Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers

Prakash Murali*
Princeton University

Jonathan M. Baker
University of Chicago

Ali Javadi Abhari
IBM T. J. Watson Research Center

Frederic T. Chong
University of Chicago

Margaret Martonosi
Princeton University

Abstract

A massive gap exists between current quantum computing (QC) prototypes, and the size and scale required for many proposed QC algorithms. Current QC implementations are prone to noise and variability which affect their reliability, and yet with less than 80 quantum bits (qubits) total, they are too resource-constrained to implement error correction. The term Noisy Intermediate-Scale Quantum (NISQ) refers to these current and near-term systems of 1000 qubits or less. Given NISQ's severe resource constraints, low reliability, and high variability in physical characteristics such as coherence time or error rates, it is of pressing importance to map computations onto them in ways that use resources efficiently and maximize the likelihood of successful runs.

This paper proposes and evaluates backend compiler approaches to map and optimize high-level QC programs to execute with high reliability on NISQ systems with diverse hardware characteristics. Our techniques all start from an LLVM intermediate representation of the quantum program (such as would be generated from high-level QC languages like Scaffold) and generate QC executables runnable on the IBM Q public QC machine. We then use this framework to implement and evaluate several optimal and heuristic mapping methods. These methods vary in how they account for the availability of dynamic machine calibration data, the relative importance of various noise parameters, the different possible routing strategies, and the relative importance of compile-time scalability versus runtime success. Using real-system measurements, we show that fine grained spatial and temporal variations in hardware parameters can be exploited to obtain an average 2.9x (and up to 18x) improvement in program success rate over the industry standard IBM Qiskit compiler. Despite small qubit counts, NISQ systems will soon be large enough to demonstrate "quantum supremacy," i.e., an advantage over classical computing. Tools like ours provide significant improvements in program reliability and execution time, and offer high leverage in accelerating progress towards quantum supremacy.

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

Keywords noise-adaptive compilation; qubit mapping

ACM Reference Format:
Prakash Murali, Jonathan M. Baker, Ali Javadi Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304075

1 Introduction

Quantum computing (QC) aims to solve intractable computational problems by leveraging quantum mechanical principles like superposition and entanglement to manipulate information efficiently. QC algorithms show potential to significantly impact areas such as quantum chemistry [31, 50], cryptography [61], machine learning [4], and others. Unfortunately, a massive gap exists between the resources required by most proposed QC algorithms, and the resources which exist in current prototype hardware.

QC systems have been announced with 49-72 qubits [25, 26, 32] and current operational systems have been demonstrated publicly with roughly 20 qubits or fewer [28]. A QC system with 72 fully-entangled qubits and sufficiently-precise operations ("gates") would likely be sufficient to show "quantum advantage" over the largest classical supercomputers, but would still be 5-6 orders of magnitude smaller than the resource requirements of Shor's well-known QC algorithm for factoring large numbers [14, 55, 61].

The term Noisy Intermediate-Scale Quantum (NISQ) computers refers to the current and near-term QC systems which

*Prakash Murali is the corresponding author and can be reached at pmurali@cs.princeton.edu.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304075

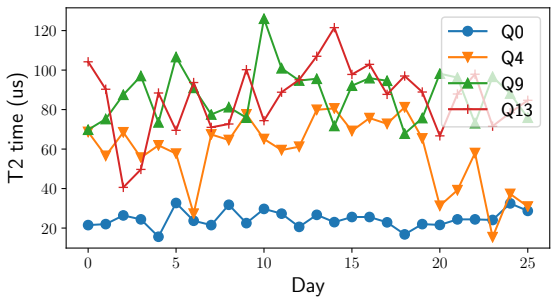
have roughly 1000 qubits or fewer—typically too small to employ error correction codes (ECC) [52]. While resource constrained, NISQ machines offer an important step forward: if used well, they can demonstrate QC applications generating useful results. Making good use of NISQ hardware, however, requires very efficient, near-optimal mappings of algorithms onto them. This paper proposes a suite of optimization- and heuristic-based approaches for mapping applications onto NISQ hardware, and evaluates them by running the mapped executables on a public 16-qubit IBM system[1].

A good mapping of a QC algorithm onto NISQ hardware requires first an intelligent initial placement of the program qubits onto the hardware qubits in order to reduce communication requirements. Second, it requires efficient orchestration of operations both for the computation itself, and also for the additional SWAP operations which communicate state between hardware qubits. Third and most importantly, mapping decisions must reduce the likelihood of operational or decoherence errors which cause the program run to fail to achieve a useful answer. Our work performs mappings using the daily calibration data provided by IBM in order to avoid using unreliable qubits and to prioritize qubit positioning which reduces the likelihood of communication (SWAP) errors. For example, Figure 1 shows large daily variations in the gate error rates and coherence times of the qubits of *IBMQ16* on which we experiment. Our contributions are:

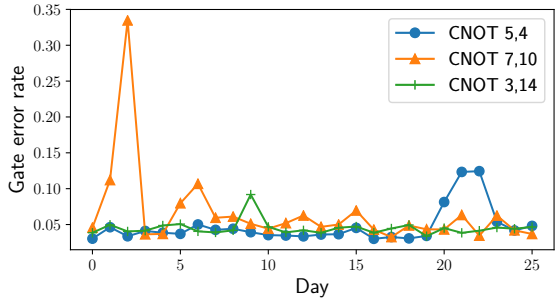
First, we develop an LLVM [36] compiler which optimally or near-optimally maps quantum programs to OpenQASM assembly code [10] and then to the web-accessible *IBMQ16* machine for real-system evaluation. For 12 QC programs written in the Scaffold quantum programming language [1], we use this framework to explore how optimal and heuristic mapping methods, qubit movement policies, and the intelligent adaptation to machine calibration data can affect the quality of the compiled code.

In particular, our compiler provides up to 1.68x gain in execution time and 9x gain in success rate over an optimal but calibration-unaware baseline. Our compiler obtains an average 2.9x improvement (up to 18x) in success rate, and an average 2.7x improvement in execution time (up to 6x), compared to the IBM Qiskit compiler [27], which is the industry standard for *IBMQ16*.

Furthermore, although compile-time is not a first-order design goal, QC compilers must scale well enough for intelligent compilation to be tractable throughout NISQ-range machines. We show that our methods based on Satisfiability Modulo Theory (SMT) scale well up to 32 qubits. Further, we have developed calibration-aware heuristic methods which produce executables with similar reliability and execution time as the SMT approaches, but with more scalable compile-times beyond 32 qubits.



(a) Coherence time (T2)



(b) CNOT gate error rate

Figure 1. Daily variations in qubit coherence time (larger is better) and gate error rates (lower is better) for selected elements in *IBMQ 16 Rueschlikon*. The qubits and gates that are most or least reliable are different across days.

Finally, across the 12 benchmarks, we study the influence of application instruction mix and time varying qubit error characteristics on compiled programs. For example, applications for which our compiler can identify zero-qubit-movement mappings have substantially higher likelihood of success (up to 2.8x), compared to programs which require even a single qubit movement operation.

Overall, NISQ systems are important to QC progress because their success in demonstrating quantum supremacy and running small but useful QC programs is an important stepping-stone in the maturation of this technology. In its leveraging of intelligent and calibration-aware mapping techniques to significantly improve execution time and success rate of quantum executions, our tool makes an important contribution in helping close the gap to quantum supremacy and advancing toward practical QC.

2 Background on Quantum Computing

Principles of Quantum Computing: A qubit is the basic unit of quantum information. Unlike classical bits, which take two values (0 and 1), superposition allows qubits to be in a probabilistic combination of the two states. If we

[1]We run all experiments on the 16-qubit IBM instance named *IBMQ 16 Rueschlikon* [28]. For the remainder of the paper, we shorten this name to *IBMQ16*.

consider the states $|0\rangle$ and $|1\rangle$ as basis vectors of $\mathbb{C}^2$, we can express the state of a qubit $|\psi\rangle$ as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex amplitudes such that $|\alpha|^2 + |\beta|^2 = 1$. The state of one or more qubits can be manipulated by modifying the complex amplitudes using operations termed as gates. Single-qubit operations include: H, X, Y, Z and others. The act of measurement or readout collapses the superposition state to one of the two basis vectors, a classical output.

A controlled NOT (CNOT) gate is an example of a two-qubit gate. A CNOT gate has a control and target qubit. When the control qubit is in the state $|1\rangle$, the state of the target bit is flipped. In quantum CNOT gates, the gate can operate on qubits to entangle them to have non-classical correlations in their states and measurement outputs. We use the notation `CNOT C, T` for a CNOT gate with control C and target T.

A quantum computer with n fully-entangled qubits has an exponential state space of size 2^n. In a QC application, a set of qubits are initialized to encode a given problem including its data input. As the program executes, qubit amplitudes are manipulated, typically to boost the probabilities of the desired outcomes in the state space. Finally, the qubits are measured to produce classical output for the given problem.

NISQ Systems: NISQ systems are near-term quantum systems expected to scale to a few hundred qubits, paving the way towards large-scale QC [52]. Qubits in NISQ systems have short coherence time, high gate error rates and and limited qubit connectivity. They are typically too resource-constrained to implement error-correcting codes (ECC).

As a concrete NISQ example, Figure 2b shows the layout of the qubits in the 16-qubit IBM system. This system implements a set of 1- and 2-qubit operations, akin to an instruction set. For 2-qubit operations, this machine only supports hardware CNOT gates being performed between *adjacent* qubits, based on the topology shown in Figure 2b. To perform CNOT gates between non-adjacent qubits, we should use SWAP operations between adjacent qubits until the two of interest for a given CNOT computation are in adjacent locations. Each SWAP operation between two adjacent qubits itself requires 3 CNOT gates[2] Our compiler aims to reduce the *time cost* of these operations. More importantly, each one of these operations incurs some error, so a key goal of our optimization is to reduce operation counts and error rates in order to increase the likelihood of an overall successful run. We refer to this as *reliability* and it is the primary design goal of this work.

In addition to compiler optimization based on attributes like gate counts, our approach also adapts based on publicly-available experimental data. In particular, the IBM Q machines are calibrated twice a day. Once a day there are public postings of experimental measurements of key properties: qubit relaxation time (T1), coherence time (T2), gate errors and readout errors [29]. From daily calibration logs, we observe that qubit coherence time is 70 microseconds on average, but varies up to 9.2x spatially and temporally across qubits and daily calibrations. The average error rate for CNOTs is 0.04, readouts is 0.07 and single qubit gates is 0.002. CNOT and readout error rates exhibit up to 9.0x and 5.9x variation across qubits and calibration cycles, respectively. CNOT gate durations vary up to 1.8x across qubits. These fluctuations stem from material defects caused by the lithographic processes used to manufacture the qubits and are expected to be present in future generations of superconducting qubits also [33].

These error rates imply only very short programs can execute reliably on the machine. A program with more than 16 CNOT operations, has less than 50% chance of executing correctly. A key goal of our compiler optimizations is to use this calibration data to boost the success rate of individual program runs, by avoiding portions of the machine with poor coherence, operation, or readout errors.

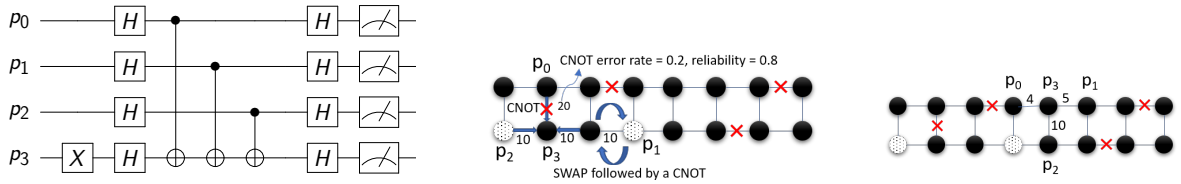
3 Compilation Framework: Overview

Our framework takes a Scaffold program [1] as input, and produces compiled OpenQASM code [10]. The Scaffold quantum programming language extends C with quantum gates. Scaffold programs are independent of the machine topology, size and qubit properties. The ScaffCC compiler [30, 58] performs automatic gate and rotation decomposition, implements high level operations like the Toffoli gate and produces an LLVM Intermediate Representation (IR) [36] of the program. The IR version of the program includes the qubits required for each operation and the data dependencies between operations. For example, Figure 2a shows the IR for the simple 4-qubit Bernstein-Vazirani algorithm which is chosen because it fits on machines of this size and has an answer which can be calculated to check our results [3]. We use the program IR as a starting point for the noise-aware backend described here.

Starting from the IR, the noise-aware backend has three primary tasks. First, qubits in the program must be **mapped** to distinct qubits in the hardware implementation, preferably in a way that reduces qubit state movement required as the program executes. Second, the compiler performs **operation scheduling** while respecting data dependencies between gates. To accomplish this, each operation is assigned a start time constraint, and the scheduler emits control code that enforces this. Third, to perform 2-qubit operations on non-adjacent qubits, the compiler should orchestrate **communication through SWAPs**. That is, it automatically inserts the required SWAP operations to bring the qubits adjacent to each other before the operation is performed.

Consider a simple compilation method where program qubits are assigned to random qubits on the hardware. Figure 2b shows such a mapping for the BV4 IR. In this mapping, the

[2]For two qubits X and Y, `SWAP(X,Y) := {CNOT X,Y; CNOT Y,X; CNOT X,Y}` [41].



(a) Bernstein-Vazirani Intermediate Representation

(b) Layout of qubits in *IBMQ16* and a naive mapping for BV4.

(c) Optimized mapping for BV4

Figure 2. Figure (a) shows the intermediate representation of the Bernstein-Vazirani algorithm on 4 qubits (BV4). Each program qubit is represented by a line. X and H are single qubit gates. The CNOT gates from each qubit $p_{0,1,2}$ to p_3 are marked by vertical connectors. The measurement or readout operation is indicated by the meter. Figure (b) shows the layout of the hardware qubits in *IBMQ16* and a naive mapping of BV4's program qubits. The black circles denote qubits and the edges indicate permitted CNOT gates. The numbers on the labelled edges indicate the CNOT gate error ($\times 10^{-2}$). The hatched qubits and crossed gates are unreliable. In this mapping, qubit movement is required to perform the CNOTs and error-prone operations are used. Figure (c) shows a mapping where qubit movement is not required and unreliable qubits and gates are avoided.

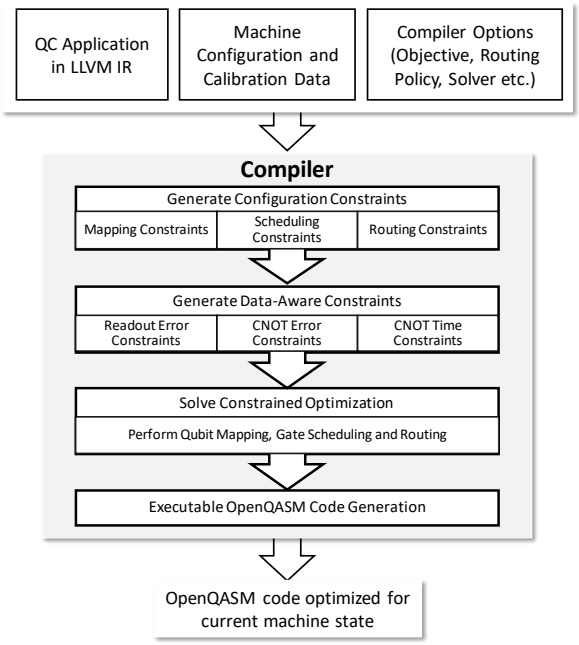


Figure 3. Optimization Pipeline. Inputs are a QC program, details about the specific hardware configuration, and a set of options, such as routing policy and solver approach. From these, compiler generates a set of appropriate constraints and uses them to map program qubits to hardware qubits and schedule operations. Finally, the compiler generates an executable version of the program, here for *IBMQ16*.

compiler must insert qubit movement or swap operations to perform the CNOT gates between p_1 and p_3. In contrast, the mapping shown in Figure 2c requires no qubit movement because the qubits required for the CNOTs are adjacent. In addition, this mapping is noise-aware; namely, it uses the calibration data to select a mapping that avoids using qubits with low coherence time and gates with high error rates. Our compiler uses machine topology and calibration data to automatically generate such mappings for a given program.

Our primary goal is to maximize the likelihood that the program runs successfully. To accomplish this, we have three main strategies. First, the compiler places program qubits on hardware locations with high reliability, based on the calibration data. The compiler considers the effect of errors due to CNOTs and readouts; for this machine, single-qubit error rates are considerably smaller so our formulation chooses to ignore them. Second, to mitigate errors due to decoherence, the compiler should schedule all gates to finish before the coherence time of the hardware qubits (intuitively analogous to making use of data within the refresh interval of a DRAM). Third, the compiler optimizes for the qubit topology to avoid unnecessary qubit movement. Qubit movement not only increases execution duration, but more importantly leads to high error rates since each qubit SWAP operation includes three error-prone CNOTs. We have designed a set of optimal and heuristic compilation variants to accomplish these goals.

Table 1 enumerates the full set of compiler variants we consider in this paper. In addition to the publicly-available IBM Qiskit compiler we use as a comparative baseline, we also develop several approaches which are either truly optimization-based or heuristic. We give an overview of these approaches here, before offering details in the following section.

3.1 Optimization-Based Mappings

In the optimization-based variants of our compiler, we implement the above goals by posing the compilation problem as a constrained optimization problem to be solved by a satisfiability modulo theory (SMT) solver. The optimization problem has variables and constraints which express program information, machine topology constraints, and machine error information. The variables include program qubit locations,

Algorithm	Objective	Parameters	Constraints
Qiskit	Heuristic, minimize duration	-	-
T-SMT	SMT solver, minimize duration	Routing policy: RR, 1BP	1-4, 7-9
T-SMT★	SMT solver, minimize duration	Routing policy: RR, 1BP	1-3, 5-9
R-SMT★	SMT solver, maximize reliability	Routing policy: 1BP Readout weight $\omega \in [0, 1]$	1-3, 5-6, 9, 10-11
GreedyV★	Heuristic, maximize reliability	Routing Policy: Best Path	-
GreedyE★	Heuristic, maximize reliability	Routing Policy: Best Path	-

Table 1. List of compiler configurations used in our study. The IBM Qiskit 0.5.7 compiler is used as a the baseline. The use of calibration data is marked by a ★.

gate start times and routing paths. The constraints specify qubit mappings should be distinct, gates should start in program dependency order, and routing paths should be non-overlapping. Fig. 3 summarizes the general compilation pipeline for the solver-based approach, beginning with an IR of a program and resulting in execution-ready code.

The optimization objective is to maximize the reliability or success rate of program runs. We express the reliability of the program as the product of the reliability of all gates in the program. (Because of the degree of entanglement in QC programs, this serves as a useful measure of overall correctness.) For a given mapping, the solver determines the reliability of each program CNOT, readout operation and single qubit gate and computes an overall reliability score. For the optimization variants which are noise-aware, the solver can maximize the reliability score over all mappings by tracking and adapting to the error rates, coherence limits, and qubit movement based on program qubit locations.

Given a target machine, our framework converts the program IR into an optimization problem by expressing an objective and constraints that can be solved using an Satisfiability Modulo Theory (SMT) solver [5, 11]. For classical programs, these solvers have been used to obtain optimal hardware mapping and scheduling for spatial architectures [43], but to our knowledge, ours is the first use of them for QC systems. SMT solvers take as input a set of linear constraints, and an objective function and search for an optimal solution. Although the reliability objective is a product of individual gate reliability scores (and therefore non-linear), we linearize the objective by instead optimizing for the additive logarithms of the reliability scores. An SMT solver can then be invoked to find a mapping which maximizes the log reliability.

Does maximizing the reliability score achieve our goal of increasing program success rate? Optimizing for the reliability score induces the compiler to place qubits at locations where CNOT and readout errors are low. It also indirectly minimizes qubit movement because CNOTs between far away qubits are error-prone. For example, for the BV4 IR, consider mapping shown in Figure 2b. Here, the reliability of the CNOT between p_0 and p_3 is 0.8 (80% chance of executing correctly), while the reliability of the CNOT between p_1 and p_3 is only 0.65[3]. Thus, the compiler will choose mappings where communicating qubits are close together, minimizing unnecessary qubit movement and allowing gates to be scheduled to finish within the coherence window.

3.2 Heuristic Mappings

We also determine whether heuristic techniques can approach the optimization-based results, but with better scalability. For this, we develop two comparative algorithms based on greedy heuristics. The greedy heuristics analyze the CNOTs in the program IR, and determine a gate frequency for each qubit and program CNOT.

We explore two policies. In the first policy, GreedyV★, we place program qubits on hardware qubits in the heaviest qubit first order. In the second policy, GreedyE★, we place program CNOTs and their control and target qubits in a heaviest edge first order. Intuitively, the first policy places qubits which use more CNOTs in locations which have good CNOT and readout error rates. The second policy places pairs of qubits which have the most frequent CNOTs first.

4 Optimal Compilation

4.1 Notations and Assumptions

Let Q_P be the set of program qubits. Let Q_H be the set of hardware qubits. In this work, we assume hardware qubits are arranged as a 2-D grid of dimensions $M_x \times M_y$. Likewise, due to the connectivity characteristics of *IBMQ16*, we assume only hardware qubits which are adjacent in the grid are permitted to participate in two qubit operations. More elaborate topology and routing assumptions can be handled in future work. For $q \in Q_P$, the ordered pair $(q.x, q.y)$ corresponds to the location of the hardware qubit assigned to the program qubit q. Let G be the set of operations in the program. This includes single-qubit gates such as H, and the 2-qubit $CNOT$ gate and qubit measurement or *Readout* operations. CNOT and readout operations dominate the reliability outcomes, so the reliability score focuses on them. The subset of CNOT

[3] p_1 has to swap once to move to a location adjacent to p_3. The net reliability of the 3 CNOTs required to perform the SWAP is $0.9^3 = 0.729$. Then the actual CNOT operation can be performed with reliability 0.9. Hence, the overall CNOT reliability is 0.65.

gates is denoted by G_{CNOT}, and the subset of readout operations is $G_{Readout}$. For each gate g in the program, the start time is denoted by $(g.\tau)$, duration by $(g.\delta)$, and reliability by $(g.\epsilon)$. To denote data dependencies between the operations, we use a binary relation $>$ on the gates, so that for two operations $g_2 > g_1$ if g_2 depends on g_1. Although the reliability objective focuses on a subset of operations, we map and schedule all operations (including single-qubit operations) to provide a valid real-system executable.

4.2 Constraints

Qubit Mapping Constraints: Constraint 1, guarantees all program qubits are mapped to actual hardware qubits. Constraint 2 guarantees each program qubit is assigned a unique location.

$$\forall q \in Q_P : 0 \leq q.x < M_x \wedge 0 \leq q.y < M_y \quad (1)$$

$$\forall q_1, q_2 \in Q_P : q_1.x \neq q_2.x \vee q_1.y \neq q_2.y \quad (2)$$

Gate Scheduling Constraints: For each gate g in the program, the compiler determines the start time and execution duration. If two gates g_1 and g_2 both operate on the same qubit, and g_2 uses the output of g_1, g_2 should start only after g_1 finishes. For every such edge in the dependency graph, Constraint 3 shows the form we use to enforce such data dependencies.

$$\forall g_1, g_2 \in G : g_2 > g_1 \Rightarrow g_2.\tau \geq g_1.\tau + g_1.\delta \quad (3)$$

The durations, δ, for single qubit operations are set using the documented durations in timeslots of the corresponding hardware operations. For CNOTs, the duration includes both the operation itself as well as the time to bring the relevant program qubit states into adjacent hardware qubits; this depends on the routing policy and is discussed below.

CNOT Duration based on Grid Distance: The duration of a CNOT gate accounts for both CNOT time and the duration of the swap paths before and after the CNOT. For a CNOT $g \in G_{CNOT}$, let the control and target qubits be q_c and q_t. Then the duration of the CNOT is: $g.\delta = 2 * (\|q_c - q_t\|_1 - 1) * \tau_{SWAP} + \tau_{CNOT}$ where $\|q_c - q_t\|_1 = |q_c.x - q_t.x| + |q_c.y - q_t.y|$ and τ_{SWAP}, τ_{CNOT} are the times to complete a SWAP or CNOT operation, respectively.

The compiler must schedule operations before the individual qubits decohere. For T-SMT (noise-unaware) we simply use an assumption of M_T as 1000 timeslots of coherence time, which is the long-term average for the machine:

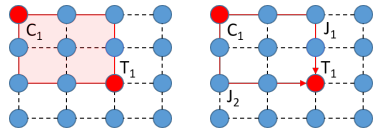
$$\forall g \in G : g.\tau + g.\delta < M_T \quad (4)$$

CNOT Duration based on Calibration Data: For T-SMT* and R-SMT*, we set durations based on calibration data. In particular, since qubit coherence time changes daily (Figure 1a) and CNOT gate durations vary across qubits, these approaches use the calibration-based data in the optimization constraint. To set durations based on calibration data, we assume a routing policy and compute the CNOT durations for each hardware qubit pair. Let Δ be an $|Q_H| \times |Q_H|$ matrix where Δ_{h_i,h_j}, $i \neq j$, specifies the duration of a CNOT between hardware qubits $h_i, h_j \in Q_H$. The duration of a program CNOT can be set as: for all $g \in G_{CNOT}$ and for all $h_1, h_2 \in Q_H$:

$$g_c = h_1 \wedge g_t = h_2 \Rightarrow g.\delta = \Delta_{h_1,h_2} \quad (5)$$

For the calibration-aware coherence time bound, constraint 6 ensures every gate finishes before the coherence time of the qubits it acts on i.e., if a gate uses a hardware qubit h, it should complete before h decoheres, with $h.\tau$ as the coherence time of a hardware qubit $h \in Q_H$. We have for all $g \in G$ and for all $h_1, h_2 \in Q_H$:

$$g_c = h_1 \wedge g_t = g_2 \Rightarrow g.\tau + g.\delta \leq \min(h_1.\tau, h_2.\tau) \quad (6)$$



(a) Rectangle Reservation (RR) **(b)** One Bend Paths (1BP)

Figure 4. Two routing policies for swap-based architectures.

4.3 Routing for CNOT Gates

To route multiple CNOTs in parallel, the compiler uses two routing policies based on policies in VLSI routing [19, 20].

Rectangle Reservation: In this policy, for every CNOT in the program, the compiler blocks a 2D region bounded by the control and target qubit, during the CNOT execution. For example, in Figure 4a, the highlighted rectangle is reserved for the duration of the CNOT.

Consider a CNOT gate $g_i \in G_{CNOT}$. Let (l_x^i, l_y^i) and (r_x^i, r_y^i) denote the top left and bottom right corners, respectively, of the bounding rectangle of g_i. These variables are defined using min and max relations on the qubit mapping variables of the CNOT. For two CNOTs g_i and g_j, the routing constraint is:

$$S(R_i, R_j) = \neg(l_x^i > r_x^j \vee r_x^i < l_x^j \vee l_y^i > r_y^j \vee r_y^i < l_y^j) \quad (7)$$

$$T(g_i, g_j) = \neg(g_i.\tau > g_j.\tau + g_j.\delta \vee g_j.\tau > g_i.\tau + g_i.\delta) \quad (8)$$

Constraint S checks if the two rectangles overlap in space. Constraint T checks whether CNOTs overlap in time. For any pair of CNOTs g_i and g_j, they cannot overlap in time if they overlap in space: $S(g_i, g_j) \implies \neg T(g_i, g_j)$.

One Bend Paths: In this policy, CNOT routes are restricted to the two paths along the bounding rectangle of the control and target qubit. For example, in Figure 4b, the CNOT is allowed to use one of the two highlighted paths. To implement this policy, the solver selects one of the two routes for every CNOT in the program.

To express constraints for this policy, we use variables to record the junction through which the CNOT is routed. The one bend path is composed of two segments: control to junction and junction to target. For generality, we can consider these segments as rectangles, and apply the same overlap check as in rectangle reservation. Denote the control to junction path for CNOT i as R_i^{cj}. Then, we can check if two CNOTs g_i and g_j overlap using:

$$Overlap(i,j) = S(R_i^{cj}, R_j^{cj}) \vee S(R_i^{cj}, R_j^{jt}) \vee S(R_i^{jt}, R_j^{cj}) \vee S(R_i^{jt}, R_j^{jt}) \quad (9)$$

Similar to rectangle reservation, we impose the condition that CNOTs do not overlap in time if they overlap in space.

4.4 Reliability Constraints

To optimize the reliability of program executions, we use a set of constraints to track the reliability scores of CNOT and readout operations in the program. Let $g.\epsilon$ denote the reliability score for the operation g. For readout operations, we set the reliability as

$$\forall g \in G_{Readout} : \forall h \in Q_H : g.q = h \Rightarrow g.\epsilon = E_h^R \quad (10)$$

where E_h^R is the reliability score for readout operations on hardware qubit h, and $G_R \subseteq G$ is the set of readout operations.

In R-SMT* we perform reliability optimization using the one bend paths routing policy. Under this policy, for CNOT gate, we set reliability tracking variables based on the junction used for routing. For each pair of hardware qubits, we compute the reliability of the two possible paths, and store them in a matrix E^C, indexed by the hardware qubits and junction. This reliability factors in the reliability of the swap paths through the junction and the actual CNOT operation. Let $g.j$ be the junction for gate $g \in G_{CNOT}$. The constraints to track CNOT error are given for all $g \in G_{CNOT}$ and for all $h_1, h_2, h_j \in Q_H$:

$$g_c = h_1 \wedge g_t = h_2 \wedge g.j = h_j \Rightarrow g.\epsilon = E^C_{h_1,h_2,j} \quad (11)$$

In our experiments, considering the error rates of single qubit gates such as H, X, Y etc. is not required for *IBMQ16*, because their error rates are much smaller than CNOTs and readouts. For systems where such errors matter, they can be easily incorporated into the optimization using similar constraints.

4.5 Optimal Compilation: Objective Function

The different optimization variants use different objective functions. For the time-oriented variants T-SMTand T-SMT*, the objective function is based on the execution time for the program. Using the gate scheduling and duration constraints in Section 4, the objective is to minimize the finish time of the last gate in the dependency order.

For the reliability-oriented variant, R-SMT*, the objective function is based on the reliability of a program execution. We define the reliability of a program execution as the product of the reliability of each of its gates. Since single qubit gates have low error, we define the reliability using CNOT and readout operations only. Ideally, the reliability objective would be the product across all gates of the readout and CNOT errors for the whole program: $\max \prod_{\forall g \in G_{Readout} \cup G_{CNOT}} (g.\epsilon)$. Because the SMT solver requires linear operations, we convert this to an additive linear objective function by considering the logarithm of the operation reliabilities, instead of their product. Finally, to allow for different emphases on readout error versus CNOT error, we convert the above objective into a weighted objective using a weight ω which is applied to the readout error rates:

$$\omega \sum_{g \in G_{Readout}} \log(g.\epsilon) + (1-\omega) \sum_{g \in G_{CNOT}} \log(g.\epsilon). \quad (12)$$

We use this objective to study the relative importance of CNOT and readout error rates.

Optimizing reliability places qubits at hardware locations with high CNOT and readout reliability. It indirectly optimizes qubit movement because CNOT gates between non-adjacent qubits have low reliability. This objective is used by R-SMT* in our experiments. The output of the solver has the optimal reliability with respect to the program and machine model assumptions. Our experiments show that it is also near-optimal in execution duration.

To compute a qubit mapping and gate schedule which maximizes this objective, we set up an optimization problem using this along with the mapping and scheduling constraints, gate durations using calibration data, routing approaches, and reliability constraints discussed before. The reliability constraints make the $g.\epsilon$ variables dependent on the qubit mapping variables.

5 Heuristic Compilation

Where tractable, the SMT-based compilation approach offers the best chance at successful application runs on real hardware. However, effective heuristic approaches may offer similar reliability but scale better to future NISQ systems with hundreds of qubits. Here we propose and evaluate heuristic mapping/scheduling alternatives as comparators to the optimization-based approaches.

Our heuristic techniques are also based on a program graph constructed from the program IR. The program graph has a node for every qubit, and an edge between every pair of qubits which is involved in a CNOT. For example, the program graph of BV4 has 4 nodes for $p_{0,1,2,3}$ and 3 edges, one from each of $p_{0,1,2}$ to p_3. For each heuristic, we first compute the most reliable path between every pair of hardware qubits using Dijkstra's algorithm, where edge weights are given as the negative log of the CNOT errors from the calibration data. For both heuristics, once we map the qubits, we schedule

gates using an earliest ready gate first policy [24] and route based on the precomputed paths.

5.1 Greatest Vertex Degree First

The GreedyV★ heuristic seeks to minimize communication distance (and therefore reduce the number of error-prone SWAP operations) by considering qubits in descending order of degree. The degree of the qubit is the number of CNOTs in which the qubit is used. First, place the highest degree program qubit at the hardware location which has highest readout reliability among high degree hardware qubits. Next, for each program qubit which shares a CNOT with an already placed qubit, place this qubit in order to maximize the total reliability of paths between it and each of its placed neighbors, where the total reliability is given by the sum of the path lengths computed between it and its neighbors.

5.2 Greatest Weighted Edge First

In GreedyE★, we map edges in the descending order of weight. The weight of an edge between two nodes is the number of times a CNOT gate is invoked between them. Therefore, placing edges with high weight first allows qubits which interact highly to be close together. Such placement reduces qubit movement and increases reliability. The algorithm starts by placing the highest weighted edge at on hardware location with maximum CNOT and readout reliability. Next, for each edge which has one mapped one unmapped endpoint, we map the unmapped qubit to the position which maximizes the total reliability of CNOTs with already mapped qubits, where the total reliability is given by the sum of the path lengths computed from before between it and its neighbors. The process is repeated for each unmapped edge in weight order.

6 Experimental Setup

Benchmarks: Table 2 lists 12 quantum programs derived from prior work on compilation and system benchmarking [2, 40, 64]. These benchmarks include the Bernstein-Vazirani algorithm [3], Hidden Shift Algorithm [7], Quantum Fourier Transform [42], a one bit adder and important quantum kernels such as the Toffoli gate [41]. We used or created Scaffold programs for each benchmark and obtained LLVM IR using the ScaffCC compiler [30]. To be runnable on real-system QC hardware, the benchmarks must be relatively small in qubit counts and short in execution time steps. Nonetheless, our ability to show order-of-magnitude improvements in success rate for these programs is a promising indicator of the value of such compilation techniques for future larger systems and programs. Furthermore, several of these programs—such as QFT and Toffoli—are important kernels for larger programs.

Beyond these, to study scalability trends across different qubit and gate counts, we generate a synthetic benchmark where we can specify the number of qubits and gates and from this, we experiment with randomly generated quantum programs with 4-128 qubits and 128-2048 gates. We generate these circuits by uniformly sampling gates from the universal gate set of H, X, Y, Z, S, T, CNOT.

Name	Qubits	Gates	CNOTs	CNOT Graph
BV4	4	12	3	
BV6	6	12	3	
BV8	8	18	3	
HS2	2	16	2	
HS4	4	28	4	
HS6	6	42	6	
Fredkin	3	19	8	
Or	3	17	6	
Peres	3	16	5	
Toffoli	3	18	6	
Adder	4	23	10	
QFT	2	13	5	

Table 2. Characteristics of benchmark programs.

Compiler Configurations: To study various compilation schemes, our framework includes various options for the solver, routing policy, use of calibration data and other parameters. We evaluate these options one factor at a time using the configurations listed in Table 1. We compare R-SMT★and T-SMT★to demonstrate the benefits of noise-adaptive compilation. We compare T-SMT★and T-SMT to demonstrate the importance of considering gate times and coherence times from calibration data.

Experimental Setup: Our compilation experiments use an Intel Skylake processor (2.6GHz, 12GB RAM) using Python3.5 and gcc version 5.4. Our optimization approach uses the Z3 SMT solver [11]. To perform experiments on *IBMQ16*, we use the IBM Quantum Experience APIs [28, 29]. The daily machine calibration data is available through the Quantum Experience APIs. The calibration data includes time data such as single qubit gate time, qubit coherence time (T2 time), durations for CNOT gates, and error rates such as single qubit gate error, CNOT gate error, and read out (measurement) error. We use IBM's Qiskit compiler/mapper as our baseline for comparison, version 0.5.7.

Metrics: Before each run, we obtain the latest calibration data, and recompile the benchmark. We execute each benchmark on *IBMQ16*, using 8192 trials in each run. We measure the success rate as the fraction of trials which gave the correct answer. For example, success rate of 0.6 means the execution produced the correct answer in 60% of the trials. The ideal success rate is 1, where all trials succeed. Results within a single graph are performed closely in time

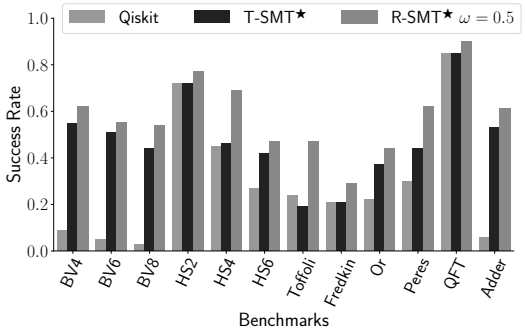


Figure 5. Measured success rate of R-SMT★ compared to Qiskit and T-SMT★. (Of 8192 trials per execution, success rate is the percentage that achieve the correct answer in real-system execution.) R-SMT★ obtains higher success rate than Qiskit because it simultaneously adapts placement according to dynamic error rates and avoids unnecessary qubit movement.

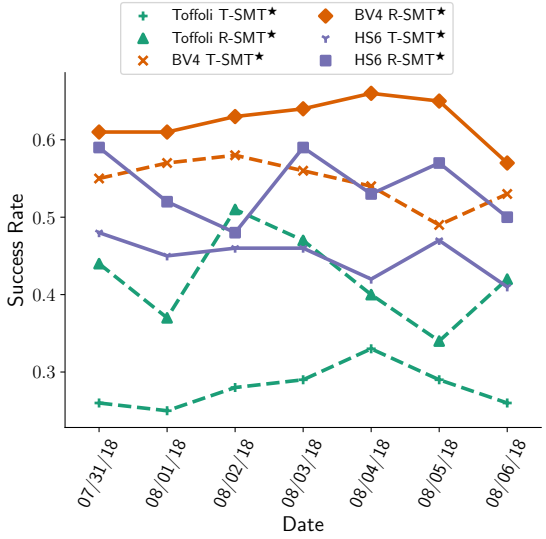


Figure 6. Executions of three benchmarks for 1 week. R-SMT★ is more resilient to errors compared to T-SMT★. Similar trends for other benchmarks.

so are comparable. Results from different graphs may not be comparable because the machine error characteristics can be different across runs. We also study quantum execution time and compilation time. Because timing granularity is so coarse, execution time is estimated using real gate duration data from the *IBMQ16* system. We report durations in terms of timeslots on *IBMQ16*, where each timeslot is 80ns.

7 Optimizing Execution Reliability

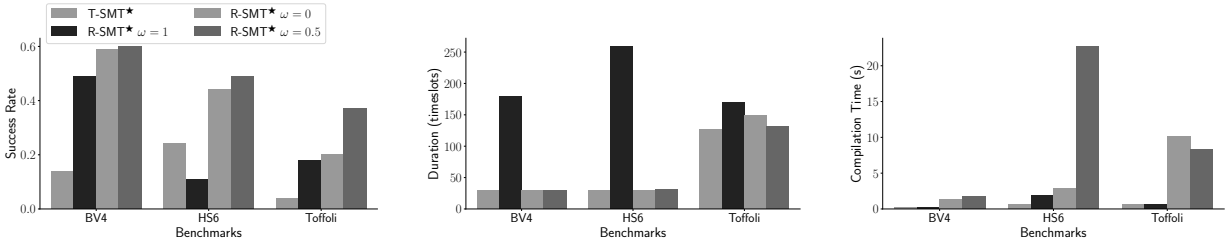
Baseline Comparison to IBM Qiskit: We compare the success rate of program runs from our compiler versus the IBM Qiskit compiler for real-system runs on *IBMQ16*. Figure 5 shows the success rate of the IBM Qiskit compiler, T-SMT★ and R-SMT★ with $\omega = 0.5$ on all the benchmarks. In all benchmarks, R-SMT★ has higher success rate than Qiskit, indicating that its reliability-oriented objective function is effective. In fact, R-SMT★ obtains geomean 2.9x improvement over Qiskit, with up to 18x gain. Figure 8 shows the mapping used by Qiskit, T-SMT★ and R-SMT★ for BV4. Qiskit places qubits in a lexicographic order without considering CNOT and readout errors and incurs extra swap operations. For BV8, the compiled code produced by Qiskit used 15 CNOT operations to move qubits (in addition to the 3 CNOTs required by the algorithm), while R-SMT★ obtains a mapping which require no qubit movement. Each extra CNOT gate increases both the error rate and the execution duration of the code and leads to poor success rate. Benchmarks which require no qubit movement such as BV, HS, QFT and Adder have higher reliability than Toffoli, Fredkin, Or, and Peres, which require at least one qubit swap.

In all benchmarks, R-SMT★ outperforms T-SMT★, even though they use the same number of qubit movement operations. While optimizing qubit communication is important, it is essential to optimize for gate error rates to improve success rate. In fact, in our experiments, when the machine state has high variability, R-SMT★ can obtain up to 9.2x improvement in success rate over T-SMT★ (see Fig. 7 and 8).

Resilience to Daily Variations: Since IBM limits the executions researchers may perform per day, we perform detailed experiments on three benchmarks, BV4, HS6 and Toffoli. These benchmarks are chosen as examples of different CNOT patterns (see Table 2). Figure 6 compares the success rate of R-SMT★ and T-SMT★ over a week for the three benchmarks. The success rate of the programs change every day because error rates of the hardware CNOT and readout units change daily. (We recompile each day before running.) For all three benchmarks, R-SMT★ is more resilient to error than T-SMT★, since it adapts the qubit mappings to account for daily variations in operation error rates. Since T-SMT★ compiles based on static information (qubit topology and gate duration), it uses the same qubits and hardware gates every day, irrespective of their dynamic error characteristics.

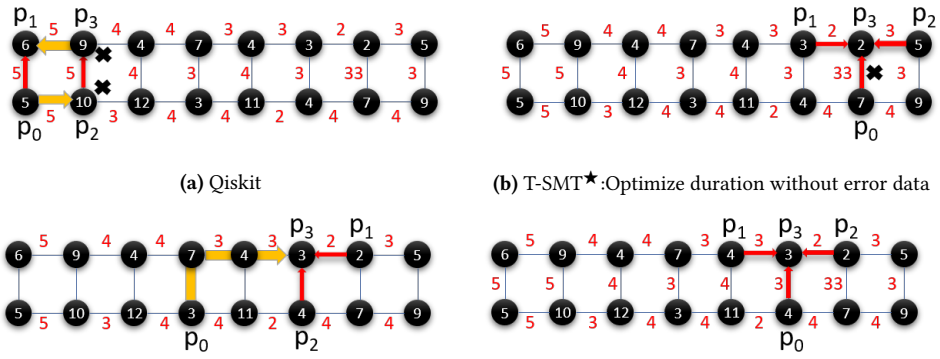
7.1 Choice of Optimization Objective

Figure 7 compares R-SMT★ with $\omega = \{0, 0.5, 1\}$ and T-SMT★ on the three benchmarks. R-SMT★ with $\omega = 0.5$ achieves the highest success rate among the methods, with up to 9.25x gain over T-SMT★. For BV4, we illustrate the mappings obtained by the these methods in Figure 8. T-SMT★ obtains a mapping which requires no qubit movement, but it uses a hardware CNOT with very high error rate. With $\omega = 1$, R-SMT★ optimizes only for readouts and uses long swap paths which reduce success rate. With $\omega = 0.5$, R-SMT★



(a) Success Rate **(b)** Execution Duration **(c)** Compile Time

Figure 7. Measured success rate, execution duration and compile time for three representative benchmarks. T-SMT★ which directly optimizes for execution duration obtains the minimum execution durations, but R-SMT★ with $\omega = 0.5$ is close, and more resilient to errors (higher success rate). All benchmarks compile in less than 1 minute.



(a) Qiskit **(b)** T-SMT★:Optimize duration without error data

(c) R-SMT★($\omega = 1$): Optimize readout reliability **(d)** R-SMT★($\omega = 0.5$): Optimize CNOT+readout reliability

Figure 8. For real data/experiment, on *IBMQ16*, qubit mappings for Qiskit and our compiler with three optimization objectives, varying the type of noise-awareness. In each figure, the edge labels indicate the CNOT gate error rate ($\times 10^{-2}$), and the numbers inside each node indicate that qubit's readout error rate ($\times 10^{-2}$). The thin red arrows indicate CNOT gates. The yellow thick arrows indicate SWAP operations. (a) Qiskit finds a mapping which requires SWAP operations and uses hardware qubits with high readout errors (b), T-SMT★ finds a a mapping which requires no SWAP operations, but it uses an unreliable hardware CNOT between p_3 and p_0. (c) Program qubits are placed on the best readout qubits, but p_0 and p_3 communicate using swaps. (d) R-SMT★ finds a mapping which has the best reliability where the best CNOTs and readout qubits are used. It also requires no SWAP operations.

maps qubits to simultaneously optimize CNOT gate error, readout error and qubit movement.

R-SMT★ with $\omega = 0.5$ also achieves near-optimal execution durations, comparable to T-SMT★, which directly optimizes for duration. From the perspective of compilation time, optimizing for reliability is harder than optimizing execution duration. However, each method finds optimal mappings in under a minute, for each benchmarks.

R-SMT★ was executed with $\omega \in [0, 1]$ to determine the relative importance of optimizing for readout error and CNOT error. In general, choosing an ω roughly near 0.5 is appropriate to obtain good success rates. On the *IBMQ16* machine, readout and CNOT error rates are fairly balanced, and hence we see that an equal weighted combination of both is suitable for optimization.

7.2 Sensitivity to Gate Durations and Coherence Time

We test whether the use of real gate time data significantly affects the execution duration of NISQ benchmarks. Our compiler is run on three settings: T-SMT(RR) which assumes all hardware CNOTs have the same gate duration and T-SMT★ (RR) and R-SMT★ (1BP) which use real gate durations. We restrict R-SMT★ to the 1BP policy to reduce the number of experimental configurations; we show in Section 7.3 that the choice of routing policy doesn't affect execution duration for NISQ benchmarks.

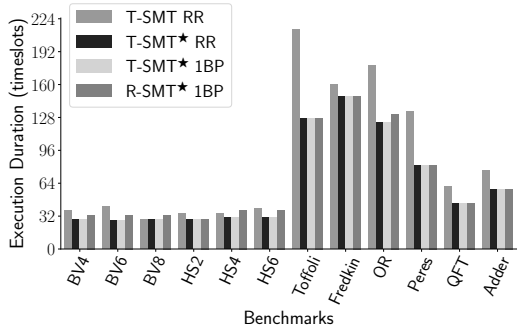


Figure 9. Effect of gate durations, routing policy and objective function on execution duration. Although reliability is our primary objective, several variants perform well on run time as well. T-SMT★(either RR or 1BP) has the best execution duration, but R-SMT★is very close in run time and offers better success rates. Noise-aware policies, R-SMT★and T-SMT★, are 1.6x better than T-SMT.

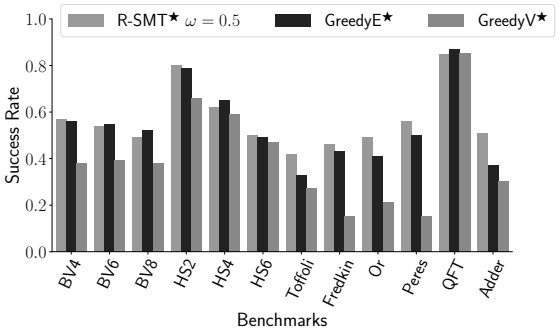


Figure 10. Noise-aware Heuristics: GreedyE★ heuristic mapping offers reliability comparable to R-SMT★on most benchmarks.

Gate Durations: Figure 9 shows execution duration, computed using the gate time data, for the three methods. Considering real gate durations can improve the execution duration for each benchmark, with up to 1.68x gain on Toffoli. Considering real durations increases the number of constraints in the optimization problem and increases the compilation time by up to 3x (not shown). Even with real durations, each benchmark requires only a few seconds of compilation time.

Coherence Time: Each benchmark finishes in less than 150 timeslots using the R-SMT★ method. Since the coherence time of the worst qubit on the machine is more than 300 timeslots, considering fine grained variations in coherence time is not necessary for our benchmarks.

7.3 Effect of Routing Policy

Figure 9 compares the execution duration and compilation time of T-SMT★ with two routing policies (RR and 1BP) and R-SMT★ (1BP). The three policies produce executables with similar execution duration since NISQ benchmarks are small, and have only few parallel CNOTs. Hence, most CNOTs execute without swapping or blocking qubits. Although R-SMT★ optimizes reliability, it obtains execution durations close to T-SMT★ on all benchmarks.

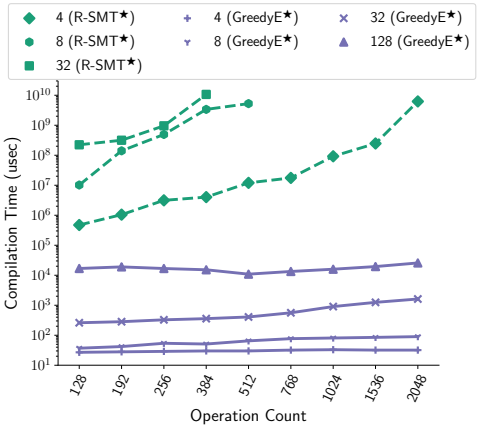


Figure 11. Scalability of optimal and heuristic methods on synthetic benchmarks. The legend shows a line's qubit count.

7.4 Success Rate and Scalability of Heuristics

We compare the success rate of heuristics to the optimal methods and evaluate the scalability of all methods.

Figure 10 compares the success rate of the heuristics and R-SMT★. Greedy methods are comparable to R-SMT★ in success rate and in some cases, they outperform R-SMT★ marginally because $\omega = 0.5$ may not the optimal value for every benchmark and machine state. GreedyE★ is as successful as R-SMT★ in all cases. Our study reveals the edge based heuristic GreedyE★, is more successful than the vertex based heuristic GreedyV★. Considering edges instead of vertices allows the heuristic to prioritize the reliability of the most frequent CNOTs.

To study the scalability of optimal and heuristic methods, we used a benchmark of randomly generated quantum programs. Figure 11 shows the compilation time on the benchmark. R-SMT★ requires up to 3 hours to compile a program with 32 qubits and 384 gates. On the other hand, the greedy methods compile programs in under one second in all cases.

8 Related Work

Quantum programming languages and their compilers have been developed by extending languages such as C and C# with quantum functionality. Examples include Quipper [21, 22], LIQUi|⟩ [69], and Scaffold [30, 58]. ProjectQ [65] is a Python framework to describe quantum circuits and compile them for different backends. PyQuil, developed at Rigetti [53, 54] is another such Python framework. Until very recently, most backends were simulators or resource-estimators, rather

than real hardware. Our work here is an early example of top-to-bottom compilation from a high-level QC language (Scaffold) to real hardware.

OpenQASM [10] and Quil [63] are low-level assembly language interfaces to QC hardware [28]. To target IBM machines, our compiler produces optimized OpenQASM code. Our compiler can be easily extended to generate code for other low-level interface languages also.

QC compilation has been studied for different hardware technologies and topologies. [23] develops a heuristic to schedule quantum circuits on linear topologies where all gates (including swaps) consume unit time. [67] uses AI planners for scheduling a specific class of quantum circuits. [24] develops heuristic techniques for ion trap systems. Recently, [62] compiled small benchmarks for IBM systems, based on only qubit topology information, not calibration data. Two recent works [71, 72] reduce swap operations and optimize 1-qubit gates for 5-qubit IBM systems. Other prior work [6, 8, 13, 15, 18, 34, 37, 38, 49, 51, 56, 57, 60, 70] are either manual methods or restricted to a specific architecture, or a specific class of quantum programs; none account for real gate durations, gate errors and variations in qubit coherence time. Similarly, other work has focused on compilation issues in future QC systems with ECC [35, 39, 44–48]. In contrast to these works, our compiler is designed and evaluated using a real IBM QC system. Using real-system measurements, we show that driving compilation decisions based on machine calibration and configuration data dramatically improves program success rates.

[66, 68] observed the usefulness of calibration data. While [68] uses error data manually to improve execution success, [66] proposes the use of calibration data-aware qubit mapping and movement policies on the 20-qubit IBM system. However, they do not perform any real hardware executions of their mapped code, making it difficult to compare results based on reliability. Their work also does not discuss how program success rates are computed on the simulator and uses error rates which are scaled by 10x. Simulated or scaled success rates may not correlate well with real performance. [16] is another recent work which maps circuits in described in the low-level OpenQASM language to *IBMQ16*. Their simulated annealing based method considers only CNOT error rates to compute the qubit mapping. In contrast, our work develops a toolflow which maps high-level programs onto *IBMQ16*, using both CNOT and readout error rates, gate times, coherence times and qubit layout. Using real-system evaluations our work determines the relative importance of these parameters and compares the performance of heuristic and optimal techniques.

9 Conclusions

This paper proposed and evaluated calibration-aware compiler techniques for NISQ systems. We considered optimal and heuristic compilation methods, the use of calibration data, different objective functions and routing policies. Our evaluations show it is crucial to adapt quantum program compilation to dynamic operation error characteristics of the machine. It is most important to consider CNOT and readout error rates, since these operations are more noisy than single qubit gates. Optimization based on qubit coherence time is also useful, but less critical here because gate errors severely limit useful computation time. Our research has shown that SMT approaches are very effective for current and near-term systems, but may not scale well to the far-NISQ machines of 500 qubits or more. For those, we have developed heuristic approaches, GreedyV* and GreedyE*, which offer nearly as good results but with much more tractable compile times.

This paper's results offer important insights on QC based on real-system measurements. Our work shows the importance of initial qubit placement. Namely, benchmarks which require more qubit movement are hard to reliably execute on systems with grid topologies. Our results show that proper placement could result in over 10X improvements in run success rate. Mapping and scheduling based on calibration data offer further benefits. Ultimately the best-performing approach offered up to 18X improvement (2.9X average) in success rate and up to 6X (2.7X average) improvement in runtime over the current IBM Qiskit baseline. Our results also give insights to future system designers. Developing richer qubit topologies will reduce the need for SWAP operations and improve the reliability of important quantum primitives such as the Toffoli gate.

Our work is relevant for future QC systems for several reasons. Fundamental unreliability in qubits [33] and short coherence times, even with Schoelkopf's coherence scaling law [59], necessitate optimizations based on error rates and gate times. Although QEC is promising in the long run, even a single logical error-corrected qubit will be composed of many noisy qubits and our methods will be useful to perform noise-adaptive compilation of error correcting circuits. Our methods can also be extended to map programs to logical qubits based on their error properties. Our techniques can be adapted for other qubit technologies such as trapped ions [12] and other routing approaches such as teleportation-based communication [9] by choosing the appropriate constraints in the optimization.

Overall, given the challenges of building reliable and scalable QC hardware, the key for the next five years or more will lie in ultra-efficient use of the resources available in NISQ systems. Our tool offers important leverage in stewarding runtime resource usage and optimizing reliability.

Acknowledgments

This work is funded in part by EPiQC, an NSF Expedition in Computing, under grants CCF-1730449/1730082, in part by NSF PHY-1818914 and a research gift from Intel.

References

[1] Ali Javadi Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Fred Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. *Scaffold: Quantum Programming Language.* Report TR-934-12. Princeton University.

[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 32, 6 (June 2013), 818–830. https://doi.org/10.1109/TCAD.2013.2244643

[3] Ethan Bernstein and Umesh Vazirani. 1993. Quantum Complexity Theory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC '93)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/167088.167097

[4] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549 (13 Sep 2017), 195 EP –. http://dx.doi.org/10.1038/nature23474

[5] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *νZ* - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–199.

[6] Amlan Chakrabarti, Susmita Sur-Kolay, and Ayan Chaudhury. 2011. Linear Nearest Neighbor Synthesis of Reversible Circuits by Graph Partitioning. arXiv:arXiv:1112.0564

[7] Andrew M. Childs and Wim van Dam. 2007. Quantum Algorithm for a Generalized Hidden Shift Problem. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1225–1232. http://dl.acm.org/citation.cfm?id=1283383.1283515

[8] Byung-Soo Choi and Rodney Van Meter. 2011. On the Effect of Quantum Interaction Distance on Quantum Addition Circuits. *J. Emerg. Technol. Comput. Syst.* 7, 3, Article 11 (Aug. 2011), 17 pages. https://doi.org/10.1145/2000502.2000504

[9] K. S. Chou, J. Z. Blumoff, C. S. Wang, P. C. Reinhold, C. J. Axline, Y. Y. Gao, L. Frunzio, M. H. Devoret, Liang Jiang, and R. J. Schoelkopf. 2018. Deterministic teleportation of a quantum gate between two logical qubits. arXiv:arXiv:1801.05283

[10] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:arXiv:1707.03429

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[12] S. Debnath, N. M. Linke, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe. 2016. Demonstration of a small programmable quantum computer with atomic qubits. *Nature* 536 (03 Aug 2016), 63 EP –. http://dx.doi.org/10.1038/nature18648

[13] Simon J. Devitt. 2016. Programming Quantum Computers Using 3-D Puzzles, Coffee Cups, and Doughnuts. *XRDS* 23, 1 (Sept. 2016), 45–50. https://doi.org/10.1145/2983545

[14] Simon J. Devitt, Ashley M. Stephens, William J. Munro, and Kae Nemoto. 2013. Requirements for fault-tolerant factoring on an atom-optics quantum computer. *Nature Communications* 4 (03 Oct 2013), 2524 EP –. http://dx.doi.org/10.1038/ncomms3524 Article.

[15] Mohammad Javad Dousti and Massoud Pedram. 2012. Minimizing the Latency of Quantum Circuits During Mapping to the Ion-trap Circuit Fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*. EDA Consortium, San Jose, CA, USA, 840–843. http://dl.acm.org/citation.cfm?id=2492708.2492917

[16] Will Finigan, Michael Cubeddu, Thomas Lively, Johannes Flick, and Prineha Narang. 2018. Qubit Allocation for Noisy Intermediate-Scale Quantum Computers. arXiv:arXiv:1810.08291

[17] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2018. A Microarchitecture for a Superconducting Quantum Processor. *IEEE Micro* 38, 3 (May 2018), 40–47. https://doi.org/10.1109/MM.2018.032271060

[18] Mrityunjay Ghosh, Amlan Chakrabarti, and Niraj K. Jha. 2017. Automated Quantum Circuit Synthesis and Cost Estimation for the Binary Welded Tree Oracle. *J. Emerg. Technol. Comput. Syst.* 13, 4, Article 51 (June 2017), 14 pages. https://doi.org/10.1145/3060582

[19] Christopher J. Glass and Lionel M. Ni. 1994. The Turn Model for Adaptive Routing. *J. ACM* 41, 5 (Sept. 1994), 874–902. https://doi.org/10.1145/185675.185682

[20] Teofilo F. Gonzalez and David Serena. 2004. Complexity of pairwise shortest path routing in the grid. *Theoretical Computer Science* 326, 1 (2004), 155 – 185. https://doi.org/10.1016/j.tcs.2004.06.027

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

[22] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. *SIGPLAN Not.* 48, 6 (June 2013), 333–342. https://doi.org/10.1145/2499370.2462177

[23] Gian Giacomo Guerreschi and Jongsoo Park. 2017. Two-step approach to scheduling quantum circuits. arXiv:arXiv:1708.00023

[24] Jeff Heckey, Shruti Patil, Ali JavadiAbhari, Adam Holmes, Daniel Kudrow, Kenneth R. Brown, Diana Franklin, Frederic T. Chong, and Margaret Martonosi. 2015. Compiler Management of Communication and Parallelism for Quantum Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 445–456. https://doi.org/10.1145/2694344.2694357

[25] Hsu, Jeremy. 2018. CES 2018: Intel's 49-Qubit Chip Shoots for Quantum Supremacy. https://spectrum.ieee.org/tech-talk/computing/hardware/intels-49qubit-chip-aims-for-quantum-supremacy. Accessed: 2018-08-05.

[26] IBM. 2018. IBM Announces Advances to IBM Quantum Systems and Ecosystem. https://www-03.ibm.com/press/us/en/pressrelease/53374.wss. Accessed: 2018-08-05.

[27] IBM. 2018. IBM Qiskit. https://qiskit.org/. Accessed: 2018-08-05.

[28] IBM. 2018. IBM Quantum Devices. https://quantumexperience.ng.bluemix.net/qx/devices. Accessed: 2018-05-16.

[29] IBM. 2018. IBM Quantum Experience. https://github.com/Qiskit/qiskit-api-py. Accessed: 2018-11-16.

[30] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2597917.2597939

[31] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549 (13 Sep 2017), 242 EP –. http://dx.doi.org/10.1038/nature23879

[32] Julian Kelly. 2018. A Preview of Bristlecone, Google's New Quantum Processor. https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html. Accessed: 2018-08-05.

[33] P. V. Klimov, J. Kelly, Z. Chen, M. Neeley, A. Megrant, B. Burkett, R. Barends, K. Arya, B. Chiaro, Yu Chen, A. Dunsworth, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, T. Huang, E. Jeffrey, Erik Lucero, J. Y. Mutus, O. Naaman, C. Neill, C. Quintana, P. Roushan, Daniel Sank, A. Vainsencher, J. Wenner, T. C. White, S. Boixo, R. Babbush, V. N. Smelyanskiy, H. Neven, and John M. Martinis. 2018. Fluctuations of

Energy-Relaxation Times in Superconducting Qubits. *Phys. Rev. Lett.* 121 (Aug 2018), 090502. Issue 9. https://doi.org/10.1103/PhysRevLett.121.090502

[34] A. Kole, K. Datta, and I. Sengupta. 2018. A New Heuristic for N-Dimensional Nearest Neighbor Realization of a Quantum Circuit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (Jan 2018), 182–192. https://doi.org/10.1109/TCAD.2017.2693284

[35] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever. 2018. Mapping of Lattice Surgery-based Quantum Circuits on Surface Code Architectures.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[37] C. Lin, A. Chakrabarti, and N. K. Jha. 2014. FTQLS: Fault-Tolerant Quantum Logic Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 6 (June 2014), 1350–1363. https://doi.org/10.1109/TVLSI.2013.2269869

[38] C. Lin, S. Sur-Kolay, and N. K. Jha. 2015. PAQCS: Physical Design-Aware Fault-Tolerant Quantum Circuit Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 7 (July 2015), 1221–1234. https://doi.org/10.1109/TVLSI.2014.2337302

[39] Y. Lin, B. Yu, M. Li, and D. Z. Pan. 2018. Layout Synthesis for Topological Quantum Circuits With 1-D and 2-D Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (Aug 2018), 1574–1587. https://doi.org/10.1109/TCAD.2017.2760511

[40] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. 2017. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3305–3310. https://doi.org/10.1073/pnas.1618020114 arXiv:http://www.pnas.org/content/114/13/3305.full.pdf

[41] N. David Mermin. 2007. *Quantum Computer Science: An Introduction*. Cambridge University Press, New York, NY, USA.

[42] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, New York, NY, USA.

[43] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 495–506. https://doi.org/10.1145/2491956.2462163

[44] Adam Paetznick and Austin G. Fowler. 2013. Quantum circuit optimization by topological compaction in the surface code. arXiv:arXiv:1304.2807

[45] Alexandru Paler, Simon J. Devitt, Kae Nemoto, and Ilia Polian. 2014. Mapping of Topological Quantum Circuits to Physical Hardware. *Scientific Reports* 4 (11 Apr 2014), 4657 EP –. http://dx.doi.org/10.1038/srep04657 Article.

[46] Alexandru Paler, Austin G. Fowler, and Robert Wille. 2017. Online Scheduled Execution of Quantum Circuits Protected by Surface Codes. arXiv:arXiv:1711.01385

[47] Alexandru Paler, Austin G. Fowler, and Robert Wille. 2017. Synthesis of Arbitrary Quantum Circuits to Topological Assembly: Systematic, Online and Compact. *Scientific Reports* 7, 1 (2017), 10414. https://doi.org/10.1038/s41598-017-10657-8

[48] Alexandru Paler, Ilia Polian, Kae Nemoto, and Simon J Devitt. 2017. Fault-tolerant, high-level quantum circuits: form, compilation and description. *Quantum Science and Technology* 2, 2 (2017), 025003. http://stacks.iop.org/2058-9565/2/i=2/a=025003

[49] M. Pedram and A. Shafaei. 2016. Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures. *IEEE Circuits and Systems Magazine* 16, 2 (Secondquarter 2016), 62–74. https://doi.org/10.1109/MCAS.2016.2549950

[50] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5 (23 Jul 2014), 4213 EP –. http://dx.doi.org/10.1038/ncomms5213 Article.

[51] Paul Pham and Krysta M. Svore. 2013. A 2D Nearest-neighbor Quantum Architecture for Factoring in Polylogarithmic Depth. *Quantum Info. Comput.* 13, 11-12 (Nov. 2013), 937–962. http://dl.acm.org/citation.cfm?id=2535639.2535642

[52] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. arXiv:arXiv:1801.00862

[53] Rigetti. 2018. PyQuil. https://github.com/rigetticomputing/pyquil. Accessed: 2018-08-01.

[54] Rigetti. 2018. Rigetti Forest. http://forest.rigetti.com. Accessed: 2018-08-01.

[55] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin Lauter. 2017. Quantum resource estimates for computing elliptic curve discrete logarithms. arXiv:arXiv:1706.06752

[56] D. Ruffinelli and B. Baran. 2016. A multiobjective approach to linear nearest neighbor optimization for 2D quantum circuits. In *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, Valparaiso, Chile, 1–8. https://doi.org/10.1109/CLEI.2016.7833378

[57] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. 2011. Synthesis of Quantum Circuits for Linear Nearest Neighbor Architectures. *Quantum Information Processing* 10, 3 (June 2011), 355–377. https://doi.org/10.1007/s11128-010-0201-2

[58] ScaffCC Compiler. 2018. Compiler for the Scaffold Language. https://github.com/epiqc/ScaffCC. Accessed: 2018-05-16.

[59] Adam Sears. 2013. *Extending Coherence in Superconducting Qubits: from Microseconds to Milliseconds*. PhD dissertation. Yale University.

[60] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. 2013. Optimization of Quantum Circuits for Interaction Distance in Linear Nearest Neighbor Architectures. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 41, 6 pages. https://doi.org/10.1145/2463209.2488785

[61] P. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (1999), 303–332. https://doi.org/10.1137/S0036144598347011 arXiv:https://doi.org/10.1137/S0036144598347011

[62] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintao Pereira. 2018. Qubit Allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 113–125. https://doi.org/10.1145/3168822

[63] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A Practical Quantum Instruction Set Architecture. arXiv:arXiv:1608.03355

[64] Mathias Soeken, Thomas Haner, and Martin Roetteler. 2018. Programming Quantum Computers Using Design Automation. arXiv:arXiv:1803.01022

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

[66] Swamit S. Tannu and Moinuddin K. Qureshi. 2018. A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. arXiv:arXiv:1805.10224

[67] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. 2018. Compiling quantum circuits to realistic hardware architectures using temporal planners. *Quantum Science and Technology* 3, 2 (2018), 025004. http://stacks.iop.org/2058-9565/3/i=2/a=025004

[68] Christophe Vuillot. 2017. Is error detection helpful on IBM 5Q chips ? arXiv:arXiv:1705.08957

[69] Dave Wecker and Krysta M. Svore. 2014. LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing. arXiv:arXiv:1402.4467

[70] Mark Whitney, Nemanja Isailovic, Yatish Patel, and John Kubiatowicz. 2007. Automated Generation of Layout and Control for Quantum Circuits. In *Proceedings of the 4th International Conference on Computing Frontiers (CF '07)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/1242531.1242546

[71] Xin Zhang, Hong Xiang, Tao Xiang, Li Fu, and Jun Sang. 2018. An efficient quantum circuits optimizing scheme compared with QISKit. arXiv:arXiv:1807.01703

[72] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2017. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. arXiv:arXiv:1712.04722