle, \dots, |\omega_{N-1}\rangle\}$, then:

$$P\left(\hat{H}|\Psi\rangle = |\omega_i\rangle\right) = \langle\omega_i | \Psi\rangle^2, \text{ and} \quad (12)$$

$$1 = \sum_{i=0}^{N-1} \langle\omega_i | \Psi\rangle^2 \quad (13)$$

As noted before, although measurement operators cannot distinguish the global phase of an arbitrary state—regardless of the bases chosen for the projection—a relative

phase does impart a physically measurable difference in quantum state, but only if measured in a basis where the amplitudes of the basis states vary by more than a relative phase. Consider the state $|\theta\rangle = e^{k\pi i}\beta|1\rangle$ in the standard basis. Measured in this basis, we observe that since $|e^{k\pi i}|^2 = e^{k\pi i} \cdot [e^{k\pi i}]^* = 1$ then $P(|\theta\rangle = |1\rangle) = |\beta|^2$ for all k .

However, appropriate measurement bases may be chosen to distinguish relative phases, depending on the phase. For example, consider $|\Psi\rangle_{[e]} = \frac{1}{\sqrt{2}}(|0\rangle + e^{\pi i}|1\rangle)$, as before, with a measurement in the standard basis we have $P(\hat{M}|\Psi\rangle = |0\rangle) = P(\hat{M}|\Psi\rangle = |1\rangle) = \frac{1}{2}$. However, by applying a change of basis transform:

$$\text{Given, } H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$H|\Psi\rangle_{[e]} = H\left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \quad (14)$$

$$H|\Psi\rangle_{[e]} = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad (15)$$

$$|\Psi\rangle_{[H]} = 0|0\rangle_{[H]} + 1|1\rangle_{[H]} \quad (16)$$

A measurement in the H basis would therefore distinguish a relative phase that is otherwise immeasurable in the standard basis, since $P(\hat{M}|\Psi\rangle_{[H]} = |0\rangle_{[H]}) = 0$, while $P(\hat{M}|\Psi\rangle_{[H]} = |1\rangle_{[H]}) = 1$. Even if immeasurable however, phase is often introduced and later eliminated as an intermediate step in quantum algorithms to select or distinguish states [14].

2.2.5 Quantum Algorithms

Quantum algorithms are distinct from classical algorithms in a variety of ways. Of foremost concern is that quantum algorithms are probabilistic [45]. As such any quantum procedure is run repeatedly from which a distribution of state vectors are measured. Although all quantum computation will necessarily engage with the physics of quantum systems occupying a continuous configuration space, it is convenient to differentiate two broad families of techniques: continuous and discrete. That is, some methods for implementing quantum algorithms depend on permitting a constructed quantum system to continuously vary over time until some certain condition is met, while others assume a discrete time evolution of the system under the application of distinct gates [12].

2.2.6 Quantum Annealing

Quantum annealing is a technique developed by Kadowaki and Nishimura [23] and used primarily in adiabatic quantum computing, as on D-Wave systems [16]. Encoding an algorithm as a quantum annealing process first requires formulating the problem being solved as a constraint satisfaction problem with a well-defined objective function [23]. In general, the goal is to map candidate solutions to the problem to the basis states of the underlying quantum system, most commonly such that each qubit's state is mapped to a single degree of freedom in the solution configuration space, and a measured basis state then defines the total configuration best satisfying the constraints by specifying all the relevant variables corresponding with those degrees of freedom.

Once the problem is mapped appropriately, the quantum system implemented by the annealer is tuned. In particular, like all quantum systems the (nominally) isolated quantum system of the annealer has an associated Hamiltonian constraining

its time-evolution. If the quantum system can be transformed such that its associated Hamiltonian reflects the constraints of the initial problem, then identifying the ground state of this Hamiltonian—which is an eigenvector and thus a measurable state—allows the user to invert the mapping and recover a particular configuration that satisfies the given constraints and maximizes the objective function.

The challenge is thus to tune the quantum system such that it is governed by a Hamiltonian that generates an energy landscape of the correct “shape” and then to ensure that this Hamiltonian is in its ground state at the time of measurement. The details for how these tasks are accomplished can vary based on the particular annealing method being used and the hardware involved [16]. On D-Wave systems, the Hamiltonian can be developed by changing the coupling values between individual qubits—that is, by changing the entanglement between them—and by varying the *bias* for each qubit, which changes the value of the scalar components of its state vector by applying a magnetic field, thereby modifying the intrinsic probability the qubit is measured in each basis state [7]. The ground state is maintained by slowly applying a transverse magnetic field to the quantum system after a uniform superposition of all possible basis states is introduced. The magnetic field then, ideally, causes a smooth transition from the ground state of this system to its first excited state, and then via quantum tunneling, into the ground state of the related Hamiltonian that encodes the objective-function-maximizing state.

2.2.7 Quantum Gate Model

Unlike quantum annealing, which uses continuously varying fields to manipulate the quantum system, and so in some sense embodies a more physical approach to modeling quantum computation, quantum gate models of computation derive more inspiration from existing classical computation by treating the procedure as a discrete

algorithm. Gate-modeled quantum computation is “the generalization of digital computing where bits are replaced by qubits and logical transformations by a finite set of unitary gates that can approximate any arbitrary unitary operation. A classical digital circuit transforms bit strings to bit strings through logical operations, whereas a quantum circuit transforms a special probability distribution over bit strings—the quantum state—to another quantum state” [12]. The user begins with a mapping, as above, that relates measured basis states to solutions to the problem being solved, with the goal of transforming the original quantum system into a state that, when measured, is likely to return the basis state corresponding to the correct solution. The gate model of quantum computation is more flexible in the types of problems to which it applies, as it is not constrained to finding solutions encoded only as the ground state of a particular Hamiltonian that must be constructed.

Gates are defined by the specific architecture, analogous with classical computers, as discrete operations on one or more qubits that transform the state in a defined way; that is, given the desired solution state $|\omega\rangle$ and some sequence of unitary gates implementing a sequence of operators $(\hat{U}_0, \hat{U}_1, \dots, \hat{U}_{n-1})$ such that $\hat{U}_i |\Psi\rangle_i = |\Psi\rangle_{i+1}$, the desired outcome is the state $|\Psi\rangle_n = \hat{U}_{n-1} \hat{U}_{n-2} \dots \hat{U}_0 |\Psi\rangle_0$ which has the property $\langle \omega | \Psi_n \rangle^2 \geq 0.5$

2.3 Qiskit and IBM QX Architecture

Qiskit is an open-source software development kit sponsored by IBM, designed to provide a standardized interface for programming IBM QX Architecture Quantum Computers (QCs) [21]; it is provided in the form of a Python 3.x library.

Qiskit is divided into sub-libraries, each specializing in particular elements of quantum computation. These sub-libraries are referred to within Qiskit as “elements” [21], as a mnemonic referencing the pre-scientific belief that the known world was com-

posed by combinations of air, fire, water, etc. In the Qiskit context, the elements are *Terra*, *Aer*, *Ignis*, *Aqua* and although technically a component of the Terra library, the IBMQ (and related) modules are often considered a fifth component - though frustratingly, they are not associated with a name referencing the traditional “spirit” component completing the classical pentalogy. A brief overview of the most relevant elements is included below.

2.3.1 Terra

Terra is “the foundation on which the rest of Qiskit lies,” which “provides a bedrock for composing quantum programs at the level of circuits and pulses”[21]. It defines a particular construction of quantum computational procedures using its circuit, pulse, and transpiler modules. At the top level, circuits are defined by Qiskit as objects containing details about available quantum gates, sequences of gates to be executed, metadata about a given procedure, virtual qubits and classical bits used, and the compilation and transpilation procedures necessary to define the circuit as a computable sequence. Each circuit contains at least one *quantum register* and zero or more *classical registers*. Each register is simply a Qiskit wrapper around a Python list object: which is to say a generically-typed, integer-indexed array. Registers have names, and own a defined, finite number of bits to which they provide access methods. Each bit is then defined in this context as a tuple of two elements; the first element is itself a 2-tuple containing the name and size of the parent register, and the second element is the index associated with the qubit. Figure 1 shows a schematic view of this structure.

Although Terra allows each QC to define and provide its own basis set of gates, at this time all IBM computers use the same set. On the IBM QX architecture, the

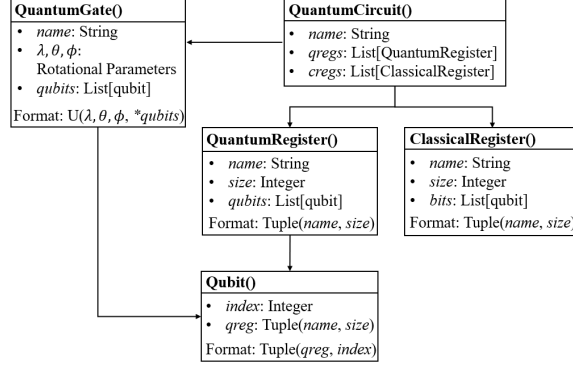


Figure 1: A schematic view of the circuit model constructed by Qiskit Terra

basis gate set is $\{U_1, U_2, U_3, Id, CX\}$, where CX denotes the sole two-qubit gate.

$$CX = CNOT = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & 1 \\ & & 1 & 0 \end{pmatrix} \quad (17)$$

$$Id = \mathbb{I} = \begin{pmatrix} 1 & \\ & 1 \end{pmatrix} \quad (18)$$

$$U_3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda} \sin(\frac{\theta}{2}) \\ e^{i\phi} \sin(\frac{\theta}{2}) & e^{i(\lambda+\phi)} \cos(\frac{\theta}{2}) \end{pmatrix} \quad (19)$$

$$U_1(\lambda) = U_3(0, 0, \lambda) \quad (20)$$

$$U_2(\phi, \lambda) = U_3(\frac{\pi}{2}, \phi, \lambda) \quad (21)$$

Each gate is represented in Qiskit by a method on objects of the `QuantumCircuit` class, and accepts one or more arguments denoting the target bit(s) of the operation. Although not in the basis set, Table 1 shows the symbol and matrix operator associated with other, common quantum gates.

The SWAP gate is of particular importance to transpiler optimization research.

Applying a SWAP operator to two adjacent qubits exchanges their quantum state. Let \hat{S} represent the SWAP operator defined in Table 1 and let q_0, q_1 be two adjacent qubits on some hardware topology such that q_0 encodes the quantum state $|\Psi\rangle$ and q_1 encodes the state $|\Theta\rangle$, then $\hat{S}|q_0q_1\rangle = |q_1q_0\rangle = |\Theta\Psi\rangle$, or equivalently $\{q_0 : |\Psi\rangle, q_1 : |\Theta\rangle\} \xrightarrow{\hat{S}} \{q_0 : |\Theta\rangle, q_1 : |\Psi\rangle\}$. On IBM QX QCs, the SWAP operator is implemented as a series of three CX gates where the direction (i.e. the control/target relationship) of the second CX gate is flipped—which is accomplished by bracketing the flipped gate with two Hadamard gates. Let \hat{C} represent the operator associated with the CX gates, then $\hat{S}|q_0q_1\rangle = \hat{C}\hat{H}\hat{C}\hat{H}\hat{C}|q_0q_1\rangle$.

An important consequence of this architectural implementation is that SWAP operations are expensive in terms of reliability cost, as they introduce noise proportional to the cube of the noise introduced by the CX gate. As an example, an otherwise noiseless circuit performing a CX over an edge of reliability 0.9 would have a final reliability of 0.9 after the CX, while that same circuit instead performing a single SWAP over that edge would result in a final reliability of ≈ 0.73 —nearly 20% less reliable.

The Pulse module of Terra can be directly accessed by the end user, but is most often a transparent translation layer between the gate methods and the physical hardware. Quantum state on IBM transmon qubits is modified by the application of microwave pulses whose frequency is determined by the qubit being addressed, and whose amplitude, duration, and shape are determined by the gate being applied [28]. The Pulse module schema reproduces the circuit schema at a lower level: the top-level container corresponding with the circuit object is the *pulse schedule*, which is an ordered sequence of defined pulses intended to implement a sequence of discrete quantum state changes. Each pulse schedule owns its component pulse object, each of which corresponds with the action of a gate object from the circuit module. Because

exposure of the pulse interface breaks abstraction, it is useful for experimentalists but has limited usefulness for the creation and execution of quantum programs [21]. A short example using the Terra library to create a two-qubit circuit implementing the Bell state is shown in Figure 2.

Terra additionally provides a transpile module, which provides the methods necessary to take a given circuit and quantum hardware system and transform the former to a format compatible for execution on the latter. Transpilation, also known as source-to-source compilation [4], is a process of mapping source code in a given language to source code at the same level of abstraction; this is distinct from compilation, which maps source code at a given level of abstraction to a lower-level language like assembly or bytecode. Pasquier *et al.* define transpilers as “software programs that take source code in a given language as input and generate the equivalent source code in a second language at an equivalent level of abstraction,” or that provide “translation of source code between different versions of the same language” [36]. In this context, we are concerned with this second sense. The Qiskit transpiler performs Python-to-Python mappings of quantum circuits that consume information about the coupling map of a given QC and transform the circuit code to conform to the constraints created by this coupling map. It additionally performs gate reductions that eliminate redundant procedures—for example, using double-negation elimination to reduce the total number of applied gates. The Qiskit Transpiler also provides a series of analysis functions that modify the object attributes of the circuit being transpiled [21]. The structure of the Qiskit transpiler is discussed in more detail in Section 2.4.

2.3.2 Aer

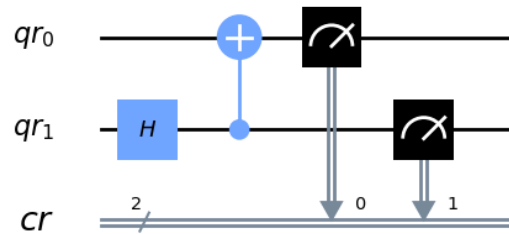
Aer is the Qiskit module that “permeates all Qiskit elements,” and helps researchers “understand the limits of classical processors by demonstrating to what

```

1      # Standard qiskit
  ↪ imports for creating a
  ↪ quantum circuit
2      from qiskit import
  ↪ QuantumCircuit,
  ↪ QuantumRegister,
  ↪ ClassicalRegister, IBMQ
3
4      IBMQ.load_account
  ↪ () # Requires pre-
  ↪ existing IBM API key
5
6      qr =
  ↪ QuantumRegister(2, name='
  ↪ q')
7      cr =
  ↪ ClassicalRegister(2, name
  ↪ ='c')
8      qc =
  ↪ QuantumCircuit(qr, cr,
  ↪ name='qc')
9
10     # Adds a hadamard
  ↪ gate to the 0th qubit of
  ↪ the quantum register
11         qc.h(0)
12
13     # Adds a CX gate,
  ↪ controlled by the 0th
  ↪ qubit and targetted at
  ↪ the 1st qubit
14         qc.cx(0, 1)
15
16     # Use a
  ↪ measurement operator
  ↪ mapping every qubit to a
  ↪ classical bit, matched on
  ↪ index
17         qc.measure(qr, cr)
18

```

(a) Simple quantum circuit program code



(b) The circuit represented in “Composer” style

Figure 2: Simple quantum program in Qiskit and its associated circuit

extent they can mimic quantum computation” [21]. It is designed primarily to provide simulators and simulation methods for executing quantum routines on classical hardware. The simulators provided by Aer are run locally on the machine executing the code, instead of being submitted as a remote job to IBM hardware. They expose interfaces mimicking that of the real hardware, to enable interoperability with circuits constructed for actual quantum computation. The simulators generally do not include noise-components and so are useful for verifying algorithmic correctness or to establish truth values. Additionally, noise model objects can be derived from real hardware and applied to some Aer simulators to provide rapid, local testing of algorithms for noise-sensitivity or noise-mitigation purposes.

In particular, Aer provides three simulators with differing purposes and interfaces. The `QASM_simulator`, where “QASM” denotes the quantum assembly language, is designed to simulate a circuit as it would behave on an IBM QC, with the exception that the default QASM simulator is noise-free. It accepts circuits containing measurement gates and given a circuit to execute will return as a result one vector in the computational basis for each execution. It additionally permits noise-models to be introduced (derived from empirical machine data or manually crafted to exhibit specific properties). The QASM simulator is the closest match to a simulation of actual quantum hardware available in Qiskit.

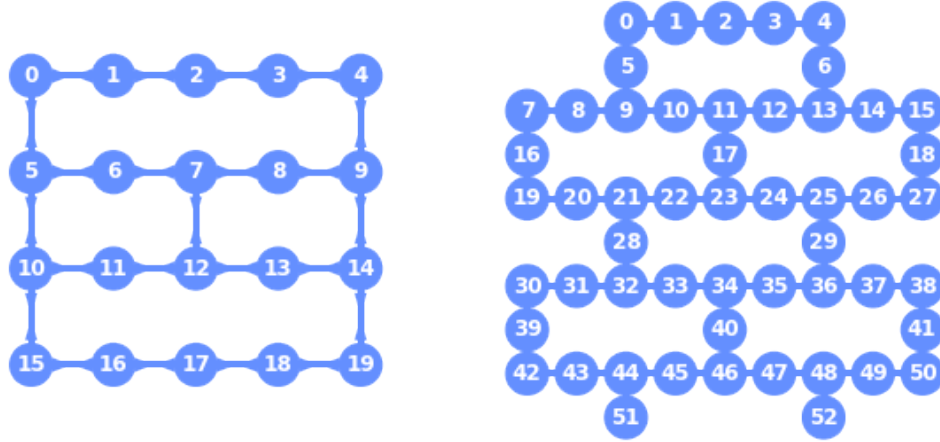
Aer also contains a `unitary_sim` and a `statevector_sim`. Both simulators are intended to expose more of the computational process than is otherwise physically possible, and neither accepts measurement gates in the circuits they simulate, since measurement collapses the quantum state being exposed by the simulators. Specifically, the unitary simulator accepts a quantum circuit that does not contain measurement gates and returns the unitary matrix composed by the sequence of gates in the circuit. That is, given a sequence of unitary gates serially applied by the circuit

$(U_0, U_1, \dots, U_{n-1})$, the unitary simulator returns the matrix $A = U_{n-1}U_{n-2} \dots U_1U_0$. The state vector simulator is similar, but returns the state vector defining the quantum state at the termination of execution of a given quantum circuit. Given a serially applied sequence of gates $(U_0, U_1, \dots, U_{n-1})$ such that $U_i |\Psi\rangle_i = |\Psi\rangle_{i+1}$, the output of the state vector simulator is the vector $|\Psi\rangle_n = U_{n-1}U_{n-2} \dots U_1U_0 |\Psi\rangle_0$. This is distinct from the QASM simulator, since the vector returned from the latter is the probabilistic result of the projection of $|\Psi\rangle_n$ onto the computational basis.

2.3.3 IBM Quantum Hardware

IBM first made QCs available to the public as part of their IBM QX Experience program in 2017 [20]. From 2017 to 2019, a total of approximately 11 QCs were made available for commercial and public use, with qubit counts ranging from four to 53 [21]. QCs are most often classified according to the type of qubit they use and the information-carrying medium of that qubit. And in particular, it is useful to distinguish two major classifications: microscopic and macroscopic. Microscopic qubits are constructed around quantum-scale qubits that naturally behave according to observable quantum principles [48]. Conversely, macroscopic systems are human-scaled devices that require exotic conditions to produce exploitable quantum behavior. “Quantum phenomena in mesoscopic qubits require extremely low electrical resistance for viability and are realized in the form of electrical circuits on ICs. These devices are often referred to as superconducting solid-state qubits” [48]. IBM QX machines use macroscopic, solid-state, superconducting qubits referred to as transmons [21].

Transmon qubits exploit the quantum coherence of a superconducting state to minimize decoherence and noise effects, and employ a Josephson junction as a non-linear element to create an anharmonic system that provides a distinctly-addressable, two-level system; this also means that transmon qubits use charge as the information



(a) “Poughkeepsie,” a 20-qubit device (b) “Rochester,” a 53-qubit device

Figure 3: Example topologies available on IBM QX QCs

carrier [18]; in many respects transmon qubits can be interpreted as modified Cooper Pair Boxes (CPBs) [18]. The quantum state of transmon qubits is manipulated by the application of microwave pulses at specific frequencies and specific durations that are driven by the underlying chip characteristics and the desired quantum operation. One significant implication of this control methodology is that all transmon qubits must be co-located with a microwave waveguide that permits microwave pulses to be targeted at them, and similarly entanglement operations require that qubits involved in the entanglement must jointly share a single waveguide to permit a single pulse to affect a multi-qubit system; this places a practical limit on the number of qubits that can be made adjacent, i.e. that can share waveguides so that they can be entangled, and the majority of qubits across all IBM QX devices have either two or three neighbors. The adjacency relationships described here are represented in topology graphs available for each device, where nodes represent individual transmon qubits, and edges represent a shared waveguide between nodes that enables an entanglement operation to be performed; Figure 3 shows the topologies of two IBM QX machines.

Although not unique to transmon architectures, the noise levels of operations on

transmon QCs is also significant in determining the feasibility of successfully executing quantum circuits on available hardware [38]. Each qubit and edge on each IBM device has an error rate associated with each gate capable of being executed on that qubit or pair. The error rate can be conceptualized as the probability that executing the given gate on the target(s) results in a quantum state matching that of the same circuit executed on a noiseless simulator.










Gate Name	Gate Symbol	Associated Matrix
Hadamard		$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix}$
NOT / Pauli-X		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
S / Phase Gate / Z90		$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
T		$\begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix}$
CX / Controlled NOT		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
SWAP		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Toffoli		$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 0 & 1 \\ & & & 1 & 0 \end{pmatrix}$

Table 1: Summary of commonly used gates and their matrix representations

2.4 IBM QX Transpilation

The IBM QX Transpiler (IQT) is responsible for transforming quantum circuits from their original format to one conducive to being executed on hardware. The transpiler’s operations can be understood as comprising *compilation* functions and *transpilation* functions. In general, compilation functions modify a given circuit by reducing it to a more fundamental form; in classical computation, this is most commonly exemplified by the mapping of code functions from their representation in the original, human-readable language to a sequence of machine-code instructions, each of which has an immediate reference to some physical state change in the executing computer [1]. Conversely, transpiler functions execute a functionality-preserving transform to an input a circuit at a given level of abstraction and create and return a different circuit at the same level of abstraction. Symbolic substitution with macro definitions, stripping of comments, or transliteration between high-level languages are all examples of transpilation functions.

Functions of either type are encapsulated by the IQT into `Pass` objects. Conceptually, each pass executes one, simple transpiler function; architecturally, each `Pass` is an object of the `Transpiler` class that defines a `run` method that performs a specific transform. For example, a single pass might substitute all references to logical, user-defined quantum registers with references to a single, combined quantum register that simplifies operations for subsequent passes. Passes are either *analysis* passes (i.e. of the type `AnalysisPass`), which collect information about the state of the circuit and store it for use by subsequent operations, or they are *transformation* passes (i.e. of the type `TransformationPass`), which actually change, delete, or insert operations into the circuit.

Passes are ordered and controlled by a *PassManager* object, which also provides shared memory so that the results of analysis passes can be consumed by others. The

PassManager not only executes the passes in their defined order, but also permits passes to be looped over and to be conditionally executed. The particular passes used by a given PassManager depend on the transpiler options used for the specific program being transpiled.

Alongside predefined sets of passes Qiskit provides as defaults, developers may modify these sets by adding or removing passes or changing flow control criteria or may alternatively define their own bespoke set of passes and add them to an empty PassManager.

2.5 Quantum Layout Problem

Given a quantum circuit to be run that includes entanglement operators (CX gates) and some particular machine topology, as in Figure 3, the transpiler must select a *layout* that maps each virtual qubit defined in the circuit to a physical qubit existing in the machine topology. Since qubit states cannot be wholly known nor cloned [51], then if two qubits are to entangle—that is, to share state—then they must be co-located on the topology so that they share a microwave waveguide [30].

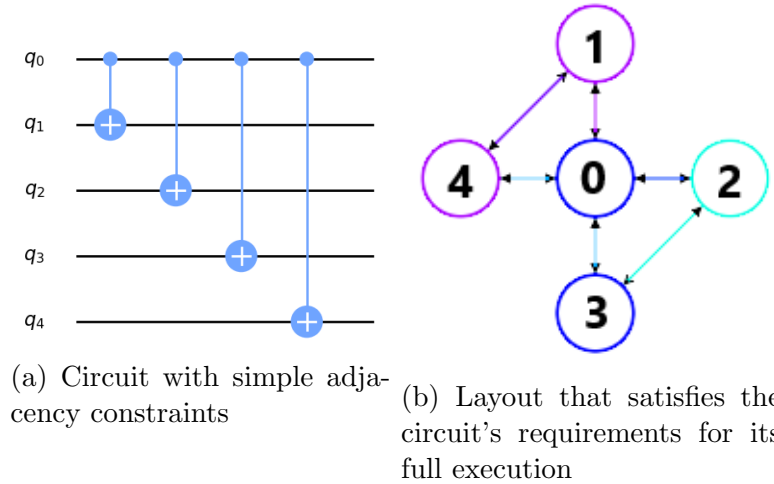


Figure 4: An example simple circuit and an associated, fully satisfactory layout

In simple circuits, this layout can be easily determined and is static for the duration of the execution of that circuit, as in Figure 4.

However, more complex circuits often have connectivity constraints that cannot be met by a static layout. This is illustrated in Figure 5, where there is no sub-graph on the hardware topology that permits all entanglement constraints to be met with a single map; no layout permits qubits q_1 and q_2 to communicate with each other and with q_0 simultaneously while also allowing qubit q_1 to control q_3 . Issues with static layouts occur very frequently for all but the simplest circuits, since Noisy Intermediate-Scale Quantum (NISQ) computers are in part characterized by their limited connectivity [38]. As such, IBM developed the *layering* technique currently used by the IBM QX transpiler.

Given a quantum circuit, the transpiler first partitions the circuit into a set of layers $\{\ell_0, \ell_1, \dots, \ell_{n-1}\}$, where each layer contains the maximum number of gates that have no data dependency between them. Each layer ℓ_i can then be associated with a layout λ_i that maps virtual qubits in the layer to physical qubits on the

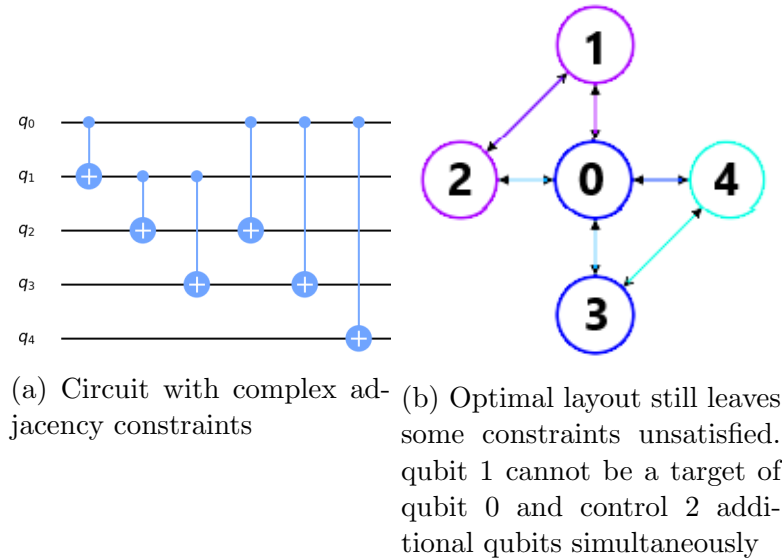


Figure 5: An example complex circuit and an associated, partially satisfactory layout

machine topology. If λ_i ensures qubits requiring entanglement via CX operations are co-located on the machine topology, then we say that λ_i satisfies ℓ_i : $\lambda_i \vdash \ell_i$. Since by construction each layer is independent from the layer preceding it, then once some set of satisfying layouts has been calculated, the transpiler is free to insert SWAP operators into the circuit to migrate the quantum state associated with some virtual qubit in the circuit to a new, physical qubit and thereby transform the mapping $\ell_i \rightarrow \ell_{i+1}$.

Because SWAP operations are particularly noisy and prone to causing execution failure for a given circuit, it is of significant interest to develop techniques for finding and assigning sequences of layouts that minimize the number of SWAP operations required to ensure every layer has a satisfying layout. Let \mathcal{M} denote a mapping for a quantum algorithm that consists of a sequence of layouts, where each layout meets the connectivity constraints of its associated layer:

$$\mathcal{M}_i = (\lambda_0, \lambda_1, \dots, \lambda_n) \text{ such that } \lambda_i \vdash \ell_i \quad (22)$$

The goal of this research is to select the best \mathcal{M}_i such that the reliability of each λ_i and the total SWAPs required to permute between them are jointly optimized in a manner that maximizes the probability that arbitrary quantum circuits executed on a given IBM QX QC return their desired state vector.

2.5.1 Optimization Techniques (Previous Work)

IBM made their first quantum devices available for public and research use in 2017. The vast majority of work on the QLP for QX architectures was published in 2018, though work on platform-dependent and simplified models extends back at least ten years further. This work can be understood as relating to one or both of two distinct, but related problems: connectivity optimization and distance optimization.

Connectivity algorithms emphasize finding Virtual-to-Physical (V2P) mappings that are optimal against some metric, e.g. sub-graph connectivity [13]; distance algorithms emphasize finding the optimal ways to assign qubits so as to minimize their traversal between qubits in the coupling map. In short, connectivity methods often find the best static layout for a given circuit or layer, while distance methods improve the dynamic behavior of the layouts during execution but may sacrifice per-layer reliability.

Early works focused primarily on models that worked for tightly constrained topologies permitting simplifying assumptions, or sought optimal solutions in small search spaces—with performance on the order of $\mathcal{O}(e^n)$ [27, 32]. Maslov in 2008 first formulated a version of the QLP, the “quantum circuit placement problem” [32]:

The **quantum circuit placement problem** is to construct an injective (one-to-one) function $P : \{q_1, q_2, \dots, q_n\} \mapsto \{v_1, v_2, \dots, v_m\}$ such that with this mapping the runtime of a given circuit is minimized. The gate’s $G(q_i, q_j)$ execution cost is defined by the mapping $q_i \mapsto v_i, q_j \mapsto v_j$ according to the formula

$$\text{GateOperatingTime}(G(q_i, q_j)) := W(v_i, v_j) * T(G(q_i, q_j)).$$

Maslov’s work applied to Nuclear Magnetic Resonance (NMR) QCs and assumed coupling maps with properties distinct from solid-state superconducting qubit architectures. This work also defined the first layering procedure for satisfying sub-circuits and concatenating them with SWAP interludes, although Maslov’s algorithm defined layers by the maximum-length sub-circuit that could be satisfied with a layout on the NMR coupling map. Maslov also provided a linear time algorithm for determining SWAP paths using graph coloring on recursively-defined sub-graphs.

In 2014, Lin developed the first QLP heuristic for a plausible superconducting qubit architecture, but assumed grid connectivity [27]. Like many early efforts, Lin’s work makes use of an external tool to transpile circuits into executable formats for

the (assumed) architecture. Lin distinguishes algorithms for *qubit placement* and *qubit routing*. However, the nature of the tiled architecture Lin focuses on limits the applicability of the work to existing IBM systems, as the algorithms inflexibly depend on assumptions about what paths exist between qubits.

Shafei in 2014 extends Mixed Integer Programming solutions from circuit placement problems occurring in very large scale integration for classical integrated circuits to develop a qubit mapping on a grid architecture, similarly to Lin [40]; however the algorithm provided does not extend to arbitrary coupling topologies [47].

Post-2017, the pace of work on the QLP accelerated drastically as did applicability to real-world architectures and circuits. In early 2018, Siraichi, Dos Santos, and Pereira first formalized both components of the QLP in the context of a superconducting qubit architecture over arbitrary coupling maps that resemble those found on IBM QX systems, including analyzing performance on the ibmqx4 QC [47]. The authors define two problems analogous to the previously defined connectivity and distance sub-problems: first, the *Qubit Assignment Problem* defined as a decision problem on the existence of a V2P mapping from “pseudo qubits” (virtual qubits) to physical qubits that satisfy Ψ , the ordered dependency list of a given program; second, the *Swap Minimization Problem*, which they formulate as a decision problem on a coupling graph, Ψ , and a constant integer K , returning whether or not “we can use up to K swaps to produce a version of Ψ that complies with G_q ” [47]. Their solution to the assignment problem requires creating a coupling graph of Ψ whose nodes are virtual qubits and whose edges represent a “control relationship” (i.e. a CX gate with those two qubits), weighted by count. SWAP minimization is solved by a recursive algorithm shown in Figure 6.

Of note is that the technique presented does not institute any “layering,” and therefore only works on circuits and architectures that permit a fully compliant map-

Given a coupling graph $G_q = (Q, E_q)$, an initial mapping ℓ_0 , and the dependencies Ψ , for each i in the domain of Ψ , let $(p_0, p_1) = \Psi(i)$. If $(\ell_0(p_0), \ell_0(p_1)) \notin E_q$, then:

1. if (p_0, p_1) appears in Ψ two or more times, then we use a swap to move p_1 closer to p_0 in the coupling graph, update ℓ_0 and re-evaluate the four cases in this algorithm;
2. else if the edge $(\ell_0(p_1), \ell_0(p_0)) \in E_q$, then we use a reversal between $\ell_0(p_1)$ and $\ell_0(p_0)$;
3. else if $\exists q \in Q$, such that $(\ell_0(p_0), q) \in E_q$, and $(q, \ell_0(p_1)) \in E_q$, then we use a bridge between $(\ell_0(p_0), \ell_0(p_1)) \in E_q$;
4. else we create swaps, i.e., apply step (1) onto $(\ell_0(p_0), \ell_0(p_1))$.

Figure 6: SWAP minimizing algorithm to extend the initial layout to cover the entire dependency list, from Siraichi *et al.* [47]

ping prior to execution. This is primarily a result of hardware limits at the time driving research towards small circuits on small topologies; in particular the provided layout algorithm was tested on a five qubit IBM machine with relatively high connectivity.

Quickly thereafter, in March 2018, Zulehner, Paler, and Willie developed the first layered approach implemented in IBM QX devices. Given a circuit already decomposed to operations in the basis gate set, they define a main objective: “to minimize the number of elementary gates which are added in order to make the mapping CNOT-constrain-compliant” [53]. The process introduced is divided into three primary components: first, the circuit is divided into layers; second, compliant mappings are derived for each layer, finally SWAP operations are introduced as required to transform between identified compliant mappings between layers.

The layering process used here will continue to be used throughout most of the following work by this and other authors, and is the dominant paradigm for accomplishing complete mappings for quantum circuits on IBM QX architecture. Given a circuit and an empty layer ℓ_i , the algorithm consumes gates from the circuit, in order, and adds them to ℓ_i until it encounters a gate that cannot be concurrently applied

with the gates already in ℓ_i , at which point ℓ_i is complete, a new layer ℓ_{i+1} is created, and the current gate is added to it, after which the process continues with $\ell_i + 1$. This algorithm is greedy in that it prioritizes adding gates to the existing layer over creating new layers, so all gates composing the circuit are moved as far left (as early in the execution process) as possible.

Once the circuit has been layered, then compliant mappings can be found for each layer, individually. In particular, Zuheler *et al.* define *permutation layer* π_i as the sequences of SWAP operations required to transform mapping λ_{i-1} , which satisfies the dependency constraints of ℓ_{i-1} , to λ_i ; this results in a mapping that consists of a sequence of layouts interleaved with permutations: $(\lambda_0, \pi_1, \lambda_1, \pi_2, \lambda_2, \dots, \pi_{n-1}, \lambda_{n-1})$. Given an existing $\lambda_i \vdash \ell_i$, λ_{i+1} is found by defining a configuration space whose nodes are V2P maps (i.e. node n represents λ_n) and each edge represents a single SWAP operation that transforms one map to another, and using an A* search over this space to find a compliant λ_{i+1} ; If an admissible heuristic were used, this search would be guaranteed to find some λ_{i+1} whose π_{i+1} is of minimal SWAP count, however the authors also include a look-ahead heuristic that, given a candidate λ_{i+1} , estimates the size of π_{i+2} with the goal of finding maps that are both compliant with their layer and also near future maps. For λ_0 , the authors begin with with an empty map, and define an initial placement scheme that minimizes π_1 according to their look-ahead heuristic.

Finally, given the complete mapping \mathcal{M} , the permutation layers $(\pi_0, \dots, \pi_{n-1})$, having been previously identified by the A* search, are interleaved into the circuit. Of primary interest in this result are: the introduction of a look-ahead to balance local and global optima, and the unique decision of the authors to prioritize the distance sub-problem such that resulting maps are simple side-effects of the SWAP-minimization algorithm and not independently chosen maps that exhibit some desir-

able connectivity properties, in stark contrast to earlier efforts.

At this point in the chronology, methods have advanced to introduce layering, enabling significantly more complex circuits to be successfully mapped, have emphasized SWAP-minimization as a metric of interest for maximizing reliability, and have identified techniques to do so on realistic topologies. In the following period, multiple authors switched focus to alternative reliability measures. Tannu and Qureshi and Murali *et al.*, independently pioneered noise-adaptive approaches in early 2019 [33, 50]. In both works, the authors devise methods for optimizing algorithms solving connectivity, distance, or both problems by changing distance metrics from gate counts to formulas that actually take into account individual variations in qubit gate errors. Tannu and Qureshi describe two algorithms. Their “Variation-aware qubit movement algorithm” (VQM) proposes a solution to the distance problem by first calculating a distance matrix D , where element d_{ij} is defined as the distance between physical qubits i and j , and the distance between any two adjacent qubits is the probability of failure of a SWAP gate exchanging state between those qubits. Path weight is then the product of the respective weights of every edge on the path. This method is fundamentally similar to that proposed by Zulehner, with the exception of edge weights used by the minimum-path algorithm.

Second, they propose a “variation-aware qubit allocation algorithm” (VQA) for determining layouts. Assuming a circuit requiring k physical qubits, VQM begins by finding the Aggregate Node Strength (ANS) for all sub-graphs of size k . Let G be the coupling graph of the QC, w_{ij} represent the reliability of a CX gate applied on the edge (q_i, q_j) , and let $d_i = \sum_{k:(q_i, q_k) \in G} w_{ij}$ be the sum of the reliability of all edges coincident with qubit q_i , then $\text{ANS} = \sum_{i \in k} d_i$. Once a sub-graph with the greatest ANS is identified, virtual qubits are ordered by the number of CX gates they are involved in throughout program execution, and then the algorithm maps the most active qubits

to the most reliable sub-graph. These methods of VQA and VQM prioritize finding optimal allocations for each layer (i.e. solving the connectivity problem), and then find the best paths to permute between layouts *given* the chosen allocations; that is, although the transpiler operation is *variation-aware*, allocation is not movement-aware, unlike, e.g., Zulehner’s method. Additionally, Tannu and Qureshi do not take into account any other factors affecting SWAP count, like out-degree of involved qubits.

Murali *et al.* construct an allocation heuristic that is substantially similar to VQA, except that it begins with a sub-graph with $k = 1$, that is allocation begins with a single qubit with the highest ANS, and each subsequent qubit is either mapped to a qubit that shares an edge with a previously mapped qubit or, if none are available, to the next-highest ANS qubit [33]. Pathing is incidental and substantially resembles previous attempts at finding shortest paths by using an A* search heuristic.

2.6 Summary

This chapter begins with a description of the mathematical and computational models defining quantum computation in Section 2.2, defining and demonstrating the construction of Hilbert spaces and associated matrix transforms that can be implemented as quantum gates to perform qubit operations. Then Section 2.3 presents information about the specific implementation of quantum services provided by the IBM QX environment and Qiskit quantum programming library. The IBM QX Transpiler and its procedures are described in Section 2.4, showing how passes are defined and used to transform quantum circuits and enable them to execute on quantum hardware. Finally, the QLP is defined and previous work on finding effective and efficient methods to map virtual qubits to physical topologies is covered, including the limitations of existing solutions, in Section 2.5.

III. Methodology

3.1 Overview

This chapter defines and describes the methodology being used to analyze the research questions proposed in Chapter I. After a brief overview of the approach, the motivation for test bed development is introduced in Section 3.3. Subsequently, a discussion of relevant software design principles and requirements analysis is provided in Section 3.3.1 and Section 3.3.2, respectively. Section 3.4 then proposes a heuristic that builds on research identifying important characteristics of Quantum Layout Problem (QLP) solvers and defines its various components. Finally, the experimental configurations and benchmarks are proposed and described in Section 3.5. The results of both the software engineering and Quantum Layout Solver (QLS) analyses are provided in Chapter IV.

3.2 Approach

There are three primary goals to this effort. First, to identify the design principles and requirements for an effective and efficient testbed for QLP solvers. Second, to identify appropriate tradeoffs within and among the functional requirements and design principles and to implement this determination into the development process to create such a testbed. Finally, the third goal is to develop a proposed heuristic that overcomes weaknesses in existing techniques for solving instances of the QLP.

3.3 Test Bed Design Principles and Goals

Because research into QLP methods is in its relative infancy, there are significant and fundamental issues with early approaches. Researchers do not yet have standardized test sets, transpilation models, or metrics of success, and there is little to

no literature justifying even simple decisions made during experiments. It is clear that there does not yet exist a *body of knowledge* informing particular experiment methodologies. Instead, early efforts are scattered, independent forays into a space still mostly unexplored. As such, there is not only an opportunity, but a demand for foundational tools to transform existing, highly technical and specialized processes and codebases into robust toolchains that provide simple, functional interfaces to regulate and structure research in this nascent field. As quantum computers become more accessible—an explicit goal of the IBM QX project—demand for physicist and mathematician involvement will decrease and demand for computational and numeric expertise will grow. The paradigm shifts from one dominated by the concept of quantum computers as objects of study to one dominated by the concept of quantum computers as tools. Thus, long-term benefit can be generated by capitalizing on this shift and providing methods that researchers emphasizing *use* of Quantum Computers (QCs) can use while avoiding engaging with the complicated code and methods developed by and used by researchers emphasizing *development* of QCs.

3.3.1 Design Principles

Although there are many factors that distinguish computational science software from commercial software, the principles of software engineering are still broadly applicable. Software design must take into account: first, that the functional performance desired is achieved; second, that training or specialized skill requirements are minimized; third, that the developed system achieves required reliability; and finally, that the developed system promotes standardization [19].

Functional Performance: Of obvious and primary importance is that software should be designed to achieve its purpose. Regardless of what other design principles it adheres to, software that does not function is not useful. Functional performance

should be validated and verified against requirements generated by system stakeholders; however, this practice must also be understood in context. Scientific software is by its nature distinct in context because verification is not always possible [11]. Oracles or other truth-value generators may not exist or be feasible, and inappropriate system behavior can be due to design errors or algorithmic or computational errors. These causes may not be easy to disentangle and even if they are, algorithmic or computational correctness may require significant effort to implement—as the property of research dictates that proper methods are not always known ahead of time. The requirements analysis for the QLP test bed is the focus for Section 3.3.2.

Skill Requirement Minimization: Understanding and implementing this principle is closely tied to an understanding of software lifecycle expectations, extensibility, abstraction, and maintainability concerns. First, the developer must anticipate the lifecycle of the application, because this changes the expected return on training and subject matter expertise development. A system intended to be used over long time periods and to have regular maintenance and patch cycles yields increasing benefits to trained personnel since training is primarily a one-time cost event.

Conversely, six months of training and familiarity for an application with a year-long lifetime is wasteful and representative of poor design. However, highly extensible software necessarily makes use of abstraction and increased, underlying complexity to provide the necessary flexibility for easy extension—e.g. by providing interfaces, metaclasses, models and schemas, or abstract base classes that are then given concrete implementations. All of these concerns are then modified by the scientific context of this research effort, which affects both the kinds of specialized skill sets available to likely users and maintainers as well as obviating commercial cost-benefit analyses and making lifecycle determinations difficult at best.

Reliability: Software reliability, the assurance that functions behave as expected,

that displayed data is accurately representative of database contents, that the system is available for use as required, and that identical operations on identical states return identical results [42], is of heightened importance in a scientific computing context. Because, as mentioned, known-good outputs are not always predictable prior to program execution, reliable program interaction is absolutely critical to ensuring the integrity of generated scientific data. Hidden or obscure errors within experimental algorithms or procedures are potentially undetectable and outside of the scope of system design, but insofar as it is feasible, scientific software should both provide reliable behaviors and increased visibility into operations to enable rapid evaluation of data reliability.

Standardization: Finally, the standardization design principle is best thought of, for the purpose of this research, as concerning two distinct development strategies. First, let *external* standardization be defined as enforcing uniformity of interfaces or libraries, using naming conventions, documentation conventions, and style conventions that adhere to best practices or industry standards. This thread of standardization makes it easier to maintain software and find developers for it by allowing people to leverage existing knowledge and skillsets. It also allows other developers to more easily create systems that integrate the application or communicate with it and is broadly speaking the primary concern of *standardization* as it is commonly used for software development.

Comparatively, let *internal* standardization be enforced uniformity of application *content*. To some extent, this alternative mode of standardization overlaps with external standardization concerns—for example, in ensuring messages generated in an application follow a specific format that would both enforce content standards and also assist with interoperability with external systems. However, internal standardization of content is uniquely important to scientific computing for two reasons: first, in

ensuring that experimental setups can be reproduced reliably. If a scientific application enforces uniformity for experimental configuration parameters, then it improves reproducibility and also makes it easier to rapidly repeat and iterate on experimental design parameters. Second, internal standardization is also important to ensuring that results generated by experiments are comparable to one another. This form of standardization is an important consideration for scientific design in that it promotes rapid development of scientific knowledge by allowing cross-comparison and easy data sharing. It also relates to reliability, in that standardized information gathering makes the occurrence of anomalous results less likely and easier to detect.

3.3.2 Requirements Analysis

Requirements analysis for the QLP testbed is a functional design-driven process. The aforementioned design principles should guide development efforts, but only secondarily to the functional parameters that fill the capability gaps identified in Section 3.3. In particular, there are clearly three functional areas whose features ought to drive the development of the testbed: experimental configuration, circuit modification, and experimental results.

A functional testbed application for IBM QX quantum circuit experiments must first provide easy access to a variety of experimental configurations. Of foremost interest is, naturally, a method of uniformly creating and executing circuits. As currently implemented, circuit creation is not extremely difficult, but it is clumsy. Circuits must be created in sequential order, with no trivial methods provided for gate insertion. Additionally, the stochastic nature of some native Qiskit operations means that even identically created circuits may be transformed into distinct configurations at execution time (see Section 3.5.2 for more information). Sometimes this behavior is desirable, but frequently it is not, and there is again no accessible functionality to

control this variation solely with native Qiskit functions.

Second, presumably a researcher seeking to perform experiments on IBM QX architecture will desire to actually effect some change on the system—else there is no experiment occurring, merely standard execution. As such, any testbed system must provide a method for modifying system internals in order to test such modifications. This feature will likely be antagonistic to the design principle of training minimization, as by their nature experimental modifications occupy an immeasurably large space. Simple modification systems would be easier to use, but permit fewer modification options; conversely, complex modification systems permit much larger families of experiments at the cost of more user complexity. In line with most existing scientific Python libraries, like `scipy`, `NetworkX`, `numpy`, and `matplotlib`, this requirement is best defined in terms of maximizing flexibility at the cost of end user accessibility. Given the infeasibility of anticipating even a small number of potential proposed experiments, it is better for the testbed to expose more functionality and thus to enable greater experimental variation.

Finally, empirical research depends ultimately on generating and analyzing data. A key function point of a testbed system is the ability to reliably gather and process important data. Observationally, it appears that many research efforts, especially those performed by individuals, have scattered and various data gathering methodologies, including unstructured text files, inconsistent data fields, and poor data integrity. Ideally, a testbed would automate statistics collection to the maximum reasonable extent. By doing so, risks of misattributing data to test cases or configurations that did not generate it, misplacing data, failing to gather key information in a timely manner, or collecting similar data in dissimilar ways can be minimized. In particular, the testbed data collection method should emphasize reliability, traceability, and reproducibility. Sufficient data to allow experimentalists to know its source and the state

of external systems that were used and to recreate it, and the collection methodology should ensure that information cannot be silently changed or corrupted. As such, at minimum the proposed testbed application should store time and date information and execution and configuration information, so that even if some key data were missed, that data can be generated *de novo* from existing records.

3.4 Algorithm Overview

Characteristic of attempts made so far at optimizing SWAP and CX use on quantum circuits for IBM QX hardware is limited or no use of global information and a lack of communication between distance- and connectivity-emphasizing passes. Methods that emphasize selecting optimal layouts from the perspective of a given layer are at risk of identifying locally-optimal solutions that blindly generate a circuit structure requiring significant inter-layer SWAPs, as in Figure 7. Although qubits q_0 and q_1 are mapped to the most reliable CX connection in Figure 7b, later layers require that qubit q_0 have a SWAP gate applied to move its state to the center hardware qubit in order to accommodate entanglement requirements for subsequent layers. Depending on the specific reliability measures, the extra noise introduced by the SWAP operation may outweigh the benefit of performing the first CX across a less reliable edge by adopting a layout similar to Figure 7c immediately. Conversely, methods that use A* or similar methods across a configuration space whose distance is defined by SWAP operations find the most efficient path between layers, but may select layouts that have poorer reliability measurements or internal connectivity.

Additionally, although recent work has demonstrated the value of including dynamic reliability information in the mapping process, the tradeoff threshold between lower-reliability, high-connectivity layouts and high-reliability, low-connectivity layouts is unclear and likely depends on the particular application. Given the choice

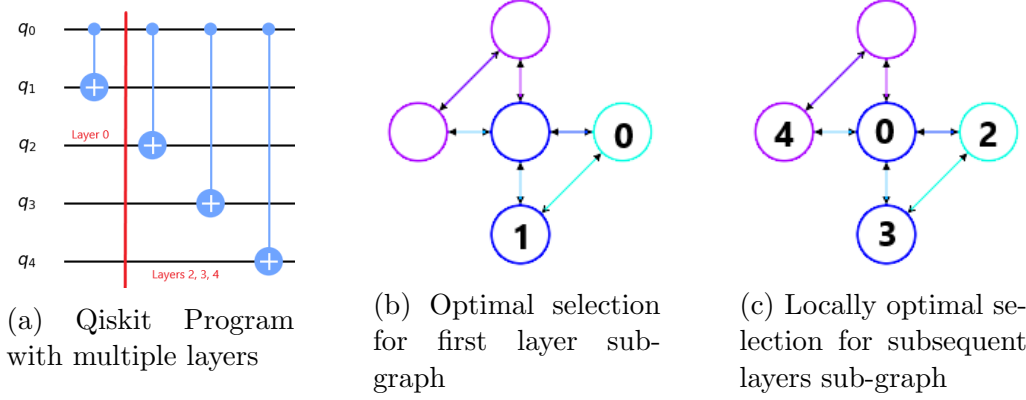


Figure 7: A circuit and its two locally optimal sub-graphs for layout in successive layers. Notice that the locally optimal selection results in many SWAP gates being required to move state between them. Brighter colors represent more reliable connections

between two otherwise equivalent layouts, connectivity is frequently still the predominant consideration; let λ_0 be a layout characterized by edge reliability p and satisfying layer ℓ_i with n entanglement constraints, and let λ_1 be a layout satisfying the same constraints with reliability q such that $p > q$, then λ_0 has a layer reliability of p^n , and λ_1 similarly has a layer reliability q^n [18]. If λ_1 requires k SWAP operations over edges with reliability r to setup, and λ_0 is coupled such that $k + 1$ SWAP operations are required to achieve its configuration, then including the inter-layer SWAP error rates, the overall reliabilities are $R(\lambda_0) = (r)^{3(k+1)}p^n$ and $R(\lambda_1) = (r)^{3k}q^n$. $R(\lambda_0) > R(\lambda_1)$ if and only if $p > \frac{q}{r^{\frac{3}{n}}}$, or equivalently $r^{\frac{3}{n}} > \frac{q}{p}$.

This is revealing for multiple reasons. First, and most obviously, it is clear that the factor of n makes selecting higher-reliability layouts increasingly valuable as the number of operations being performed in the layout increases; in short, the additional cost of reaching such a configuration can be amortized over all the entanglements—the longer you spend in a given layout, the more worthwhile it is to spend reliability to get there. Second, it lays bare the importance of the SWAP edge reliability to the decision-making involved. Although many algorithms include reliability metrics

for both selecting layouts and SWAP paths, none explicitly relate SWAP reliability costs to layout reliability as a decision-making component. For example, current noise-aware algorithms, faced with the decision between selecting a layout with an edge-reliability of 0.9 or taking a single additional SWAP over a high-reliability edge to instead achieve a layout with an edge reliability of 0.95 will uniformly select the latter option, but this actually reduces the overall reliability of the layer by $\approx 1.5\%$ unless there are at least three entanglement operations being applied in the layout. Understanding and applying these tradeoffs will become increasingly important as layer-size and travel distances increase, and in the current paradigm with relatively small qubit and entanglement counts, it's almost always true that the extra travel cost associated with achieving high-reliability layouts does not yield a net benefit.

Most important is that these initial considerations lay bare the necessity of circuit characterization to the layout process. That is, the most effective future heuristics for solving the QLP will likely need to engage with the properties of the circuits being mapped. For example, by classifying the circuits into families based on the distribution of entanglement constraints across layers or by the topology implied by the constraints (hub-and-spoke, mesh, pairwise, etc.) [49]. A thorough and structured treatment of characterization categories and methodologies is beyond the scope of this work, however as demonstrated previously, some understanding of the relationship between operator reliability and connectivity can be important and is relatively easy to implement as an initial step towards this paradigm.

3.4.1 Initialization and Pre-processing

In order to improve efficiency, a variety of non-circuit-specific information is first derived as a pre-processing step. In a full implementation of the QLP solver, these characteristics can be routinely calculated and made available by the service provider

independent of the transpilation and execution of circuits. Since they do not need to be re-calculated per execution, their cost can be amortized across all circuits executed over relatively long periods of time.

First, as part of a topological characterization, a *reduced coupling map* is created from the original coupling map. Given n nodes representing hardware qubits, mapped to integers in the interval $\llbracket 0, n - 1 \rrbracket$, then each edge present in the hardware can be defined by a tuple of the two nodes constituting the endpoints of that edge. The coupling map is then a list of all such edges in the topology. Each edge has an associated CX reliability measure, and a reduced coupling map can be constructed by removing from the coupling map all edges with a reliability less than the cube of the median reliability. This reduction has the effect of constraining the search space for optimal layouts in an efficient manner, and concordant with this, increasing the likelihood that the algorithm will find beneficial tradeoffs between layouts with additional SWAP requirements and layouts with fewer SWAPs but lower reliability sub-graphs.

Second, once the reduced coupling map is derived, Dijkstra’s algorithm is used to calculate the shortest-path information between all nodes. This information is stored in an adjacency matrix where each element M_{ij} contains the length of the shortest path connecting nodes i and j and a list of nodes on that path, along with the total reliability of a SWAP along it, calculated as the product of the cubes of the reliability of each edge on the path.

Third, the exogenous parameters p, γ are selected from the interval $[0, 1]$. p is used to select the weight—that is, preference—that each component score contributes to the final result. γ is a discount factor used by the distance metric to define the importance of satisfying future circuit constraints in the current layer. Both of these variables are discussed in more detail in their relevant sections.

Using the reduced coupling map, the given circuit, and p, γ , a series of candidate sub-graphs of the reduced coupling map is produced, each of which is potentially the target for the initial layout of the circuit. Each sub-graph is measured for suitability as mapping targets, and then associated with a proposed layout. These layouts are then respectively measured for their distance from future, suitable mappings—that is, a heuristic measure of how many SWAPs might eventually be required to create a fully compliant mapping from the initial layout. Finally, a combined score is calculated that identifies the initial layout most compatible with the prioritized connectivity and distance requirements explicated by the experimental variables. The intention is that this prioritization will reduce total errors over the execution lifetime of the circuit. An overview of this process is provided by Algorithm 1.

3.4.2 Connectivity Component

The connectivity heuristic provides a metric to measure the value of selecting one sub-graph over others. In particular, it is clear that the most generally optimal sub-graph to select to run a circuit requiring n qubits would be a sub-graph isomorphic to κ_n where each edge and qubit has reliability 1, since such a sub-graph can accommodate the largest variety of constraint topologies. Since this is generally not possible, sub-graphs are ranked according to their number of edges and subsequently to the product of the reliability of all their edges, which constitutes a heuristic for measuring overall sub-graph reliability. The heuristic takes as input a number of sub-graphs to explore and the size of sub-graph to return, and returns a list of candidate sub-graphs. The optimal solution to finding such a list is trivial but intractable: simply exploring the graph to find all connected sub-graphs of the appropriate size. Unfortunately, such an operation is \mathcal{NP} -hard [3] and therefore not a suitable technique for even relatively small problems.

Instead, the connectivity heuristic begins by picking likely seed nodes by identifying the most connected nodes in G' . Each seed node is then identified as an independent sub-graph and grown. This is done by exploring all neighbors of the sub-graph that are not currently members, and then selecting the neighbor that shares the most neighbors with nodes within the sub-graph, breaking ties on the reliability of the edges connecting the candidate to its neighbors. This process continues iteratively until the sub-graph has *size* nodes, and then the next seed node is selected to develop a new sub-graph. After each sub-graph has reached the appropriate size, it is associated with its connectivity measure, the ratio of the number of edges within the sub-graph to the number of edges in a complete sub-graph of the same size: $C_s = \frac{1}{edges(\kappa_n)} \sum_n edges(n)$. Finally, when all seed nodes have been consumed and the list of sub-graphs has size *stop*, this list is returned. This process is more rigorously defined in algorithm 2.

3.4.3 Distance Component

In order to improve the ability of the IBM transpiler to find more globally effective solutions, the final layout choice must be made based on more than a consideration of sub-graph quality, unlike current methods [33]. Towards that end, the list of sub-graphs and their connectivity measures returned by the connectivity component of the QLS is then passed to a distance measurement routine which additionally accepts the quantum circuit to be executed and a discount factor *gamma*. First, making use of the standard layering pass in the IBM transpiler, the provided circuit is separated into a sequence of layers, $L = \{\ell_0, \dots, \ell_n\}$. Each layer contains a set of gates that can be applied simultaneously; equivalently, each layer contains as many gates as possible where each gate has no data dependency on the execution of any other gate in the layer. Each $\ell_i \in L$ is then fed into a function that derives a constraint graph c_i . This

Algorithm 1 Find Best Connected Sub-Graph

1. **Parameters:** $size$ – size of desired sub-graph
 G – Coupling map
 2. Create a reduced coupling map, G' by removing all edges whose reliabilities are less than the cube of the median reliability.
 3. Using Dijkstra's algorithm on G' , create an adjacency matrix M , each of whose elements m_{ij} contains a tuple $(distance, reliability, path)$. Where $distance$ is defined as the number of SWAPs required to move state from qubit i to qubit j ; $reliability$ is the product of the reliabilities of all edges on the most reliable path between i and j ; and $path$ is a dictionary containing the nodes connecting i and j .
 4. Let w_i be the weight of qubit i , where the weight is defined as the number of edges incident to node i in G . Create a sorted, descending list, W whose sort key is this weight function.
 5. (a), if there exists at least one qubit i such that $w_i > size$, then select the most reliable sub-graph, g centered on a qubit whose weight is maximal.
 6. (b) if there exists no such qubit, then select the most connected sub-graph of size $size$ using the procedure defined in algorithm 2
 7. The output of algorithm 2 is then fed into algorithm 3, this returns a list of explored sub-graphs, each associated with their connectivity score, C_s – that is, a measure of how complete the sub-graph is – and their distance score, D_s – a measure of how near the initial sub-graph is to fully satisfying the constraint requirements of all layers.
 8. Each sub-graph s is then given a final ranking $R_s = pC_s + (1 - p)D_s$, with p provided as an experimental input to weight the scores.
 9. The sub-graph with the highest R is then fed to the transpiler as the initial layout.
 10. Finally, all subsequent layouts for each circuit layer are generated by the transpiler according to the default, IBM shortest path SWAP algorithm.
-

Algorithm 2 Connected Sub-graphs

Parameters: *stop* – number of sub-graphs to explore
size – The desired number of nodes each returned sub-graph should have

```
1: subgraphs = {} ▷ an empty dictionary
2: while i  $\leftarrow$  0; i + +; i < stop do
3:   subgraph  $\leftarrow$  []
4:   while len(subgraph) < size do
5:     max_weight_node  $\leftarrow$  W[i]
6:     all_neighbors  $\leftarrow$  max_weight_node.neighbors()
7:     max_child_weight  $\leftarrow$  0
8:     candidate  $\leftarrow$  None
9:     for node in all_neighbors do
10:      if len(all_neighbors  $\cap$  node.neighbors()) > max_child_weight then
11:        max_child_weight  $\leftarrow$  len(all_neighbors  $\cap$  node.neighbors())
12:        candidate  $\leftarrow$  node
13:      end if
14:    end for
15:    subgraph.append(candidate)
16:    all_neighbors.append(candidate.neighbors)
17:  end while
18:  connectivity  $\leftarrow$   $\frac{\text{len}(\text{subgraph.edges()})}{\kappa_{size}}$  ▷ ratio of edges to complete graph of size nodes
19:  subgraphs[subgraph]  $\leftarrow$  (connectivity, inf) ▷ inf represents the unknown distance factor
20: end while
```

graph contains a node for each qubit involved in an entanglement operation in that layer, and an edge connects two nodes when they have a control-target relationship in that layer. This results in a sequence of constraint graphs $C = c_0, \dots, c_n$.

A nested loop is then executed wherein each sub-graph in the *subgraphs* list is given a layout. This is accomplished by leveraging existing transpiler routines that make use of the *greedyE** layout pass provided by Murali *et.al.* [33] and included in the Qiskit library. This layout pass takes the provided sub-graph, and maps the virtual qubits of the circuit to them by selecting the virtual qubit with the most entanglement constraints across circuit execution and mapping it to the most reliable qubit in the sub-graph. For each subsequent virtual qubit, if an entanglement partner has already been mapped then the virtual qubit is mapped to the most reliable shared edge; if the virtual qubit has no entanglement partners already mapped, then it is mapped to the most reliable, free qubit. In this manner, each sub-graph is associated with a single layout that maps each virtual qubit in the circuit to one node in the sub-graph.

Subsequently, each layout is assessed against each constraint in C , returning the number of constraints met by the provided layout, per layer: let $M()$ represent a function that counts the number of met constraints, then $m = M(\text{subgraph}, c_i)$. This results in each candidate sub-graph being associated with a series of m_i values, corresponding with the number of constraints met per layer by that sub-graph. This set of m_i values is finally converted into a single score by normalizing against the total number of constraints the circuit requires be met in each layer, i.e. the number of CX gates present, then discounting by the factor γ^i , and finally summing the resulting series and associating each sub-graph with this distance score. Algorithm 3 represents this process in pseudocode.

Algorithm 3 Sub-graph Coupling Distance

Parameters: *subgraphs* – A list of sub-graphs of G'

circuit – Circuit to be executed on a sub-graph.

gamma – Discount factor to reduce the weight of the subsequent layer constraints.

```
1: layers  $\leftarrow$  layer(circuit) ▷ use standard circuit layering
2: constraints  $\leftarrow$  []
3: distance_score  $\leftarrow$  []
4: for layer in layers do
5:   constraints.append(layer.get_cx_constraints()) ▷ Stores CX constraints per layer
6: end for
7: for subgraph in subgraphs do
8:   for constraint in constraints do
9:     layer_score  $\leftarrow$  met_constraints(subgraph, constraint)
10:    distance_score  $\leftarrow$  distance_score +  $\gamma^i * \text{layer\_score}$ 
11:   end for
12:   subgraphs[subgraph][1]  $\leftarrow$  distance_score ▷ Replaces inf
13: end for
```

3.5 Benchmarks and Evaluation

The evaluation of the Quantum Layout Problem Testbed (QLP-TB) design naturally assesses the performance of the library against the defined requirements. Thus, the verification procedure must assess the QLP-TB’s capability to enable rapid creation, modification, and evaluation of quantum circuit experiments in an efficient and effective manner.

A subset of the available test circuits provided within the library have been selected to exhibit a range of behavior. Then a series of transpiler configurations are constructed using distinct algorithms to determine circuit layouts and SWAP paths. Each of the circuit and configuration combinations is then measured against the performance of the IBM baseline transpiler configuration in terms of the euclidean distance separating each resulting measurements distribution from an ideal variant, in terms of additional SWAP gates induced by the transpiler configuration, and in

terms of the time taken by each transpiler configuration to transpile each circuit. Verification is confirmed provided the QLP-TB demonstrates the ability to meet the defined functional requirements, and that the experimental test script used for execution demonstrates efficiency gains relative to a similar experiment performed absent the QLP-TB capabilities.

The circuits and transpiler modifications relevant to the QLP-TB verification procedure are detailed below, and a complete listing of circuits and their descriptions provided by the QLP-TB can be found in [Appendix A](#).

3.5.1 Evaluation Circuits

From a high-level view, the ultimate goal of improving layout selection is to broaden the scope of successful quantum circuit execution. That is, to enable circuits requiring more qubits or greater number of gates to generate useful results. Towards this end, benchmarks circuits have been selected with a diversity of behaviors in mind to ensure tests characterize a variety of constraint topologies that might naturally arise as sub-circuits. As circuits grow larger, the probability they succeed falls precipitously, and SWAP-induced error is only one cause. Additionally, circuits may experience time-based decoherence [\[34\]](#), environmentally induced decoherence [\[39\]](#), accrue errors from imprecise gate calibration [\[33\]](#), and generally experience as of yet uncharacterized errors [\[18\]](#). As such, the benchmark circuits have been kept relatively small in order to minimize the effects of unrelated errors on the results [\[50\]](#). Additionally, the test set is closely related to test sets used by previous, related efforts in that small algorithms with known truth values are used—like Grover’s algorithm and the Bernstein-Vazirani algorithm—and a few circuit primitives that are likely to be useful components of larger, complex algorithms are also included [\[33, 50, 53, 54\]](#). A brief description of the test set is provided, summarized by [Table 2](#).

Two Bell: The two-bell circuit creates random pairs of bell states across the width of the circuit, repeatedly until the final circuit is square. In particular, it operates only on sub-graphs with an even size and randomly permutes the available qubits. Given this permutation, a Bell state is formed from each consecutive pair of qubits in the permutation. Then the process repeats, with a new permutation, until the circuit depth is equal to the number of Bell state pairs formed in each layer.

Grover: A full explanation of Grover’s algorithm can be found in [14], but in summary it is a circuit designed to perform an efficient search on an unsorted list. Beyond the fact that it is a practical quantum circuit where improved error performance can have real world consequence, it is included as a test circuit primarily because of the entanglement constraints it requires. Grover’s algorithm requires multi-controlled entanglement gates—that is, entanglements of more than 2 qubits simultaneously. Since the IBM QX architecture does not directly implement entanglements on > 2 qubits, the derived circuits are extremely SWAP intensive. Since the SWAP counts are not particularly tied to topological or layering concerns, it is unlikely the proposed QLS will show significant improvement over existing methods. However, doing so would be particularly valuable.

Uniform Random: Creates a circuit whose gates are uniformly chosen from H, X, Y, Z, S, T, CX and whose CX endpoints are chosen uniformly from available qubits. The loop is iterated until the circuit is square.

Bernstein-Vazirani: This test circuit implements the Bernstein-Vazirani algorithm [2]. An integer in the interval $0, size$ is selected as a truth value for the circuit. This truth value is encoded as binary representation into an oracle sub-circuit using CX gates. Then, the remainder of the circuit creates a uniform superposition to be fed into the oracle and the output is read into another uniform superposition. This

algorithm is designed to permit recovery of the encoded truth value in constant time, while classical equivalents require *size* queries. The use of CX gates to encode the oracle and the fact that the truth value is well-defined and easily measurable make this a valuable contribution to the test set.

Quantum Fourier Transform: Finally, the Quantum Fourier Transform (QFT) is implemented as another practical circuit often implemented as a sub-circuit to algorithms [43]. The QFT is generally used to recover phase information from a given quantum state provided as input, and more generally serves to provide a structured basis transform: Let $|x\rangle = \sum_{i=0}^{N-1} x_i |i\rangle$ be the original basis and $|y\rangle = \sum_{i=0}^{N-1} y_i |i\rangle$ be the desired output basis of the transform. Then under a QFT each coefficient y_k for $k \in \llbracket 0, N-1 \rrbracket$ is defined as $y_k = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} x_i \omega_N^{ki}$, where ω_N is the N^{th} root of unity [6]. This can also be thought of as the quantum analogue to the classical inverse Fourier transform. Additionally, since the key operator used by the QFT is a controlled phase shift—implemented by IBM using a CX gate—then there is significant opportunity for improving implementation with the QLS.

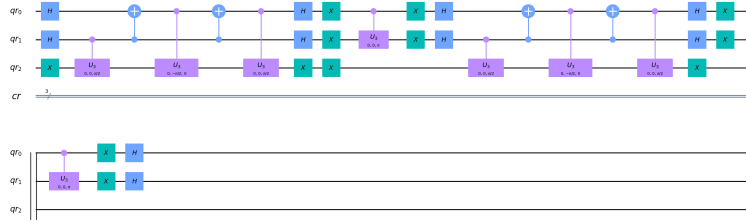


Figure 8: Grover’s Algorithm implementation in the test set, with a search space size of 8 and a truth value of 3—separated from Tables 2 and 3 due to length

3.5.2 Evaluation Transpiler Configurations

Four distinct transpiler configurations were chosen for the verification procedure. Two of these configurations represent the IBM Qiskit baseline transpiler configurations: one that provides the default pass set created if users do not specify otherwise,

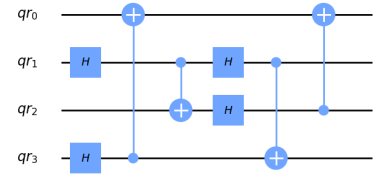
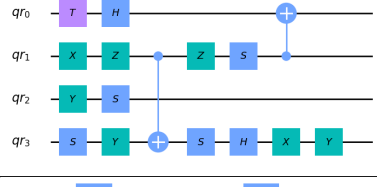
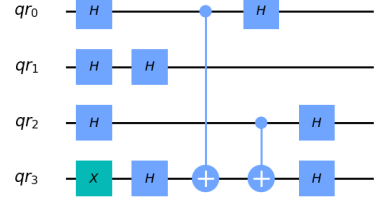
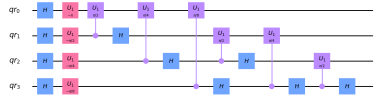
Circuit Name	Function Name	Example Implementation
Two Bell	two_bell	
Uniform Random	uniform_random	
Bernstein-Vazirani	bv	
QFT	qft	
Grover's	grover	See fig. 8

Table 2: Summary of test set circuits for evaluating QLS performance

and one that represents the Qiskit-defined optimal pass set—where optimal is understood to mean most likely to generate efficient circuits, but not necessarily the most time-efficient. The other two configurations are modifications to the baseline pass set and are included both to exploit QLP-TB features for verification purposes and also because they reproduce efforts by previous researchers to optimize IBM QX transpilation operations.

1. **IBM Baseline Configuration:** The IBM baseline configuration is automatically used unless the user explicitly requests a distinct PassMaanger configuration. The complete list of passes implemented in this configuration can be found below in Figure 9, though an explanation of most is beyond the scope of this effort. Of particular note, however, are the **TrivialLayout** and **Stochastic-Swap** passes, as they provide connectivity and distance metrics, respectively,

as identified in Section 2.5.

The **TrivialLayout** pass is intended to identify the correct hardware sub-graph to map a given circuit’s virtual qubits onto. In the case of **TrivialLayout**, the procedure is simple: each virtual qubit is mapped to the hardware qubit with the same index. For example, given a quantum circuit defined on qubits q_0, q_1, \dots, q_{n-1} , the Virtual-to-Physical (V2P) mapping is defined by **TrivialLayout** as $q_0 : 0, q_1 : 1, \dots, q_{n-1} : n - 1$. Clearly, this process although quite efficient does not take into account connectivity, CX weight, CX constraints, reliability, or other desirable sub-graph properties.

The **StochasticSwap** pass is intended to identify the best SWAP paths, but does so using random selection. After generating a random seed and a pre-processing step that collects information about edges in the backend coupling map, computes a distance matrix, and gathers the gates of the circuits, the **StochasticSwap** pass iterates through edges and randomly permutes layouts on those edges, calculates if the resulting layout results in a lower cost than the best candidate layout—where the cost is defined by the number of SWAP gates required to reach the layout from the current layout—and if so, sets it and begins iteration anew. If no successful SWAP path is found after a set number of iterations, the procedure fails.

2. **Lookahead SWAP**: This transpiler configuration is based on the IBM baseline but exchanges the **StochasticSwap** pass for a **LookaheadSwap** pass. This configuration was chosen because it implements the proposal of the IBM Qiskit Developer Challenge winner for transpiler optimization routines [22]. Instead of stochastically searching for compliant layouts, the pass instead performs a narrowed breadth-first search. Given an existing mapping and a list of upcoming CX gate constraints, the pass finds the top four individual SWAP gates which

each result in a layout that minimizes the total distance between the qubits in that mapping and the CX constraints. Any constraint that is met by the candidate mapping is removed from its list of upcoming CX constraints. This process is then iterated on four items, with each iteration taking the candidate mappings and reduced constraint lists from the previous iteration and again returning the four best, single layouts. Once a total of 256 mappings have been generated then the SWAP path resulting in the best final layout is selected.

3. **Noise-Adaptive Layout:** This transpiler configuration is based on the IBM baseline, but exchanges the **TrivialLayout** pass for the **NoiseAdaptiveLayout** pass. Whereas the **TrivialLayout** pass simply maps matching indices, the **NoiseAdaptiveLayout** pass operates by finding the densest sub-graph of appropriate size to layout the given circuit on. In particular, the procedure performs a breadth-first search beginning at every node in the backend topology and for a circuit of size n , inspects the first n nodes in each search graph. Each node is scored by how many other nodes in the first n of each search graph are also in the first n nodes of the search graph starting with that node. Once the connected collection of n nodes with the highest scores are discovered, a V2P mapping is created by sorting each hardware node by its total edges and then by edge reliability, and each virtual qubit is sorted by the number of CX gates it is involved in. Finally, the mapping is defined on matching indices in the two sorted lists.

This transpiler configuration was chosen both because it differs from the baseline in exactly the connectivity algorithm used and because it is the implementation proposed by Murali *et al.* [33]

4. **IBM Optimal Configuration:** similar to the IBM Baseline configuration, this transpiler configuration is a default provided natively in Qiskit. IBM defines

four “optimization levels” that aggregate passes into standard PassManagers, and whereas the IBM Baseline is the default level one, the IBM optimal configuration uses optimization level three—the highest available. Although this configuration is not necessarily reflective of the most cutting edge techniques proposed, it is representative of the pass set a researcher could naturally and easily access as a generic “best effort” configuration that does not require significant tailoring. Although this configuration uses the same SWAP and layout passes as the Noise Adaptive Layout, it includes other gate optimizations that are intended to provide an efficient and effective PassManager. The specific passes implemented in this configuration are located below in Figure 9.

IBM Baseline Passes (Optimization Level 1)	IBM Optimal Passes (Optimization Level 3)
SetLayout	Unroller
TrivialLayout	SetLayout
CheckMap	NoiseAdaptiveLayout
FullAncillaAllocation	FullAncillaAllocation
EnlargeWithAncilla	EnlargeWithAncilla
ApplyLayout	ApplyLayout
Unroller	CheckMap
CheckMap	BarrierBeforeFinalMeasurements
BarrierBeforeFinalMeasurements	Unroll3qOrMore
Unroll3qOrMore	StochasticSwap
StochasticSwap	Decompose
Decompose	Depth
RemoveResetInZeroState	FixedPoint
Depth	RemoveResetInZeroState
FixedPoint	Collect2qBlocks
Optimize1qGates	ConsolidateBlocks
CXCancellation	Unroller
	Optimize1qGates
	CommutativeCancellation
	OptimizeSwapBeforeMeasure
	RemoveDiagonalGatesBeforeMeasure

Figure 9: Full pass sets for the IBM pre-populated PassManager configurations

3.6 Summary

After brief coverage of the general approach, this chapter defined the design principles and goals of an effective QLP-TB development project, based on human engineering principles defined in [19], and presented in Section 3.3.1. A functional requirements analysis was presented in Section 3.3.2 that identified three key function points necessary for the QLP-TB. Next, an algorithmic overview was presented identifying the limitations and inefficiencies of existing approaches and proposing solutions to them, in Section 3.4. Finally, the set of test circuits and transpiler configurations constituting the verification testing procedure for the QLP-TB are presented in Section 3.5.

IV. Results and Analysis

4.1 Overview

The research results and data analysis are presented in this chapter. The results are separated into four components: the first research question is answered in Section 4.2.1 with an analyses of the features and qualities of application development and scientific computing. The second research question is approached in Section 4.2.2, where a concrete testbed implementation is presented and implementation features are traced directly to the requirements analysis presented in section 3.3.2. The results of full-scale verification test are presented and analyzed in Section 4.2.3. Finally, Section 4.3 analyzes the last research question and a brief treatment is given in pursuit of avenues for future research.

4.2 Quantum Layout Problem Testbed (QLP-TB) Design Implementation

Based on the requirements identified in Section 3.3.2: experiment configuration, experiment modification, and data controls—the final QLP-TB implementation sought to balance basic design principles with the unique execution environment of scientific computing.

The final environment is packaged as a Python library, which provides portability at the cost of usability. Although most applications are self-contained and have accessible user interfaces, the demands of scientific computing for significant access to and modification of system details makes such an effort unreliable at best. Similar tools, like Matlab, Mathematica, and scipy are similarly situated [8]. In all cases, the tools are designed and optimized for environments where users have and will seek to use scripting experience to formulate and resolve research problems.

The QLP-TB package consists of a variety of Python modules, each of which provide a set of functionality related to the identified requirements. In summary, the main `run_experiment` module functions as a driver class, where users can script their experiment and call or coordinate all library operations. All functionality is alternatively available when accessed from an external module importing the QLP-TB. From this central scripting location, users have access to `circuits` – which provides access to circuit creation and initial experimental setup, `transpilertools` – which exposes transpiler functionality from Qiskit, and also provides helper and wrapper functions to easy configuration changes, `dbconfig` – which provides functions to regulate and structure read and write operations from a variety of database tables storing experimental data, and `statblock` – which allows easy modification of statistics gathered during experiments.

By using the provided functionality, an end user can quickly script a complete experiment from initialization and test-set definition, to transpiler configuration modification, and finally data storage and retrieval. More details of each component are provided below at Section 4.2.2, following an analysis of the relevant design principles and tradeoffs chosen to accommodate them. The code snippets referenced here can be found at Appendix C.

4.2.1 Design Principle Analysis and Results

1. **Skill Requirement Minimization:** The most relevant feature of development in this research environment is that IBM Qiskit is in alpha phase. Alpha development is characterized by rapid feature and interface changes, and inconsistent behavior across patch cycles. As such, planning for an extended lifecycle would incur the associated costs of more robust development but be unlikely to return reasonable dividends for the invested effort. Indeed, the native Qiskit

transpiler interface underwent major changes during the QLP-TB development process. Bolstering this conclusion is the fact that scientific computing, as opposed to commercial applications, often does not provide maintenance cycles or the opportunity to hire or use dedicated developers.

Both of these conclusions drive the determination that the QLP-TB is best implemented concretely and with minimal abstraction, since the flexibility provided by extensive abstraction would likely still be insufficient to survive major Qiskit codebase changes and because the skillset to make use of such abstraction cannot be relied upon to be available. Fortuitously, these conclusions also harmonize with the design goal of minimizing required skills, as the simplified codebase and shorter lifecycle promote simple use cases. Additionally, the QLP-TB is written to use an industry standard documentation format, extensive use is made of the Python `Annotations` module, and Pep8 style recommendations are enforced. Cumulatively, these factors make the software significantly more accessible to developers and permit deep introspection by Pylint into program behavior. These features also make function parameter information, docstrings, and return types accessible to most IDE interfaces, speeding the acquisition of application information by users. It is important to note, however, that the skillset required to design and evaluate experiments is a distinct, academic concern and unrelated to the design goal of minimizing the skills required to use the software.

2. **Reliability:** Reliability is perhaps the most critical design principle for QLP-TB implementation, but also the simplest to enforce with existing best practices. First, reliable program interaction is best guaranteed in this context with strong use of Python error catching and raising constructs. Such use provides clear error traceability and ensures errors cannot propagate beyond their point

of initiation and especially cannot do so silently. Second, data reliability is enforced with a well-developed logging interface that provides constant visibility into program execution throughout its lifetime and additionally promotes traceability by ensuring every module is provided its own, unique entry point to the common logging system. In service of these features, the library also implements standard verbosity flags, enabling users to raise the logging and reporting level at each execution, increasing visibility when required. Finally, the data storage mechanisms use database best practices, controlling access with context managers that regulate transactions and permit automatic rollback in the event of a data integrity failure.

3. **Standardization:** External standardization was determined to be of minimal value to the QLP-TB. Very few system even enable quantum computation, and quantum computing systems from competing companies do not yet implement any universal interface or enable cross-communication. The components of external standardization relating to code style and documentation conventions were implemented, assisting any experienced developer with extending functionality or interfacing with the QLP-TB, but public-facing APIs were not considered valuable for the current effort. Internal standardization was, conversely, determined to be another critical design component. Towards this end, it was concluded that a common interface must be provided for circuit creation. This allows the user to provide a relatively small number of configuration options to generate a wide variety of test circuits, which promotes a uniformity of test design and allows results to be compared more easily.

Additionally, circuits are saved both prior to and following transpilation operations, which permits a researcher to rigorously identify and re-use stochastically-constructed transpiled circuits. Additionally, transpilation configuration ob-

jects are saved alongside their circuits, permitting researchers to recreate the state of the IBM Qiskit backend being targeted by the transpiler at arbitrary points in the future. Finally, although experiments may require gathering specific and distinct data to accomplish a specific mission, a broad selection of circuit data—including a serialization of the objects themselves—is stored in non-volatile memory and updated after every circuit operation, minimizing the need of users to design bespoke data collection methods.

4.2.2 QLP-TB Requirements Results

The requirements analysis provided in Section 3.3.2 resulted in the development of the QLP-TB, and conclusions about the best tradeoffs between standard design principles, functional requirements, the nature of quantum development efforts and Qiskit, and the limitations and peculiarities of development in a scientific computing context are represented by and instantiated within the codebase.

1. **Experimental Configuration:** Conclusions about the functional implementation of the experimental configuration requirement are embodied primarily in the `PreMades` and `TestCircuit` classes of the `circuits` module.
 - (a) **TestCircuit:** The `TestCircuit` class is designed to wrap circuit creation functionality to prevent unnecessary exposure of circuit creation functionality and assist with rapid experimental configuration. The `TestCircuit` constructor (referenced in Python as the `__init__()` method) shown in Listing C.1 demonstrates the core results of this goal. `TestCircuit` objects automatically instantiate uniform statistics gathering via a `statblock` member, associate themselves with a `Premades` quantum circuit object, and are capable of holding additional information about the particular execution environment the `TestCircuit` is intended for. Since `TestCircuit` instantia-

tion does not require a `Premades` quantum circuit object, then the user can provide configuration information and rapidly generate distinct `TestCircuit` objects without binding them to a particular algorithm or circuit.

The `TestCircuit` class also provides simple wrapper methods to run transpilation tests and gather statistics on SWAP insertions performed by the transpiler and automatic transpile timing measurements in Listing C.2 and similarly provide actual execution testing in Listing C.3 which additionally uses the provided `TestCircuit` fields to fully transpile and execute families of `TestCircuit` objects. This implementation, alongside the `Premades` class, fully adheres to the experimental configuration requirement by providing quick configuration and simple utility functions which prevent end user exposure to system internals while enabling experiments to be intelligently set up and executed.

- (b) **Premades:** In tandem with the `TestCircuit` construct, `Premades` are subtypes of the native Qiskit `QuantumCircuit` class, intended to provide easy access to test sets for experimentation. The constructor shown in Listing C.4 demonstrates that `Premades` objects are relatively simple `QuantumCircuit` objects with the addition of size, truth value, and measurement fields. These fields provide a simple, universal interface for circuit creation. Although not all circuits have a known truth value, those that do encode some particular basis state as the correct result of execution all can automatically do so with the given truth value. This means a user can provide 3 simple parameters and generate a variety of distinct circuits for testing purposes.

The class is also easily extensible, as method interfaces are designed to accept native `QuantumCircuits`, which permits end users to design and use any custom circuit as part of the testing process or they can quickly

add a new `Premades` circuit by simply defining it and modifying a single circuit library. More details of the particular test circuit code can be found at Appendix C and motivation for test circuit selection can be found at Section 3.5.

2. **Circuit Modification:** The circuit modification requirement is implemented throughout the QLP-TB, but is primarily identifiable in the `circuits` module previously discussed and in the `transpiler tools` module. Because circuit modification possibilities are endless, constrained selection of modification options was determined to be a poor choice. Instead, the conclusion was that users are best served by controlled, simplified exposure to system internals that accomplished two tasks: first, it provided a relatively small number of available modifications that could be predicted and made substantially easier than through the native Qiskit interface; second, it also allowed researchers who needed it the option of performing arbitrary, but complex, modifications and then passing the results of those modifications into the existing QLP-TB system.

(a) **circuits:** The task of permitting arbitrary modification is enabled by development decisions in both classes of the `circuits` module. Testing interfaces make extensive use of default parameters, so that users desiring simple functionality can easily and smoothly access it, but researchers are still able to independently generate a variety of modifications but use them within the QLP-TB environment. For example, although the ability to modify transpiler configuration options is provided by the `transpiler tools` \leftrightarrow module, there is no reasonable way to provide easy access methods to all potential transformations. Instead, by exposing the transpiler configuration field of the `TestCircuit` class, but also automatically populating this field as necessary, the QLP-TB permits users who do not desire such

modification to ignore the feature, but also permits researchers to separately generate any transpiler configuration necessary and provide it to the testbed for further automation.

Similarly, functions in the `circuits` module automate compiled circuit generation, but users are free to compile circuits separately—even by use of independent compilers as in [33]—and then pass the pre-compiled circuits to the testbed. Users can then set testing parameters to use these compiled circuits instead of requiring the tools to compile their own copies.

- (b) **transpilertools**: In contrast with the paradigm described in the `circuits` module, the `transpilertools` module of unbound functions is intended to predict the most commonly sought modifications and provide constrained, but simple access to them and avoid exposing Qiskit internals. Given the primary functional purpose of the QLP-TB is to enable transpiler modification testing, focusing on simplifying transpiler transforms was the natural choice.

In particular, functions are provided to easily create a family of `TestCircuit` \leftrightarrow objects to individual transpiler configurations (Listing C.5), to quickly access Qiskit pre-populated `PassManager` objects (Listing C.6), and to make simple modifications to existing `PassManager` objects by automating replacement of layout and swap `Pass` objects (Listing C.7). Although the set of available modifications is relatively small, it is well tailored to the primary purpose of the QLP-TB and in conjunction with the exposed interface design of the `circuits` module it provides a diverse set of behaviors to accommodate user goals regardless of complexity.

3. **Experimental Results**: The experimental results requirement was the most conducive to being solved with existing, best practices widely implemented

across industry software. Given the breadth of data to be gathered, functionality exists throughout the QLP-TB that assists with meeting this requirement, however all such functionality stems from the interaction of two primary modules of the testbed: the `statblock` module and the `dbconfig` module. Essentially, reliable data gathering and processing was determined to depend on two capabilities that were implemented: first, insight must be provided into the data actually generated by the testbed, this was the domain of the `statblock` module and class; second, the data gathered should be reliably and automatically structured and stored, and this capability was provided by the `dbconfig` module and class.

- (a) **statblock:** The `statblock` class, contained in the module of the same name, provides a consistent data structure that can be instantiated as an object and then attached to all `TestCircuits`. By doing so, statistics gathered throughout testbed execution can be gathered and stored in a consistent way. In particular, this design decision ensures each `statblock` is always in the relevant scope during `TestCircuit` execution, but also permits easy extensible modification of statistics gathered. Any new statistic can be introduced in a single location—the `statblock` constructor—and then is automatically available for use throughout the testbed. The existing constructor is shown in Listing C.8 and currently is designed to store data of clear importance for general experimentation (e.g. date and time of testing) and also data values of particular note for Quantum Layout Problem (QLP) testing in particular (e.g. transpiler-induced SWAP gate count).

Methods and functions throughout the QLP-TB automatically add data to the `statblock` as it is generated, ensuring users are not required to define or remember information to collect on an *ad hoc* basis. The stat-

block also generates a UUID allowing every generated `TestCircuit` to be uniquely associated with its statistics. Although the UUID4 generation procedure is not guaranteed unique, the time-based PRNG functionality makes collision exceedingly unlikely, severely mitigating the risk of data misattribution [26].

- (b) **dbconfig**: While the `statblock` associated with each `TestCircuit` provides the capability for every test, function, or method to ensure updated and consistent data is saved, this data is associated only with the object involved and is not automatically available in a human-readable format. The `dbconfig` class, in the module of the same name, provides controlled access to non-volatile storage and retrieval of objects and their associated statistics. The `Circs` table stores the unique ID of every object and its associated serialization, making it possible for any researcher or user to recreate objects and retrieve all their properties, including transpiler configuration, compiled and uncompiled circuit variants, and the backend and its properties associated with the `TestCircuit` at the time of creation or execution, making reproduction of experimental results trivial, rapid, and reliable.

In service of this requirement, `dbconfig` provides insertion, update, delete, and read operations to its tables. Each method uses context managers, error catching, and a transactional paradigm to ensure data integrity is maintained at all times. Additionally, the `dbconfig` class also provides a method for retrieving stored objects and their associated statistics to store into a `stats` table that makes the `statblock` information available in a human-readable format and permits easy export of `TestCircuit` data csv or similar formats for use in external applications. Finally, the `Running` table is used by the driver module `run_experiment` to track running ex-

periments and ensure results are not lost even under application restart or crash, as automated functionality is provided to retrieve running jobs, associate them with the original `TestCircuits` the execution was called on, and update the statistics generated by IBM QX execution.

4.2.3 Verification Results

QLP-TB functional capabilities were discussed and fully traced back to requirements and design principle constraints in Section 4.2.2 and Section 4.2.1, respectively. Those results confirm that the design and implementation of the QLP-TB conform to the stated standards. However, the verification procedure is additionally included to confirm the effectiveness and efficiency of the QLP-TB at performing the practical task of experimentation. In particular, the verification is intended to assess whether an experiment in the scope of QLP testing can be *easily* constructed, modifications to existing native Qiskit functionality introduced, and results reliably gathered. Adhering to the testing procedure identified in Section 4.2.3, the experiment was defined in four, simple phases. These phases are described below, and the full experimental configuration script is available in Appendix B.

1. **Phase 1:** Initial circuit creation. Leveraging the capabilities of the circuits module, ≈ 26 lines of code are used to define the common circuit generation interface, create 50 copies each of five distinct circuits exhibiting diverse behavior, and finally register those circuits with the statistics and database modules.
2. **Phase 2:** PassManager creation and Modification. In phase two, the experimental modification requirement is verified by requiring four distinct modifications to native Qiskit transpiler configurations to test both connectivity and distance algorithms against existing baselines. In ≈ 13 lines of code, making significant use of the `transpilerutils` module, a transpiler configuration is created

for each of the 1000 eventual executions, that configuration is used to generate a pre-populated `PassManager` object for each execution, and four groups of 250 circuits each have a unique modification to their `PassManagers` applied that reflect the identified transpiler configuration test cases.

3. **Phase 3:** Execute Tests. The actual execution phase requires only four lines of code, primarily due to the availability of automated testing routines in the `TestCircuits` class.
4. **Phase 4:** Data Collection. Finally, data concerning initial circuit construction parameters and statistics that can be derived prior to actual execution have been gathered throughout all three preceding phases, but the final results generated by measurements after hardware execution on quantum systems must be gathered to complete the process. Because of the automated execution checking routine that stores information about circuits pending execution at the IBM QX backend and that automatically checks and retrieves completed jobs as they become available, this phase takes only one line of code, which simply passes the list of unique `TestCircuit` IDs to the checking routine.

Including comments and logging calls, 77 total lines of code are required to create 20 distinct variations of circuit type and transpiler configuration pairs and to generate 50 copies of each of them, to modify test parameters, execute tests, and gather the raw data.

The final results after statistical processing—independent of the QLP-TB—allow comparison between the effectiveness of each transpiler configuration at three tasks, measured on five circuits. Each configuration was assessed against SWAP count, Euclidean distance, and transpiler time.

Since SWAP gates drastically increase a circuit’s probability of failure, connec-

tivity and distance algorithms implemented by the various passes in the transpiler configurations seek to define mappings that minimize the total number of SWAPs that they must introduce into the provided circuits to successfully execute them. Lower numbers are better, and in some cases the transpiler configurations were successful enough at identifying efficiencies that via gate cancellation they reduced the necessary SWAPs to zero.

Since all quantum programs are fundamentally probabilistic in nature, the output of an algorithm cannot be well-defined by a scalar. Instead, each circuit probabilistically generates one measurement after each execution, with each measurement resulting in exactly one basis state of the Hilbert space the computation occurred in. Each trial circuit used 4 qubits and, in accordance with standard IBM QX procedure, each individual circuit was executed a total of 1024 times. This results in each circuit returning 1024 measurements, each measurement in one of 16 buckets. Let the result set of a circuit on \sqrt{n} qubits, run on a noiseless simulator be $S = \{s_0, s_1, \dots, s_{n-1}\}$ such that $\sum_{s \in S} s = 1024$ and the result set from execution on quantum hardware similarly be $R = \{r_0, r_1, \dots, r_{n-1}\}$. Then by calculating the Euclidean distance $D_e = \sqrt{\sum_{i=0}^{n-1} (s_i - r_i)^2}$, a metric of how reliable the circuit is can be generated – as defined by how near the actual output from execution on a Quantum Computer (QC) is to the expected result. Lower numbers are better.

An additional distance metric, the Jensen-Shannon distance, is also used to provide an alternative perspective of how near the actual distributions are to their ideal counterparts. Let $D(P||Q)$ be the Kullback-Leibler divergence of the probability distribution Q from the probability distribution P , where both are defined on the probability space X .

$$D(P||Q) = \sum_{\chi \in X} P(\chi) \log \frac{P(\chi)}{Q(\chi)} \quad (23)$$

And let $M = \frac{1}{2}(P + Q)$ be the pointwise mean of P and Q , then the Jensen-Shannon

distance is defined by,

$$JSD(P, Q) = \sqrt{\frac{D(P||M) + D(Q||M)}{2}} \quad (24)$$

This metric can be thought of as an assessment of the similarity of P and Q by measuring the information gained towards discriminating which of P and Q random variables were sampled from, as the number of samples increases—assuming a uniform prior probability. More information about the Jensen-Shannon divergence, from which Jensen-Shannon distance is derived, can be found in [10].

Finally, given the \mathcal{NP} -hard nature of the QLP, the time efficiency of various heuristics is of substantial interest. It is well known that the best possible mapping can be found, but not in any reasonable time using known methods. As such, QLP heuristics that are highly effective but extremely slow are of little interest. Lower numbers are better.

Although direct analysis of this information is not the primary goal of this research, some results are discussed in the context of the proposed Quantum Layout Solver (QLS) as confirmation of analyses made during its design.

4.2.4 QLP-TB Results Summary

As identified throughout Section 4.2.1 and Section 4.2.2, a series of tradeoffs were identified and weighed as part of the QLP-TB development process. In summary, these tradeoffs were:

1. **Usability vs. Extensibility:** Although in a commercial environment usability is often of prime importance, the nature of the scientific environment identified as the primary use context of the QLP-TB led to an alternative priority. Researchers are both more accustomed to using libraries and scripts to access

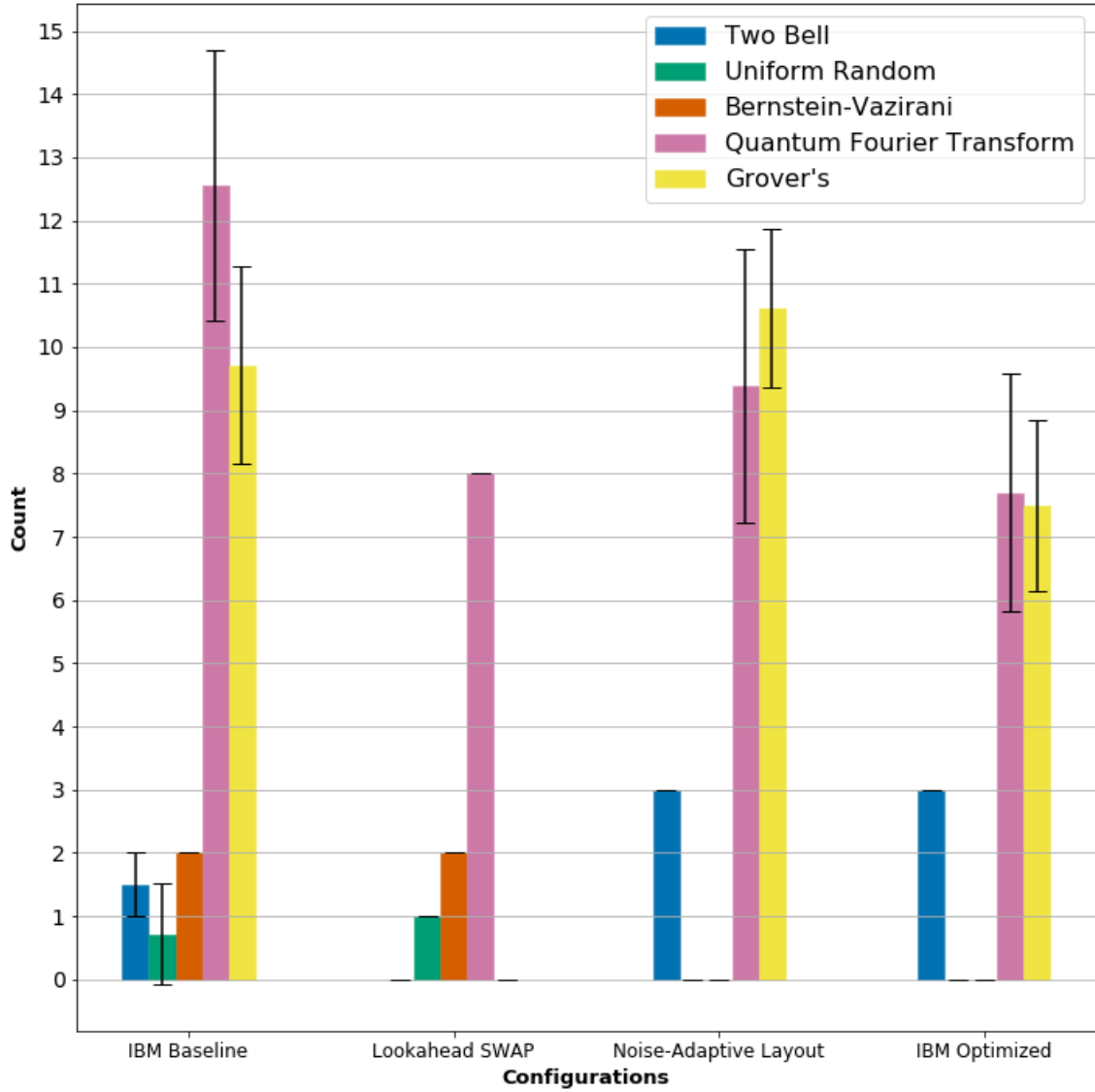


Figure 10: Increase in SWAP count after transpilation

functionality than commercial end users and also more likely to need to do so to more finely control experimental behavior.

2. **Reliability vs. Functionality:** Reliable data collection was identified to be of the utmost priority for scientific software. Unlike commercial software where applications frequently have known-good outputs that permit relatively easy verification, experiments do not. Data integrity often cannot be determined

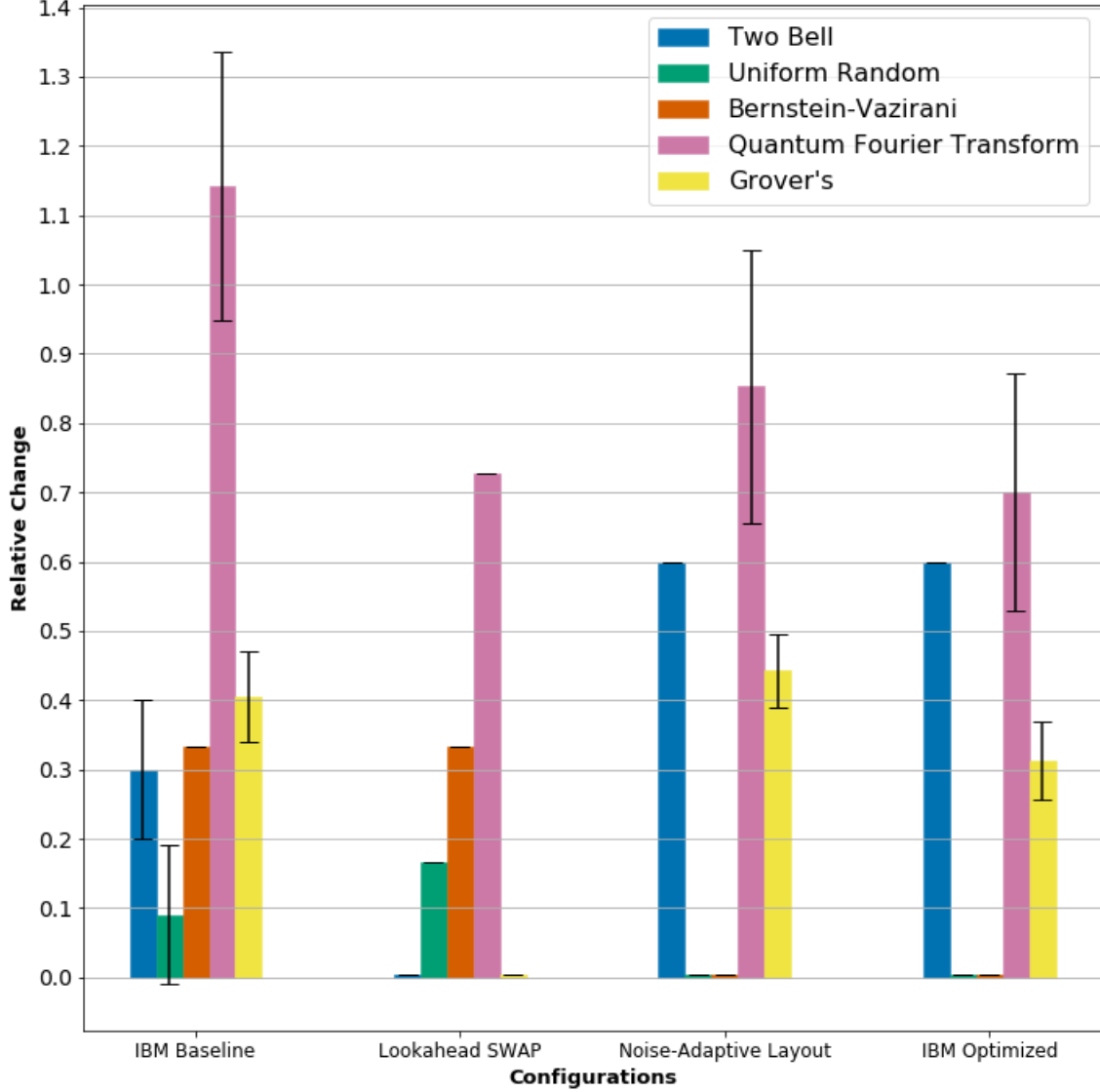


Figure 11: Increase in SWAP count after transpilation, relative to the starting size of the circuit

a posteriori. Instead, integrity must be ensured as much as possible at the framework level, even at the cost of functionality. In this case, the QLP-TB limits end user ability to define fully custom statistics to gather per experiment.

3. **Standardization vs. Maintainability:** Maintainability as a best practice is most frequently understood in a commercial context. Facts about this context

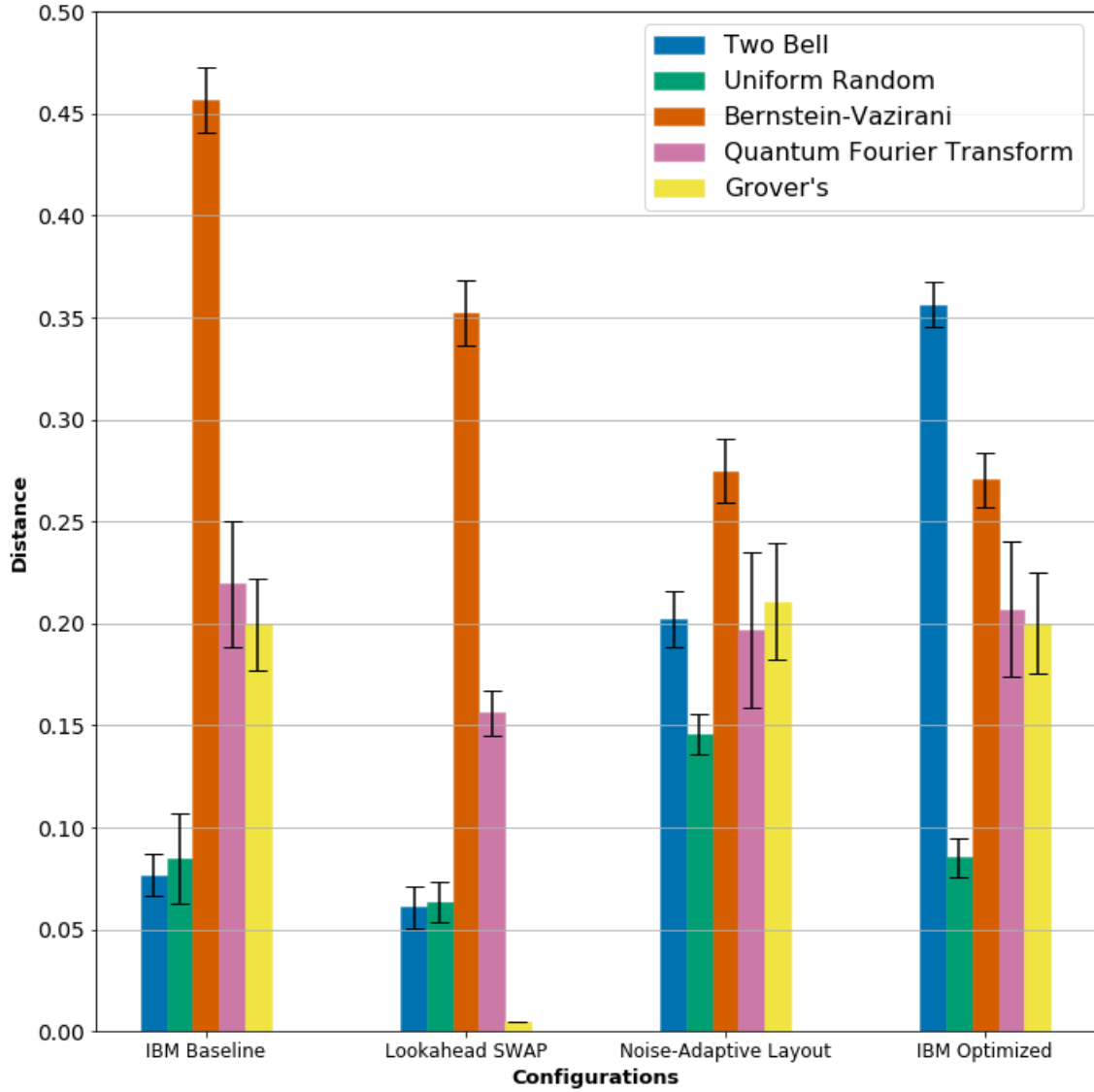


Figure 12: Euclidean distance between the results distributions from real and ideal executions

are not true in a scientific context: e.g., the availability of dedicated developers or reliable funding streams. Additionally, commercial products are, generally, not released until they have stable versions. Qiskit is an alpha product undergoing rapid change. The QLP-TB was made more maintainable in an academic environment by eschewing best practices like use of abstraction to separate interfaces and implementations.

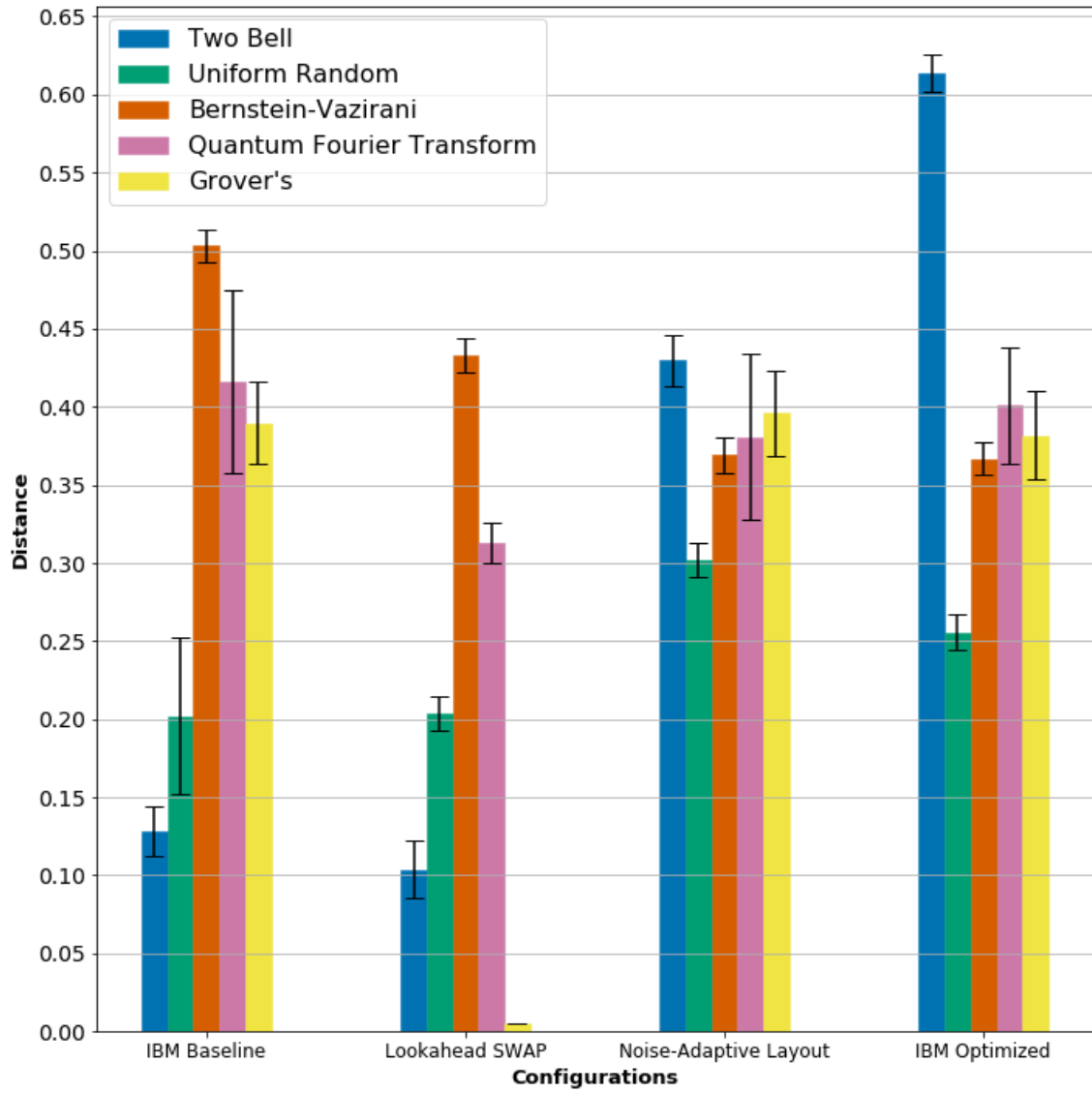


Figure 13: Jensen-Shannon distance between the measurement distributions of noise-less circuit simulation and actual execution on quantum hardware

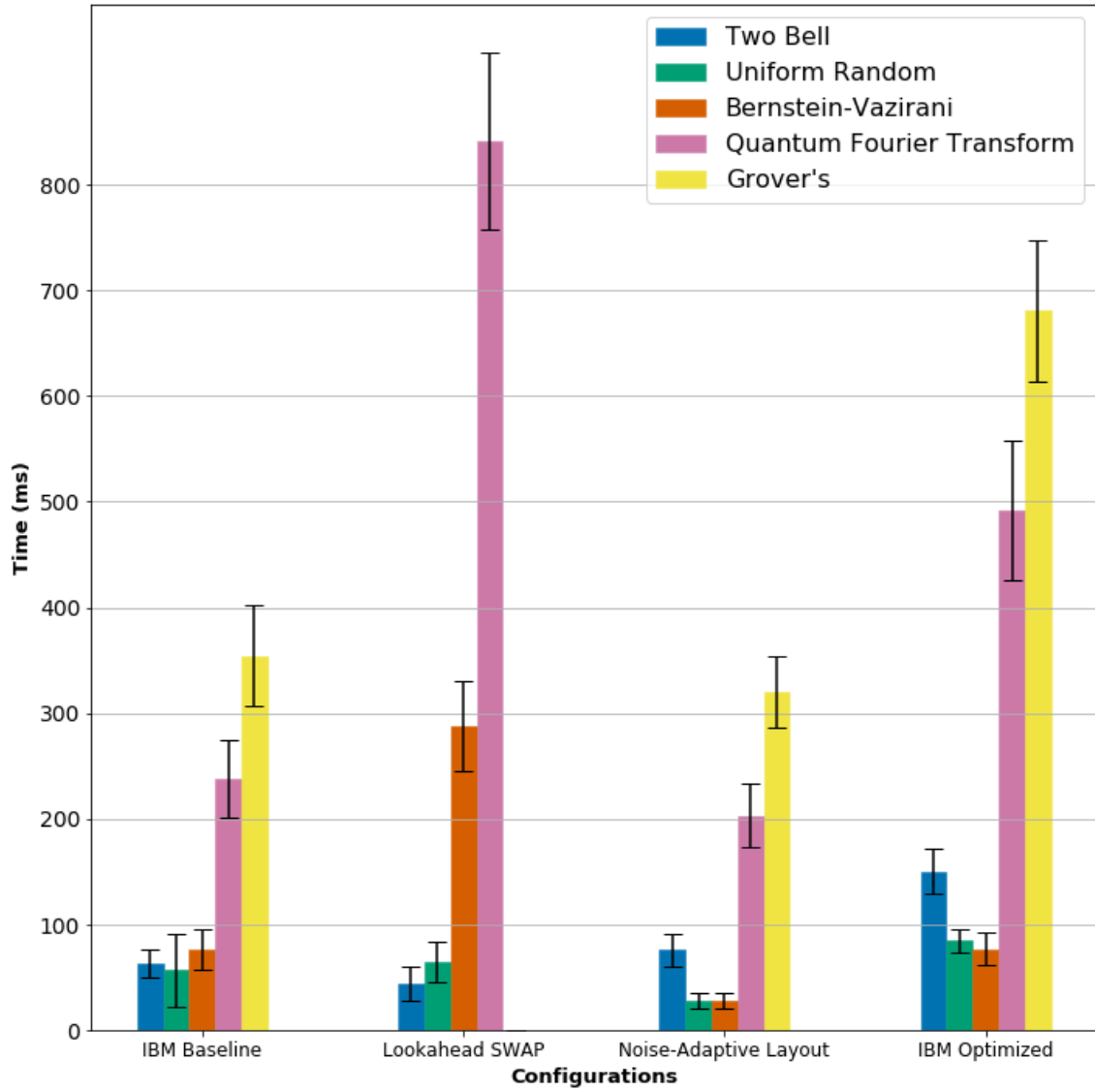


Figure 14: Transpilation time (ms) of each configuration and circuit

4.3 QLS Heuristic Results

There are two notable facets of the QLS heuristic design process that are weakly indicated by the experimental results generated as part of the QLP-TB verification process.

First, as was noted in Section 3.4 and expanded on in Section 5.4, circuit characterization appears to be a potentially important process that has yet to be implemented or tested. Ostensibly more effective connectivity and distance algorithms implemented by the `NoiseAdaptive` and `LookaheadSwap` passes show strong SWAP and Euclidean distance reduction in the Bernstein-Vazirani and Quantum Fourier Transform (QFT) circuits, respectively in Figures 10 and 12. But conversely shorter circuits with different constraint topologies like the Two Bell test circuit show significantly worse results. Although identifying the exact cause of this behavior is beyond the scope of this effort, the fact that the Bernstein-Vazirani and QFT circuits both tend towards a many-to-one constraint topology where the quantum state from many qubits is collected into a single qubit while the Two Bell and Uniform Random circuits have many-to-many constraints is at least indicative of an avenue for further exploration.

Second, it was noted in section 3.4.1 that existing algorithm methodologies that do not permit tradeoff between sub-graph connectivity and sub-graph distance from future layouts might make locally optimal but globally sub-optimal decisions. It was further noted that such mistakes were more likely when relatively few CX gates were implemented on each sub-graph before the next layout was instantiated. The fact that the `NoiseAdaptiveLayout` pass doubled the number of SWAP gates applied to the Two Bell circuit, and thus also significantly increased the resulting circuit’s distance from the correct result, is indicative that this hypothesis holds.

In particular, the `NoiseAdaptiveLayout` pass emphasizes selecting the best sub-graph for layout—where best is determined by the number of edges in the sub-graph and the

reliability of those edges—and is prone exactly to selecting layouts that are best for their layer but that might potentially make it more difficult or expensive to find good layouts for subsequent layers. Additionally, the Two Bell circuit is quite small, so any benefits generated by ensuring CX gates occur over reliable edges are mitigated by the fact that any given edge may be used very few times.

In this case, it appears that the pass selected a layout for the Two Bell circuit that would serve larger, more CX-intensive circuits well—as happened with Bernstein-Vazirani—but the extra cost of moving into or out of those layouts ended up creating a net negative reliability to circuit execution.

4.4 Summary

This chapter discussed the results generated from exploration of three research questions:

1. What are the design principles and requirements of an effective testbed for proposed QLP solvers?
2. What tradeoffs should be made among various software engineering principles in a testbed implementation satisfying those requirements?
3. Can a method be devised that mitigates the limitations to effectiveness and efficiency that exist with current QLP solutions?

The first question was answered by the design principle and requirements analysis found in Section 4.2.1 and Section 4.2.2, with the result that a system employing a diverse set of wrapper and utility functions, careful interface exposure of Qiskit functionality, and best practices database, logging, and error correction methods was determined to adequately meet all listed requirements.

The second question was answered in Section 4.2.4 and Section 4.2.3. The unique context of experimental and scientific software significantly modified ideal capabilities, user interface design, and the priority of considerations like data integrity, system reliability, and security. Finally, the verification procedure demonstrated the value of these tradeoffs when implemented and applied to practical experimentation.

The final question is left without a rigorous analysis and full experimental testing but was discussed in Section 4.3. However, data generated as part of the verification procedure made strong indications that hypotheses provided as part of the QLS development process are worth future investigation.

V. Conclusions

5.1 Overview

This chapter describes the contributions made to the quantum computing research field by this research in Section 5.2. In Section 5.3, avenues for growth and future development of the Quantum Layout Problem Testbed (QLP-TB) are discussed, with an emphasis on increasing the flexibility of the tool and promoting portability. Finally, additional formulations of the proposed Quantum Layout Solver (QLS) are discussed and proposed for research by interested parties in Section 5.4.

5.2 Contribution

Quantum computing has experienced an incredible surge in capability over the last few years [17]; Quantum Computers (QCs) have ever increasing numbers of available qubits and coherence times, decreasing error rates, and ever more varied topologies. Unfortunately, the libraries and tools available for research on these devices have not kept pace. Significant engagement with low-level hardware and behavior is still required by researchers seeking to improve the operational abilities of quantum hardware and quantum programming. This research has resulted in a contextually-specific analysis of the design principles and requirements of a new tool to provide this missing functionality. By means of a verification procedure, the QLP-TB has demonstrated how it can make quantum computer research more efficient and effective, and has also made steps towards guiding future transpiler optimization efforts towards the creation of a more cohesive body of knowledge. One that promotes collaboration, reproducibility, and more productive and structured experimentation.

Additionally, initial efforts were made towards the construction of a superior Quantum Layout Problem (QLP) solver that engages with identified limitations of ex-

isting methods. The underlying hypotheses related to the construction of this routine were also credibly supported by data gathered as part of the QLP-TB verification routine, providing a strong impetus and clear direction for future efforts directly related to this proposal.

5.3 Future work on the QLP-TB

The QLP-TB is an alpha product, and offers many opportunities for future development. The most significant existing limitation relates to the database model and operations. Currently, database operations are defined statically and are inflexible in the sense that they do not change or accommodate distinct table formats, and similarly although additional statistics can easily be gathered by *TestCircuit* objects, modifying the existing database schema to store and retrieve these statistics in a human-readable format is not trivial. The implementation of an Object-Relational Mapper (ORM)—as by the use of the Python SQLAlchemy library—would effectively solve this issue and yield notable benefit. ORMs provide an additional abstraction layer between program objects and the underlying database used to store attributes. This abstraction layer permits improved processing of data values to make storing, retrieving, and modifying database records significantly easier and provides flexibility for a wider variety of experimental statistics to be gathered.

Additionally, although experimental data is stored by the QLP-TB, data analysis is expected to be handled by external tools like *scipy*. Although it would be redundant and difficult to implement a custom statistical analysis capability, functions to wrap data in *pandas* dataframes and more easily hook into existing *scipy* interfaces could be of benefit.

One of the development priorities of the QLP-TB was to create a library accessible to improvement by follow-on researchers seeking to extend its capabilities. However,

as with many scientific computing libraries, the accessibility of functionality to end users can be less than ideal. Although documentation is extensive, improvements made by providing tutorial configurations, associated material, or even implementing some form of user interface may be beneficial for improving the use of the QLP-TB in academic and educational contexts.

Finally, the tradeoff paradigm established by the QLP-TB sought to provide exposure to native Qiskit functionality to enable extensive changes to underlying operations, while also simultaneously providing simple functions for tuning or modifying specific subsets of the transpiler operation. There is ample opportunity for future efforts at increasing the variety, scope, and population of simple transformation techniques—for example, to make the connectivity and distance metric comparisons described in Section 5.4 much easier to implement by creating a modular system for exchanging the metrics used within transpiler SWAP and layout routines.

5.4 Future Work on the QLP

Having first become publicly available in 2017, the IBM QX project is still in its infancy, and there are ample opportunities for optimization and further testing of heuristic solutions to the QLP. There are three, inter-related areas that are most fruitful for further efforts in characterizing QLP behavior: global optimizations, connectivity optimization, and distance optimization. Primarily, future work should emphasize building a solid foundation of comparative knowledge, since although research has introduced and tested specific techniques, little-to-no research has sought to develop a fundamental understanding of how varying techniques varies outcome in a systematic manner, and justifications for various decisions are elided.

A series of alternative functions for measuring concepts of connectivity and distance are introduced, any of which may conceivably demonstrate some advantage,

but primarily the central problem demonstrated by these alternatives is not *if they work* or show such advantage, but rather if decisions from among these alternatives *can be justified* in a rigorous, structured manner. Each quantum layout methodology consists of a variety of seemingly minor, arbitrary choices between metrics that must be rigorously evaluated to formalize this evolving field of research.

5.4.1 Global Improvements

Perhaps the most critical area for future research is to define and test circuit characterization schema. Current efforts attempt to measure themselves broadly across a variety of circuit topologies in order to generate average metrics indicative of generalized behavior and performance. However, it has become clear throughout work on this project that distinct circuit topologies are plausibly better dealt with by distinct layout solutions. Currently, look-ahead heuristics exist to attempt to predict how currently selected layouts may benefit future constraint operations—e.g. by reducing SWAPs required to meet future layouts, or by selecting to place virtual qubits in locations that maximize reliability for frequently entangled qubits [50, 54]—however, these efforts operate by identifying specific qubits with specific traits like maximal weight, or by examining the topology of the computational substrate.

Characterization at the circuit level shows promise for providing efficient means to distinguish types of circuits and then mapping those types to specific layout procedures. For example, characterization efforts could assess differences between the most entanglements in any circuit layer and the average entanglements per layer; doing so would potentially allow the transpiler to identify programs that have significant entanglements requirements for a specific sub-circuit, but primarily single-qubit operators for most of the algorithm, and in such a situation a layout procedure that is more computationally intensive can be selected to better optimize the mapping

associated with the entanglement sub-circuit at the cost of reducing time devoted to optimizing the relatively unimportant sub-circuits that have few entanglements.

Similarly, characterizing based on the ratio of $\rho = \frac{\text{vertices}_{\text{entangled}}}{\text{edges}_{\text{entangled}}}$ could provide insight into the kinds of entanglement occurring. $\rho \rightarrow 2$ when entanglement constraints apply to many distinct pairs (as in a circuit with many, independent, Bell pairs) while $\rho \rightarrow 1$ if entanglement tends to have a hub-and-spoke arrangement, and this property is distinct from existing metrics that merely sort qubits by entanglement count [33]. A similar measurement may be made by counting the number of entanglement operations per layer in an N qubit circuit, $c_\ell \in [1, \frac{N}{2}]$. Circuits characterized by many hub-and-spoke entanglements will tend towards $c_\ell \rightarrow 1$, while conversely distinct-pair entanglement will result in $C_\ell \rightarrow \frac{N}{2}$. Understanding these characteristics could easily change the value of optimizing ideal sub-graph selection (for hub-and-spoke) versus optimizing the minimum distance between a variety of clustered sub-graphs (for distinct-pair entanglement). Allowing the exogenous parameters p and γ to vary as a function of these metrics may show marked improvement over a naive, uniform solution for all circuits.

Ultimately, these characterization efforts could be integrated into a complete system for automatically tuning circuit transpilation. Given some quantum circuit to be executed and a reduced version exhibiting similar topological constraint properties (e.g. a full Grover’s algorithm search that is infeasible to simulate on classical hardware and an implementation on few qubits that is feasible to simulate), a routine could be developed to generate the ideal results of the reduced circuit on a simulator and then to execute the reduced version on quantum hardware. Taking the difference between the ideal and real results distributions with a continuous metric like that proposed by Guerrero [15] might permit gradient descent—or an analogous process—to be used to tune p and γ until some minimum distance is found. The resulting

transpiler tuning could then be applied to the full circuit being executed solely on quantum hardware.

5.4.2 Connectivity Improvements

The connectivity measure defined for the QLS is to measure internal connectivity of sub-graphs, $s \in S$, relative to the connectivity of a complete graph of the same size, $C(s) = \frac{\text{edge_count}(s)}{\text{edge_count}(\kappa_{s.\text{size}()})}$; this is driven primarily by existing methods [33]. However, while this decision has some empirical support in measured improvement in transpiler mapping behavior, this decision has not been justified directly against alternatives. Additional connectivity measures are likely of significant use especially when paired with circuit characterization, since presumably distinct connectivity concerns arise from distinct circuit topologies. For example, given the recent work on noise-adapting transpiler operations [33, 50], a modification to $C'(s) = \frac{\text{edge_count}(s) \prod_{e \in \text{edges}} \text{weight}_e}{\kappa_{s.\text{size}()}}$ would balance the internal connectivity of the selected sub-graph against the quality of those edges. It might conversely be more important to assess the average number of edges per node in the sub-graph than the global connectivity—as when multiple qubits regularly require multiple entanglements—and so $C''(s) = \frac{\sum_{n=\text{nodes}} \text{edge_count}(n)}{s.\text{size}()}$ might show improvement in selecting sub-graphs that are better for such a circuit. If the circuit contains a single, or few, extremely important hub qubits for entanglement, simply assessing the quality of the sub-graph with $C(s) = \max(\text{edge_count}(n))$ for $n = \text{node} \in s$ could be an extremely computationally efficient method for assessing sub-graph suitability for such circuits.

5.4.3 Distance Improvements

Thematically, potential for future research into distance component improvement is similar to that for connectivity improvements. That is, different functions for de-

termining or defining distance could plausibly show improvement across a variety of circuit topologies or may be well-suited to a characterization-based effort to select particular distance functions to optimize a narrower set of topologies. The existing distance component provides a computationally efficient method of look-ahead functionality by simply calculating the number of *future* but not *existing* circuit constraints that are met by the current layout. Since it is less important that constraint requirements in layers occurring significantly later in execution order are met than it is that constraints near in time are met, the distance function uses an exponential discount factor $\gamma^k \in [0, 1)$, where k increases with the number of layers separating the current layout from the constraint being assessed, to reduce the weight of future constraint satisfaction: let c represent the circuit to be executed, with c_ℓ being the sub-circuit of c partitioned into layer ℓ and let λ_ℓ be the layout defined on layer ℓ then $D_\lambda(c, s) = \sum_{k=\ell+1}^{\ell_{final}} \gamma^k * met_constraints(\lambda, \ell_k)$.

This distance formula should be assessed against a variety of alternatives, none of which have been used and measured in any systematic way. Since the purpose of $D_\lambda(c, s)$ is to heuristically determine layouts that will likely reduce SWAP requirements to meet future layer constraints, other methods of achieving this goal can be readily identified. For example, given a graph G and a sub-graph s of G , we can define the *external diameter* of s as the maximum path length from any node n in $G \setminus s$ to any node $m \in s$. The external diameter then functions as a measure of how far s is from an arbitrary qubit in G , and thus may provide a useful measure of how “far” s is from other sub-graphs used in mapping the circuit being executed. Similarly, external diameter could be modified to measure the average or minimum path length instead of maximum.

As a distinct measure, given some s of suitable size to layout all qubits used in circuit execution, an alternative distance function could be developed that measures

the total or average number of edges with exactly one end-point in s ; such a measure of *external edges* may again be useful in determining the ease with which virtual qubits that need to be entangled with a partner they currently do not share an edge with can be moved out of and into the chosen sub-graph. If instead the connectivity function were altered to select, for example, sub-graphs with a size less than the number of qubits required to execute the full circuit (as might happen if smaller, higher quality sub-graphs were prioritized over full sub-graphs due to the number of entanglements required being much less than the total number of qubits involved) then a distance measure on external edges might have significant value in heuristically assessing the flow rate between the high-quality “entanglement cluster” and the periphery that merely stores quantum state.

5.5 Concluding Remarks

Increasing the complexity of systems available to quantum computing researchers serves no purpose if there is not a corresponding improvement in the capabilities provided to researchers to exploit the additional opportunities presented by quantum system development. The continued necessity to perform manual, individual, or low-level operations on quantum systems will, if not resolved, hamper research by preventing the development of standards, impeding collaboration, and making efforts to extend and reproduce existing work ever more difficult. This issue is of concern not only to the academic community, but governmental organizations concerned with exploiting quantum capabilities to build and maintain the strategic margin of the United States over its adversaries.

First efforts were made at improving the ability of quantum computing researchers to create, modify, test, and report quantum circuit experiments, especially those concerned with transpiler optimization algorithms. These efforts comprised a design

principle analysis, functional requirements analysis, and testbed implementation and verification. Using data generated by the QLP-TB verification, additional inroads were made towards identifying potential improvements to existing transpiler routines.

Although significant progress has been made by this effort, there is ample opportunity for future research to build on these results. First, to improve and extend testbed functionality for broader use cases and provide quality of life improvements to improve the accessibility of this system to users. Second, to develop and test a newer class of optimization algorithm and, in doing so, to more solidly justify algorithmic procedures currently in use or proposed for future application.

The potential exists for vast improvement that brings into the grasp of the academic and government communities a new form of computational power, the limit of whose capabilities are not yet even known. By building and improving the tools required to enable more efficient and effective improvements in this realm, these capabilities can be brought closer to fruition.

Appendix A. Quantum Circuits Provided by the QLP-TB

Two Bell: The two-bell circuit creates random pairs of bell states across the width of the circuit, repeatedly until the final circuit is square. In particular, it operates only on sub-graphs with an even size and randomly permutes the available qubits. Given this permutation, a Bell state is formed from each consecutive pair of qubits in the permutation. Then the process repeats, with a new permutation, until the circuit depth is equal to the number of Bell state pairs formed in each layer.

Many to Many: This circuit is conceptually very simple. Given a circuit size, each available qubit is added to a candidate pool. Each candidate is given a number of available CX edges chosen uniformly at random from the interval $\llbracket 0, 5 \rrbracket$. Once the number of edges are selected, then that number of additional candidates are selected, uniformly at random, from the candidate pool to be the targets of a CX operation controlled by the original candidate. This process is continued for each candidate in the pool until all have been assigned CX targets. Essentially, this circuit forms an arbitrary web of many-to-many CX gates.

Grover: A full explanation of Grover’s algorithm can be found in [14], but in summary it is a circuit designed to perform an efficient search on an unsorted list. Beyond the fact that it is a practical quantum circuit where improved error performance can have real world consequence, it is included as a test circuit primarily because of the entanglement constraints it requires. Grover’s algorithm requires multi-controlled entanglement gates – that is, entanglements of more than 2 qubits simultaneously. Since the IBM QX architecture does not directly implement entanglements on > 2 qubits, the derived circuits are extremely SWAP intensive. Since the SWAP counts are not particularly tied to topological or layering concerns, it is unlikely the proposed QLS will show significant improvement over existing methods, doing so would

be particularly valuable.

Moving Island: The Moving Island circuit is a more structured variant of the Many to Many circuit. The primary difference is that while the Many to Many circuit permits each qubit to be both a control and a target to a CX operation, the Moving Island circuit is designed to mimic algorithm sub-routines that require specific qubits to store evolving quantum state. First, all available qubits are added to a candidate set. On each loop, a qubit is consumed from the candidate set and made the control of an “island”. The island is then associated with a number of edges, selected uniformly at random from the interval $[0, \text{len}(\text{candidates})]$. A number of qubits are consumed from the candidates set equal to the number of edges and set as CX targets of the central qubit. The loop is then repeated until all candidates are consumed. The entire loop is then repeated until the circuit is square. The final circuit contains clusters of distinct qubit entanglements, with the hub of the cluster varying on each iteration.

Uniform Random: Creates a circuit whose gates are uniformly chosen from H, X, Y, Z, S, T, CX and whose CX endpoints are chosen uniformly from available qubits. The loop is iterated until the circuit is square.

Bernstein-Vazirani: This test circuit implements the Bernstein-Vazirani algorithm [2]. An integer in the interval $0, \text{size}$ is selected as a truth value for the circuit. This truth value is encoded as binary representation into an oracle sub-circuit using CX gates. Then, the remainder of the circuit creates a uniform superposition to be fed into the oracle and the output is read into another uniform superposition. This algorithm is designed to permit recovery of the encoded truth value in constant time – while classical equivalents require *size* queries. The use of CX gates to encode the oracle and the fact that the truth value is well-defined and easily measurable make this a valuable contribution to the test set.

Toffoli: A Toffoli gate is an algorithmic primitive used in a variety of quantum algorithms [31]. Also known as the CCX gate, it implements a Pauli-X gate on the target qubit \iff both of two identified control qubits have a non-zero $|1\rangle$ component. Because this gate allows arbitrary continuation of entanglement gates with n controls to structures providing $n+1$ controls, it has significant value in both the functionality it provides and as a sub-circuit to optimize, since any derived performance advantage will accrue throughout execution of the algorithm being implemented. In particular, this test circuit uses the IBM Aqua library’s *mcz()* implementation.

Quantum Fourier Transform: Finally, the Quantum Fourier Transform (QFT) is implemented as another practical circuit often implemented as a sub-circuit to algorithms [43]. The QFT is generally used to recover phase information from a given quantum state provided as input, and more generally serves to provide a structured basis transform: Let $|x\rangle = \sum_{i=0}^{N-1} x_i |i\rangle$ be the original basis and $|y\rangle = \sum_{i=0}^{N-1} y_i |i\rangle$ be the desired output basis of the transform. Then under a QFT each coefficient y_k for $k \in \llbracket 0, N-1 \rrbracket$ is defined as $y_k = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} x_i \omega_N^{ki}$, where ω_N is the N^{th} root of unity [6]. This can also be thought of as the quantum analogue to the classical inverse Fourier transform. Additionally, since the key operator used by the QFT is a controlled phase shift – implemented by IBM using a CX gate – then there is significant opportunity for improving implementation with the QLS.

Circuit Name	Function Name	Example Implementation
Two Bell	two_bell	
Many to Many	m_to_m	
Moving Island	moving_island	
Uniform Random	uniform_random	
Bernstein-Vazirani	bv	
Toffoli	toff	
QFT	qft	
Grover's	grover	See fig. 8

Table 3: Summary of test set circuits available in the QLP-TB

Appendix B. Experiment Script for the QLP-TB

Listing B.1: Complete Experiment Run for Verification for the QLP-TB

```
1 def run_local_experiment() -> List[str]:
2     """By Brandon Kamaka, 30 Jan 2020.  Reproducibility experiment
3     ↪ to validate test bed
4     Create a series of test circuits, and transpile each series
5     ↪ with distinct options from various layout and SWAP
6     optimizing papers.  Compare success, SWAP efficiency, and
7     ↪ time efficiency metrics
8     """
9
10    from qiskit.transpiler import CouplingMap
11    from qiskit.transpiler.passes import LookaheadSwap, DenseLayout
12
13    dbc.set_db_location('data/circuit_data.sqlite')
14    pass_configurations = {
15        0: 'IBM Baseline',
16        1: 'Lookahead SWAP',
17        2: 'Noise-Adaptive (GreedyE)',
18        3: 'IBM Optimized'
19    }
20
21    circuits_to_test = {
22        0: 'two_bell',
23        1: 'uniform_random',
24        2: 'bv',
25        3: 'qft',
26        4: 'grover'
27    }
28
29    num_trials = 50
```

```

27     tests_all_ids = []
28
29     for conf in pass_configurations.keys():
30         for circ_case in circuits_to_test.keys():
31             test_config = (conf, circ_case)
32             logger.info(f'+++++++TEST CONFIG: {test_config
↳ }+++++')
33
34             # ***** Phase 1: Make initial
↳ circuits and prep
35             case = circuits_to_test[test_config[1]]
36             filename = f'{pass_configurations[test_config[0]]} - {
↳ case}'
37
38             # Make the Premades object.
39             # It does not contain a circuit but stores the uniform
↳ information for circuit creation.
40             exp_size = 4
41             exp_truth_value = 3
42             circ = Premades(size=exp_size, truth_value=
↳ exp_truth_value, measure=True)
43
44             # Actually add a specific QuantumCircuit instance based
↳ on the stored parameters
45             Premades.circ_lib[case](circ)
46             circ.draw(output='mpl',
47                         filename=filename)
48             tests = []
49             for i in range(num_trials):
50                 # Create distinct TestCircuit objects (so that each
↳ gets its own unique ID),
51                 # but each TC gets the same PreMade

```

```

52         tc = TestCircuit()
53         tc.add_circ(circ, size=exp_size, truth_value=
↳ exp_truth_value, measure=True)
54         tc.stats.name = case
55         tc.stats.notes = filename + f' - {i}'
56         tests.append(tc)
57
58         # Register initial statistics
59         dbc.write_objects(dbc.db_location, tests)
60
61         # ***** Phase 2, make the distinct
↳ PassmMnagers for each test config and circuit
62
63         # Start by getting a transpiler config from the circuits
↳ and backend
64         level = 1 if pass_configurations[test_config[0]] != 'IBM
↳ Optimized' else 3
65         configs = transpilertools.get_transpiler_config(circs=
↳ tests, be=backend, optimization_level=level)
66
67         # Then we use the configs to get the appropriate
↳ PassManager for each configuration
68         pms = []
69         for idx, config in enumerate(configs):
70             pm = transpilertools.get_basic_pm(config, level=
↳ level)
71             cm = CouplingMap(backend.configuration().
↳ coupling_map)
72
73             if test_config[1] == 1:
74                 pass_type = 'swap'
75                 new_pass = LookaheadSwap(coupling_map=cm)

```

```

76
77         elif test_config[1] == 2:
78             pass_type = 'layout'
79             new_pass = DenseLayout(coupling_map=cm,
↪ backend_prop=backend.properties())
80
81         else:
82             modified_pm = pm
83             continue
84
85             modified_pm = transpilertools.get_modified_pm(
↪ pass_manager=pm, version=level, pass_type=pass_type,
86
↪ new_pass=new_pass)
87             pms.append(modified_pm)
88
89             # logger.info(f'+++++++PM BEING USED
↪ ++++++')
90             # logger.info(transpilertools.get_passes_str(pms[0]))
91             # logger.info(f
↪ '+++++')
↪
92
93             # ***** Phase 3: Run tests on
↪ circuits with custom PassManagers
94
95             # Just in case our number of test_cases exceeds a
↪ reasonable size (25)
96             test_batches = get_batches(tests)
97             pms_batches = get_batches(pms)
98             assert len(test_batches) == len(pms_batches)
99             for test_batch, pms_batch in zip(test_batches,

```



```

    ↪ pms_batches):
100         TestCircuit.run_all_tests(test_batch, pass_manager=
    ↪ pms_batch, be=PREFERRED_BACKEND, attempts=5)
101
102         # ***** Phase 4: Return final circ
    ↪ ids so the normal routine can save them to Stats
103         tests_all_ids.extend([test.id for test in tests])
104
105     return tests_all_ids

```

Appendix C. Code Snippets

Listing C.1: Constructor for the *TestCircuit* class

```
1 def __init__(self):
2     self.stats = Statblock(parent=self)
3     self.compiled_circ = None
4     self.backend = None
5     self.job_id = None
6     self.transpiler_config = None
7     self.circuit = None
8
9     # if isinstance(circuit, QuantumCircuit):
10    #     self.circuit = circuit
11    # elif circuit is not None:
12    #     raise TypeError(f'Circuit must be a QuantumCircuit, or
    ↳ Premade. Was given type: {type(circuit)}')
```

Listing C.2: Example wrapper function to automate testing transpiler operations

```
1 def transpile_test(self, pass_manager=None, default_be=
    ↳ preferred_backend, ATTEMPTS: int = 1) -> QuantumCircuit:
2     """ Transpile TestCircuit with provided pass_manager and
    ↳ register statistics, but do not execute.
3
4     Args:
5         pass_manager (PassManager): Custom PassManager to use to
    ↳ transpile this circuit.
6         default_be (str): Optional. Default backend to use for
    ↳ transpilation; defaults to preferred_backend defined
7         in run_experiment.py
8         ATTEMPTS (int): Optional. Number of transpile tests to
    ↳ be run to generate averages.
```

```

9
10     Returns:
11         qiskit.circuit.quantumcircuit.QuantumCircuit: Returns
12         ↳ the compiled circuit for chaining; also saves it to
13             self.compiled_circ as a side-effect.
14
15     """
16
17     if self.backend is None:
18         logger.warning(f'Transpiler: Circuit ({self.id}) had no
19         ↳ backend.  Resorted to default: {preferred_backend}')
20         self.backend = default_be
21
22     transpile_times = []
23
24     # Get the average transpile time over ATTEMPTS number of
25     ↳ trials
26
27     for i in range(ATTEMPTS):
28         start_time = time.process_time()
29         self.compiled_circ = transpile(self.circuit,
30                                     backend=self.
31         ↳ get_circ_backend(),
32                                     optimization_level=0,
33                                     pass_manager=pass_manager
34         ↳ )
35
36         transpile_times.append(time.process_time() - start_time)
37
38     tc: QuantumCircuit = self.compiled_circ
39     stats = self.stats
40
41     logger.info(f'Transpiled and registered {self.stats.name}: {
42     ↳ self.id}')
43
44

```

```

35     # Returns average in ms
36     stats.compile_time = (sum(transpile_times) * (10 ** 3)) /
    ↪ len(transpile_times)
37     stats.post_depth = tc.depth()
38
39     logger.info(f'Transpiled circ of depth {stats.post_depth} in
    ↪ {stats.compile_time}ms.')
40
41     pre_cx = 0
42     post_cx = 0
43     if 'cx' in self.circuit.count_ops().keys():
44         pre_cx = self.circuit.count_ops()['cx']
45     if 'cx' in tc.count_ops().keys():
46         post_cx = tc.count_ops()['cx']
47
48     stats.swap_count = (post_cx - pre_cx) / 3
49
50     dbc.write_objects(dbc.db_location, [self])
51
52     return tc

```

Listing C.3: Example wrapper function to automate execution testing, executing, and timing transpiler operations

```

1 @staticmethod
2     def run_all_tests(tests: Union[List[TestCircuit], List[
    ↪ QuantumCircuit], TestCircuit, QuantumCircuit],
3         pass_manager: Union[PassManager, List[
    ↪ PassManager]] = None, generate_compiled: bool = True,
4         be: str = preferred_backend, attempts: int =
    ↪ 1) -> None:
5     """ Given a circuit or list of circuits to execute, it

```

```

6         executes all of them and writes all results to the
           appropriate db. Depending on parameters, a custom
7         PassManager can be used, and the circuits will also be
           compiled before execution.
8
9         Args:
10             tests (List[TestCircuit]): Circuits to be tested
11             pass_manager (PassManager): Custom PassManager to use
12             for transpilation, if desired. Default: IBM default
13             generate_compiled (bool): If True, will transpile
14             circuits prior to execution
15             be (Backend): IBM backend to use for transpilation and
16             execution. Default: preferred_backend
17             attempts: Number of times to transpile the circuits to
18             generate average compile time
19
20         Returns: None (but writes results to statistics database as
21         a side effect)
22
23         """
24
25         if not isinstance(tests, List): tests = [tests]
26         if len(tests) > 25:
27             logger.warning(f'Batch size might exceed maximum.
28             Currently {len(tests)}')
29
30         # If the circuits have been separately transpiled, we need
31         to ensure they were done so uniformly
32
33         compiled_circs = []
34
35         if not generate_compiled:
36             if len({tc.backend for tc in tests}) != 1:
37                 raise ValueError(f'All circuits in the same batch

```

```

    ↪ must use the same backend.')
29
30     compiled_circs = [tc.compiled_circ for tc in tests]
31     if None in compiled_circs:
32         raise ValueError(f'Test Run failed on batch (first
    ↪ id: {tests[0].id}). ')
33         f'No transpiled circuits available.
    ↪ ',
34         f'Set generate_compiled=True to
    ↪ have this done automatically')
35     else:
36         # If a a list of PassManagers of the same len() as tests
    ↪ was provided, we're good. Otherwise listify.
37         if not isinstance(pass_manager, List):
38             pass_manager = [pass_manager for t in tests]
39
40         elif len(pass_manager) != len(tests):
41             raise IndexError(f'Error in function run_all_tests:
    ↪ Mismatch in len(tests) && len(pass_manager)')
42
43         for idx, tc in enumerate(tests):
44             tc.backend = be
45             tc.transpile_test(pass_manager=pass_manager[idx],
    ↪ default_be=be, ATTEMPTS=attempts)
46             compiled_circs.append(tc.compiled_circ)
47             tc.stats.iteration = idx
48
49     dbc.write_objects(dbc.db_location, tests)
50     job = execute(compiled_circs, backend=tests[0].
    ↪ get_circ_backend())
51     for tc in tests:
52         tc.get_ideal_result()

```

```
53         tc.job_id = job.job_id()
54
55         dbc.insert_in_progress(dbc.db_location, tests)
56         dbc.write_objects(dbc.db_location, tests)
```

Listing C.4: Constructor for the *Premades* class

```

1 def __init__(self, size: int, truth_value: int, measure: bool = True
    ↪ , seed: int = None):
2     """ Creates a Premades object that wraps QuantumCircuits to
    ↪ carry additional information. Most important is
3         that the PreMades object stores the uniform interface
    ↪ parameters for generating new circuits.
4
5     Args:
6         size (int): Width of the desired circuit. i.e. the
    ↪ register size of the quantum register defining it.
7         truth_value (int): An integer to encode in any oracles
    ↪ that the circuit uses. Usually used to define
8         the "right" value for the circuit to return. E.g.
    ↪ the correct value for a grover's search to find.
9         measure (bool): Optional. If True, adds measurement
    ↪ operators to the end of the circuit.
10        seed (int): Optional. If not None, the provided seed is
    ↪ used to set random state for reproducibility.
11    """
12    if truth_value == 0:
13        logger.warning('Truth values that evaluate to basis 00
    ↪ ... 00 may cause misleading accuracy measurements')
14
15    qr = QuantumRegister(size, 'qr')
16    cr = ClassicalRegister(size, 'cr')
17    super().__init__(qr, cr, name='qc')
18    self.circ_size = size
19    self.truth_value = truth_value
20    self.meas = measure
21    self.seed = seed

```


Listing C.5: Easy access function to retrieve transpiler configurations for a list of *TestCircuits*

```

1 def get_transpiler_config(circs: Union[List[TestCircuit],
    ↪ TestCircuit, List[QuantumCircuit], QuantumCircuit],
2
    be: basebackend, layout: Layout = None,
    ↪ optimization_level: int = None,
3
    callback: callable = None) -> List[
    ↪ TranspileConfig]:
4
    """ Given a list of circuits and a backend to execute them on,
    ↪ return a list of transpiler configs of the same
5
        length such that configs[i] is the config for circs[i]
6
7     Args:
8         circs (Union[List[qls.circuits.TestCircuit], qls.circuits.
    ↪ TestCircuit]): List of circuits to
9
        compile configurations for
10        be (qiskit.providers.ibmq.ibmqbackend.IBMQBackend): Backend
    ↪ object to execute the circuits on.
11        layout (Layout): Optional. Initial layout to use.
12        optimization_level (int): Optional. IBM transpiler
    ↪ optimization level to target [0, 3].
13        callback (Callable): Optional. Function to call at the end
    ↪ of execution of each pass in the PassManager.
14
15     Returns:
16         List[qiskit.transpiler.transpile_config.TranspileConfig]:
    ↪ List of transpiler configurations associated with
17
        circs.
18
19     """
    # First, parse the input type of circs and process it correctly
    ↪ to return a list of only QuantumCircuits

```

```

20     # Also set a flag to save TestCircuit.transpiler_config to
    ↪ member field if TestCircuits were provided.
21     circuits = []
22     save_configs = False
23     if isinstance(circs, List):
24         if isinstance(circs[0], TestCircuit):
25             circuits = [tc.circuit for tc in circs]
26             save_configs = True
27         elif isinstance(circs[0], QuantumCircuit):
28             circuits = circs
29     elif isinstance(circs, TestCircuit):
30         circuits.append(circs.circuit)
31         circs = [circs]
32         save_configs = True
33     elif isinstance(circs, QuantumCircuit):
34         circuits.append(circs)
35     else:
36         raise TypeError(f'The circuit must be a single
    ↪ QuantumCircuit (or subclass) or list of elements of that type.
    ↪ ')
37         f'Instead received: {type(circs)}')
38
39     # _parse_transpile_args will call _parse_x_args() where x is
    ↪ each parameter type.
40     # If this parameter is None, then each _parse_x_arg function
    ↪ will retrieve that parameter from backend.
41     # Hence backend being the only requirement. All other params
    ↪ exposed by get_transpiler_config are for custom tests
42     configs = _parse_transpile_args(circuits, backend=be,
    ↪ basis_gates=None, coupling_map=None,
43                                     backend_properties=None,
44                                     initial_layout=layout,

```

```

45         ↪ seed_transpiler=None,
                                optimization_level=
46         ↪ optimization_level,
                                pass_manager=None, callback=
47         ↪ callback, output_name=None)
48
49     if save_configs:
50         for idx, circ in enumerate(circs):
51             circ.transpiler_config = configs[idx]
52
53     return configs

```

Listing C.6: Utility function to retrieve native Qiskit pre-populated PassManager objects

```

1 def get_basic_pm(transpiler_config: TranspileConfig, level: int = 0)
2     ↪ -> PassManager:
3
4     """ Get a pre-populated PassManager from the native Qiskit
5     ↪ implementation.
6
7     Args:
8         transpiler_config (qiskit.transpiler.transpile_config.
9     ↪ TranspileConfig): Configuration used to generate the
10         tailored PassManager.
11         level (int): Optional. Qiskit Transpiler optimization level
12     ↪ to target.
13
14     Returns:
15         qiskit.transpiler.passmanager.PassManager: PassManager
16     ↪ instance associated with the provided config.
17
18     """
19
20     pm_funcs = {

```

```

13         0: level_0_pass_manager ,
14         1: level_1_pass_manager ,
15         2: level_2_pass_manager ,
16         3: level_3_pass_manager
17     }
18
19     pm_func = pm_funcs[level]
20     return pm_func(transpiler_config)

```

Listing C.7: Wrapper function to enable quick modification of pre-populated Pass-Manager objects

```

1 def get_modified_pm(pass_manager: PassManager, version: int,
2     ↳ pass_type: str, new_pass: BasePass) -> PassManager:
3     """ Modifies a provided PassManager instance by exchanging swap
4     ↳ or layout passes with others of the same basic type.
5
6     Args:
7         pass_manager (qiskit.transpiler.passmanager.PassManager):
8         ↳ PassManager instance to modify.
9         version (int): Which optimization level the original
10        ↳ PassManager was targeted at.
11         pass_type (str): Type of pass to exchange. Must be one of
12        ↳ ('swap', 'layout')
13         new_pass (BasePass): The pass to insert into pass_manager in
14        ↳ place of that pass_manager's pass of type (type).
15
16     Returns:
17         qiskit.transpiler.passmanager.PassManager: Modified
18        ↳ PassManager instance.
19     """
20
21

```

```

14     if version not in range(4):
15         raise ValueError(f'version must correspond to an existing
↳ optimization level (range(4)). Got {version}')
16
17     if not pass_type == 'swap' and not pass_type == 'layout':
18         raise KeyError(f'Can only exchange swap or layout passes.
↳ Was given type {pass_type}')
19
20     # Map of which indices the pass of each type are located at for
↳ each basic pm
21     locations = {
22         0: {'swap': (6, 0), 'layout': (1, None)},
23         1: {'swap': (7, 1), 'layout': (1, None)},
24         2: {'swap': (6, 1), 'layout': (2, None)},
25         3: {'swap': (6, 1), 'layout': (2, None)}
26     }
27
28     # The particular pass to replace depends on which test group the
↳ given pm is supposed to work for
29
30     # The transpiler passmanager format is a mess. The PassManager
↳ is actually gives us a list of dictionaries
31     # of dictionaries of lists of passes. No, I'm not kidding.
32     pass_list = pass_manager.passes()
33     first_index = locations[version][pass_type][0]
34     inner_index = locations[version][pass_type][1]
35     passes_dict = pass_list[first_index]
36
37     passes = []
38     for i in range(len(passes_dict['passes'])):
39         if inner_index is None:
40             passes = new_pass

```

```
41         break
42     elif i == inner_index:
43         passes.append(new_pass)
44     else:
45         passes.append(passes_dict['passes'][i])
46
47     pass_manager.replace(first_index, passes)
48
49     return pass_manager
```

Listing C.8: Constructor for the *statblock* class

```
1 def __init__(self, parent):
2     self.id = uuid.uuid4().hex
3     self.parent = parent
4     self.name = None
5     self.truth_value = None
6     self.ideal_distribution = None
7     self.results = None
8
9     self.circ_width = None
10    self.pre_depth = None
11    self.seed = None
12
13    self.backend = None
14    self.post_depth = None
15    self.swap_count = None
16
17    self.compile_time = None
18
19    self.datetime = None
20    self.iteration = None
21
22    self.batch_avg = None
23    self.global_avg = None
24
25    self.notes = None
```

Bibliography

1. A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques & Tools*, 2nd ed. Pearson, 1986.
2. E. Bernstein, U. Vazirani, and S. J. Comput, “QUANTUM COMPLEXITY THEORY,” *Society for Industrial and Applied Mathematics*, vol. 26, no. 5, p. 7, 1997. [Online]. Available: <http://www.siam.org/journals/sicomp/26-5/30092.html>
3. A. Botea, A. Kishimoto, and R. Marinescu, “On the Complexity of Quantum Circuit Compilation,” in *The Eleventh International Symposium on Combinatorial Search*. Association for the Advancement of Artificial Intelligence, 2018. [Online]. Available: www.aaai.org
4. M. Bysiek, A. Drozd, and S. Matsuoka, “Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints,” in *Proceedings of PyHPC 2016: 6th Workshop on Python for High-Performance and Scientific Computing - Held in conjunction with SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*. Institute of Electrical and Electronics Engineers Inc., 1 2017, pp. 9–18.
5. D. Chandra, Z. Babar, H. V. Nguyen, D. Alanis, P. Botsinis, S. X. Ng, and L. Hanzo, “Quantum Topological Error Correction Codes are Capable of Improving the Performance of Clifford Gates,” *IEEE Access*, vol. 7, pp. 121 501–121 529, 8 2019.
6. D. Coppersmith, “An approximate Fourier transform useful in quantum factoring,” IBM Research Division, Yorktown Heights, New York, Tech. Rep., 1 1995. [Online]. Available: <http://arxiv.org/abs/quant-ph/0201067>

7. D-Wave Systems Inc., “D-Wave System Documentation,” 2019. [Online]. Available: https://docs.dwavesys.com/docs/latest/c_gs_2.html
8. S. Developers, “Numpy and Scipy Documentation — Numpy and Scipy documentation,” 2020. [Online]. Available: <https://docs.scipy.org/doc/>
9. E. Deza and M. M. Deza, *Encyclopedia of Distances*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. [Online]. Available: www.springer.comhttp://link.springer.com/10.1007/978-3-642-00234-2
10. D. Endres and J. Schindelin, “A new metric for probability distributions,” *IEEE Transactions on Information Theory*, vol. 49, no. 7, pp. 1858–1860, 7 2003. [Online]. Available: <http://www.stat.cmu.edu/~minka/pa-http://ieeexplore.ieee.org/document/1207388/>
11. D. Falessi and F. Shull, “Towards flexible automated support to improve the quality of computational science and engineering software,” in *2013 5th International Workshop on Software Engineering for Computational Science and Engineering, SE-CSE 2013 - Proceedings*, 2013, pp. 88–91.
12. M. Fingerhuth, T. Babej, and P. Wittek, “Open source software in quantum computing,” *PLOS ONE*, vol. 13, no. 12, p. e0208561, 12 2018. [Online]. Available: <http://dx.plos.org/10.1371/journal.pone.0208561>
13. W. Finigan, M. Cubeddu, T. Lively, J. Flick, P. Narang, and J. A. Paulson, “Qubit Allocation for Noisy Intermediate-Scale Quantum Computers,” Cambridge, MA, 2018. [Online]. Available: <https://arxiv.org/pdf/1810.08291.pdf>
14. L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*

- *STOC '96*. New York, New York, USA: ACM Press, 2003, pp. 212–219.
[Online]. Available: <http://portal.acm.org/citation.cfm?doid=237814.237866>
- 15. N. Guerrero, “Solving Combinatorial Optimization Problems using the Quantum Approximation Optimization Algorithm,” Wright-Patterson, AFB, 2020.
- 16. R. Hamerly, T. Inagaki, P. McMahon, D. Venturelli, A. Marandi, T. Onodera, E. Ng, E. Rieffel, M. M. Fejer, S. Utsunomiya, H. Takesue, and Y. Yamamoto, “Quantum vs. Optical Annealing: Benchmarking the OPO Ising Machine and D-Wave - IEEE Conference Publication,” in *2018 Conference on Lasers and Electro-Optics (CLEO)*. San Jose: IEEE, 2018. [Online]. Available: <https://ieeexplore-ieee-org.aft.idm.oclc.org/document/8426769>
- 17. K. Hartnett, “A New “Law” Suggests Quantum Supremacy Could Happen This Year - Scientific American,” *Quanta Magazine*, pp. 1–1, 6 2019.
- 18. A. A. Houck, J. Koch, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, “Life after charge noise: recent results with transmon qubits,” *Quantum Information Processing*, vol. 8, no. 2-3, pp. 105–115, 6 2009. [Online]. Available: <http://link.springer.com/10.1007/s11128-009-0100-6>
- 19. Human Engineering, “MIL-STD-1472G Design Criteria Standard,” 2012. [Online]. Available: http://everyspec.com/MIL-STD/MIL-STD-1400-1499/MIL-STD-1472G_39997/
- 20. IBM, “IBM Announces Advances to IBM Quantum Systems & Ecosystem,” 11 2017. [Online]. Available: <https://www-03.ibm.com/press/us/en/pressrelease/53374.wss>
- 21. IBM Quantum Experience, “Qiskit API documentation,” 2020. [Online]. Available: <https://qiskit.org/documentation/index.html>

22. S. Jandura, “Improving a Quantum Compiler,” 2018. [Online]. Available: <https://medium.com/qiskit/improving-a-quantum-compiler-48410d7a7084>
23. T. Kadowaki and H. Nishimori, “Quantum annealing in the transverse Ising model,” *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, vol. 58, no. 5, pp. 5355–5363, 1998.
24. P. Knight, “Quantum communication and quantum computing,” in *Technical Digest. Summaries of Papers Presented at the Quantum Electronics and Laser Science Conference*. Baltimore, Maryland: IEEE, 1992, p. 32. [Online]. Available: <http://ieeexplore.ieee.org/document/807126/>
25. A. Kole, S. Hillmich, K. Datta, R. Wille, and I. Sengupta, “Improved Mapping of Quantum Circuits to IBM QX Architectures,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
26. P. Leach, Microsoft, M. Mealling, Refactored Networks, R. Salz, and I. DataPower Technology, “RFC 4122,” 2005.
27. C. C. Lin, S. Sur-Kolay, and N. K. Jha, “PAQCS: Physical Design-Aware Fault-Tolerant Quantum Circuit Synthesis,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1221–1234, 7 2015.
28. N. M. Linke, D. Maslov, M. Roetteler, S. Debnath, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe, “Comparing the architectures of the first programmable quantum computers,” in *Optics InfoBase Conference Papers*, vol. Part F81-EQEC 2017. OSA - The Optical Society, 2017.
29. D. Loss and D. P. DiVincenzo, “Quantum computation with quantum dots,” *Physical Review A*, vol. 57, no. 1, p. 120, 1 1998. [Online]. Available: <https://journals.aps.org/pr/abstract/10.1103/PhysRevA.57.120>

30. J. Majer, J. M. Chow, J. M. Gambetta, J. Koch, B. R. Johnson, J. A. Schreier, L. Frunzio, D. I. Schuster, A. A. Houck, A. Wallraff, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, “Coupling superconducting qubits via a cavity bus,” *Nature*, vol. 449, no. 7161, pp. 443–447, 9 2007.
31. D. Maslov, “On the Advantages of Using Relative Phase Toffolis with an Application to Multiple Control Toffoli Optimization,” 2016. [Online]. Available: <https://arxiv.org/pdf/1508.03273.pdf>
32. D. Maslov, S. M. Falconer, and M. Mosca, “Quantum Circuit Placement *,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 752–763, 2008.
33. P. Murali, J. M. Baker, A. J. Abhari, F. T. Chong, and M. Martonosi, “Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers,” in *proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, I. Bahar and M. Herlihy, Eds. Providence, Rhode Island: Association for Computing Machinery, 2019. [Online]. Available: <https://arxiv.org/pdf/1901.11054.pdf>
34. M. A. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, 10th ed. New York, New York, USA: Cambridge University Press, 2011.
35. Y. Park, R. Scott, and S. Sechrest, “Virtual Memory versus File Interface for Large, Memory-Intensive Scientific Applications,” in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. Pittsburgh: IEEE, 1996. [Online]. Available: <https://ieeexplore-ieee-org.aft.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=1392923>

36. T. Pasquier, D. Eyers, and J. Bacon, "PHP2Uni: Building unikernels using scripting language transpilation," in *Proceedings - 2017 IEEE International Conference on Cloud Engineering, IC2E 2017*. Institute of Electrical and Electronics Engineers Inc., 5 2017, pp. 197–203.
37. M. Pedram and A. Shafaei, "Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures," *IEEE Circuits and Systems Magazine*, vol. 16, no. 2, pp. 62–74, 4 2016.
38. J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 8 2018. [Online]. Available: <http://arxiv.org/abs/1801.00862><http://dx.doi.org/10.22331/q-2018-08-06-79><https://quantum-journal.org/papers/q-2018-08-06-79/>
39. M. Sandberg, M. R. Vissers, T. A. Ohki, J. Gao, J. Aumentado, M. Weides, and D. P. Pappas, "Radiation-suppressed superconducting quantum bit in a planar geometry," *Applied Physics Letters*, vol. 102, no. 7, 2 2013.
40. A. Shafaei, M. Saeedi, and M. Pedram, "Qubit placement to minimize communication overhead in 2D quantum architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2014.
41. N. Sharma, T. K. Rawat, H. Parthasarathy, and K. Gautam, "Performance analysis of quantum unitary gates in presence of noise in the field of quantum communication," in *India International Conference on Power Electronics, IICPE*, vol. 2016-November. IEEE Computer Society, 6 2016.
42. B. Shneiderman, *Designing the User Interface*, 3rd ed. Reading, Massachusetts: Addison Wesley Longman, 1998.

43. P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press, 2002, pp. 124–134. [Online]. Available: <http://ieeexplore.ieee.org/document/365700/>
44. M. Y. Simmons, "Atomic qubits in silicon," in *2019 Silicon Nanoelectronics Workshop, SNW 2019*. Institute of Electrical and Electronics Engineers Inc., 6 2019.
45. D. R. Simon, "On the Power of Quantum Computation * Quantum Probability computation and," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 116–123, 1994.
46. M. Sipser, *Introduction to the Theory of Computation*, 2nd ed., <https://dl.acm.org/doi/book/10.5555/524279>, Ed. Cambridge, MA: International Thomson Publishing, 1996.
47. M. Y. Siraichi, S. Collange, V. F. Dos Santos, and F. M. Q. Pereira, "Qubit allocation," *CGO 2018 - Proceedings of the 2018 International Symposium on Code Generation and Optimization*, vol. 2018-Febru, pp. 113–125, 2018.
48. K. N. Smith and M. A. Thornton, "Automated Mapping Methods for the IBM Transmon Devices," Dallas, Texas, 2018.
49. M. Suchara, J. Kubiawicz, A. Faruque, F. T. Chong, C.-Y. Lai, and G. Paz, "QuRE: The Quantum Resource Estimator toolbox," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 10 2013, pp. 419–426. [Online]. Available: <http://arxiv.org/abs/1312.2316><http://dx.doi.org/10.1109/ICCD.2013.6657074><http://ieeexplore.ieee.org/document/6657074/>
50. S. S. Tannu and M. K. Qureshi, "Not All Qubits Are Created Equal," in *Proceedings of the Twenty-Fourth International Conference on Architectural*

Support for Programming Languages and Operating Systems - ASPLOS '19. New York, New York, USA: ACM Press, 2019, pp. 987–999. [Online]. Available: <https://doi.org/10.1145/3297858.3304007><http://dl.acm.org/citation.cfm?doid=3297858.3304007>

51. W. K. Wootters and W. H. Zurek, “A single quantum cannot be cloned,” *Nature*, vol. 299, no. 5886, pp. 802–803, 10 1982. [Online]. Available: <http://www.nature.com/articles/299802a0>
52. Y. Zhang and Q. Yuan, “A multiple bits error correction method based on cyclic redundancy check codes,” in *International Conference on Signal Processing Proceedings, ICSP*, 2008, pp. 1808–1810.
53. A. Zulehner, A. Paller, and R. Wille, “Efficient mapping of quantum circuits to the IBM QX architectures,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 3 2018, pp. 1135–1138. [Online]. Available: http://www.jku.at/iic/eda/ibm_qx_mapping<http://ieeexplore.ieee.org/document/8342181/>
54. A. Zulehner and R. Wille, “Compiling $SU(4)$ quantum circuits to IBM QX architectures,” in *Proceedings of the 24th Asia and South Pacific Design Automation Conference on - ASPDAC '19*. New York, New York, USA: ACM Press, 8 2019, pp. 185–190. [Online]. Available: <https://arxiv.org/abs/1808.05661><http://dl.acm.org/citation.cfm?doid=3287624.3287704>

Acronyms

CPB Cooper Pair Box. 26

IQT IBM QX Transpiler. 29

NISQ Noisy Intermediate-Scale Quantum. 1, 31

NMR Nuclear Magnetic Resonance. 33

ORM Object-Relational Mapper. 86

QC Quantum Computer. viii, 18, 19, 21, 22, 24, 25, 26, 27, 32, 33, 34, 37, 40, 75, 85

QFT Quantum Fourier Transform. 57, 58, 96, 97

QLP Quantum Layout Problem. vii, 4, 5, 6, 7, 8, 9, 32, 33, 34, 38, 39, 41, 43, 47,
71, 73, 76, 83, 85, 87

QLP-TB Quantum Layout Problem Testbed. vii, ix, 54, 55, 58, 62, 63, 64, 65, 66,
67, 69, 70, 71, 73, 74, 76, 78, 79, 82, 85, 86, 87, 93, 94, 97, 98

QLS Quantum Layout Solver. vii, ix, 8, 39, 50, 56, 57, 58, 76, 82, 84, 85, 90, 94, 96

V2P Virtual-to-Physical. 4, 5, 6, 33, 34, 36, 59, 60

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 26-03-2020		2. REPORT TYPE Master's Thesis			3. DATES COVERED (From — To) Sept 2018 — Mar 2020	
4. TITLE AND SUBTITLE <div style="text-align: center;">Quantum Transpiler Optimization: On the Development, Implementation, and Use of a Quantum Research Testbed</div>				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
				5d. PROJECT NUMBER		
6. AUTHOR(S) Brandon K Kamaka				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765					8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-20-M-029	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL Quantum Science & Technology 525 Brooks Rd. Building 3, Suite H6-2 Rome NY 13441 DSN 587-2504, COMM 315-330-2504 Email: afrl.ritc@us.af.mil					10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RITQ	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT Quantum computing research is at the cusp of a paradigm shift. As the complexity of quantum systems increases, so does the complexity of research procedures for creating and testing layers of the quantum software stack. However, the tools used to perform these tasks have not experienced the increase in capability required to effectively handle the development burdens involved. This case is made particularly clear in the context of IBM QX Transpiler optimization algorithms and functions. IBM QX systems use the Qiskit library to create, transform, and execute quantum circuits. As coherence times and hardware qubit counts increase and qubit topologies become more complex, so does orchestration of qubit mapping and qubit state movement across these topologies. The transpiler framework used to create and test improved algorithms have not kept pace. A testbed is proposed to provide abstractions to create and test transpiler routines. The development process is analyzed and implemented, from design principles through requirements analysis and verification testing. Additionally, limitations of existing transpiler algorithms are identified and initial results are provided that suggest more effective algorithms for qubit mapping and state movement.						
15. SUBJECT TERMS Quantum Computing, Transpiler Optimization, IBM QX, Qiskit, qisquick						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT <div style="text-align: center;">UU</div>		18. NUMBER OF PAGES <div style="text-align: center;">136</div>	
a. REPORT	b. ABSTRACT	c. THIS PAGE				
U	U	U	19a. NAME OF RESPONSIBLE PERSON Laurence D Merkle, AFIT/ENG			
				19b. TELEPHONE NUMBER (include area code) (312) 785-3636 x4526 Laurence.Merkle@afit.edu		