Automating NISQ Application Design with Meta Quantum Circuits with Constraints (MQCC)

HAOWEI DENG, University of Maryland, College Park, USA
YUXIANG PENG, University of Maryland, College Park, USA
MICHAEL HICKS, University of Maryland, College Park, USA
XIAODI WU, University of Maryland, College Park, USA

Near-term intermediate scale quantum (NISQ) computers are likely to have very restricted hardware resources, where precisely controllable qubits are expensive, error-prone, and scarce. Programmers of such computers must therefore balance trade-offs among a large number of (potentially heterogeneous) factors specific to the targeted application and quantum hardware. To assist them, we propose Meta Quantum Circuits with Constraints (MQCC), a meta-programming framework for quantum programs. Programmers express their application as a succinct collection of normal quantum circuits stitched together by a set of (manually or automatically) added meta-level choice variables, whose values are constrained according to a programmable set of quantitative optimization criteria. MQCC's compiler generates the appropriate constraints and solves them via an SMT solver, producing an optimized, runnable program. We showcase a few MQCC's applications for its generality including an automatic generation of efficient error syndrome extraction schemes for fault-tolerant quantum error correction with heterogeneous qubits and an approach to writing approximate quantum Fourier transformation and quantum phase estimation that smoothly trades off accuracy and resource use. We also illustrate that MQCC can easily encode prior one-off NISQ application designs —multi-programming (MP), crosstalk mitigation (CM)— as well as a combination of their optimization goals (i.e., a combined MP-CM).

1 INTRODUCTION

Quantum computers offer potentially significant performance advantages over classical ones for important classes of problems [Grover 1996; Shor 1997]. Hardware advances have been bringing this potential ever closer to reality, but we are not there yet. Near-term, intermediate-scale quantum computing (NISQ) devices have few quantum bits (qubits), and these are prone to errors from several sources. General-purpose error correction techniques [Calderbank and Shor 1996; Gottesman 1997; Shor 1995; Steane 1996] consume a substantial number of qubits, so they are not a practical remedy.

As a result, the design of NISQ applications needs to explore ways to optimize the efficiency and/or reliability of programs by balancing competing tradeoffs. Consider the following few examples.

Syndrome extraction for fault-tolerant quantum error correction [Gottesman 2010; Steane 2006] is an example where one wants to minimize the fidelity loss in extracting quantum error syndrome information by measurements. Sequentially extracting syndromes, one by one, would incur many measurements, which each introduces noise. Chao and Reichardt [2018] propose a scheme to extract multiple syndromes at once, *reducing* the total number of measurements; however, this scheme requires *more* (noisy) qubits. Chao and Reichardt present an analytical tradeoff strategy for when all qubits have the same quality. However, such theoretical analysis becomes impossible when qubits are heterogeneous, which is common on NISQ machines.

Another example is the generation of **approximate circuits of important quantum subroutines**, like Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE) [Barenco et al. 1996]. The goal is to save *resources* by reducing *accuracy*. The approach is to identify and

Authors' addresses: Haowei Deng, Department of Computer Science and Joint Center for Quantum Information and Computer Science, University of Maryland, College Park, USA; Yuxiang Peng, Department of Computer Science and Joint Center for Quantum Information and Computer Science, University of Maryland, College Park, USA; Michael Hicks, Department of Computer Science, University of Maryland, College Park, USA; Xiaodi Wu, Department of Computer Science and Joint Center for Quantum Information and Computer Science, University of Maryland, College Park, USA, xwu@cs.umd.edu.

prune gates in the fully accurate QFT and QPE circuits, where the pruned gates won't significantly change the generated unitary under some distance measure. A specific gate pruning strategy was suggested by Barenco et al. which is however non-optimal for specific input sizes and gate selection.

A final category of example is **scheduling programs to best leverage target architectural constraints**. By doing *multi-programming* (MP), we can *increase* overall computer utilization by running multiple programs at once (in "parallel"). But doing so may *decrease* reliability: particular gates and qubits may be more error-prone than others, so multi-programming tightens scheduling options. Das et al. [2019] propose an algorithm to balance the tradeoff. *Crosstalk mitigation* (CM) is another such example, where nearby gate/qubit pairs scheduled in parallel may experience noise due to crosstalk. Placing them in sequence *decreases* crosstalk noise, but *increases* the chances of error due to decoherence. Murali et al. [2020] propose a layout algorithm to balance the tradeoff.

All of these works offer mechanisms to relax a program's output fidelity so as to optimize some other attribute of its performance. In this sense, they support *approximate computing* [Carbin et al. 2013; Hung et al. 2019; Misailovic et al. 2014], which aims to make the best use of imperfect hardware. Unfortunately, each only offers one-off improvements. The works either lack algorithmic/automated support entirely, or when they have it, this support cannot be composed or combined easily, due to conflicting tradeoffs that themselves would need balancing. For example, removing "parallelism" to mitigate crosstalk reduces the utilization MP hopes to gain, but also adds a new source of error MP should consider.

Contribution

In this paper, we present **Meta Quantum Circuits** with **Constraints** (MQCC), the first general-purpose approximate computing framework for quantum programs. MQCC is a framework that makes it easy to design, implement, and experiment with optimizations, and to support *programming* their customized composition while leveraging automated reasoning. Crucially, MQCC allows users to express and optimize a *variety* of metrics. Because we are in a stage of rapid development for quantum hardware, application designers need to balance trade-offs among many emerging or even unknown factors.

To use MQCC, an optimization designer starts by writing (or reusing) routines to compute various **attributes** of quantum programs; these are the basis of optimization. We have implemented seven attributes so far: *qubitcount*, *gatecount*, *crosstalk* [Murali et al. 2019a], *fidelity*, *accuracy*, *circuit depth*, and *quantum circuit space-time volume* [Fowler et al. 2012; Holmes et al. 2019]. Next, the designer writes a *transpiler* that takes the optimization's input program(s) and introduces meta-level **choice variables** into them.[1] In essence, a program with choice variables is a *family* of programs, and a valuation of those variables identifies one member of the family. Finally, the designer states an optimization goal in terms of the attributes of interest, e.g., to maximize one attribute while keeping another below a threshold.

Now, given a transpiled input, MQCC selects the values of the choice variables that satisfy the goal. It analyzes the program with respect to the attributes of interest and generates symbolic *cost expressions* over the choice variables which express the program's attributes' values with respect to the goal. MQCC encodes these as Satisfiability Modulo Theories (SMT) formulae and solves for the choice variables, thereby selecting a final program to run on an actual platform. In the worst case, formula sizes are exponential in the number of choice variables due to the essential hardness associated with the worst-case optimization problem. But for NISQ-era programs, formula sizes are often not large so running times are often reasonable. The scalability study in Section 5.6 shows that most middle-size QASMbench programs, which have up to 10^4 gates, can still be handled by

[1]Transpilation is not strictly needed—users of an optimization could insert choice variables manually.

MQCC efficiently. We have also identified a special case of *additive* attributes whose formulae are linear in the number of choice variables, and thus scale better.

We demonstrate MQCC's generality by using it to implement several case studies. We employ MQCC to automate the selection of a syndrome extraction scheme of Chao and Reichardt [2018], where we easily handle the heterogeneous qubit case in MQCC. We also implemented an automated procedure to trade accuracy for savings of circuit volume in implementations of Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE). Lastly, we implemented both the MP and CM optimizations listed above (each involves a tradeoff of two attributes) and also implemented a novel composition of MP and CM which balances the tradeoff among the *three* attributes from MP and CM combined—both use *depth*, but individually they use *noise* and *crosstalk* attributes. MQCC makes this composition simple to express.

We compared our syndrome extraction strategy with Chao and Reichardt [2018]'s original extraction scheme as well as an alternative, more parallel scheme on quantum systems that have of qubits with heterogeneous random errors, developed by Reichardt [2020]. Our experiments suggest that the MQCC-based solution can always achieve the minimum logical error rate for all kinds of random errors. In the experiment, MQCC generates the solution in less than 0.1 seconds.

For gate-pruning QFT and QPE, MQCC's automation is able to identify more efficient strategies than existing ones [Barenco et al. 1996] for the entire parameter range. MQCC can generate the solution for QFT/QPE instances with 50 qubits and 11k space-time volume [Fowler et al. 2012] in 40 seconds.

For the remaining case studies, we applied the optimizations to a benchmark of quantum programs and demonstrated the benefits by running on actual NISQ machines. For CM and MP, we match or improve previously reported results. Our new optimization, which combines both MP and CM attributes, allows one to take the crosstalk-induced noise into the consideration in MP tasks. As a result, we generate multi-programming schedules with both high success probability and small circuit depth, compared with the one generated with MP attributes alone, on actual NISQ machines. In all these cases, MQCC's solver performs well, taking less than 0.1 seconds for each program.

Because NISQ-ready programs are small, we also ran a separate experiment on a benchmark of larger programs (too big to run on today's hardware) to see how well MQCC scales. In particular, we test MQCC's performance of MP, CM, and MP-CM tasks on a collection of middle-size circuits (10∼20 qubits with 100∼1000 gates) from representative quantum applications in QASMBench [Li et al. 2021]. MQCC is able to generate the solution for most test cases within a few seconds, with some exceptions in a few minutes.

As a final remark, MQCC can be easily integrated into the existing ecosystem of quantum computing tool-chains. MQCC builds on top of OpenQASM [Cross et al. 2017] and can produce executable programs in Qiskit and AWS Braket. All code is freely available.

Related Work. Fast but error-prone chips in classical computation inspired the development of frameworks to trade correctness for performance. Carbin et al. [2013] proposed Rely to handle reliability specifications and analysis. Users of Rely can specify the quantitative reliability of each component, and the compiler automatically reasons whether the program is reliable enough. Misailovic et al. [2014] made one step further with Chisel, automatically optimizing the tradeoff between reliability and accuracy via integer linear programs. MQCC is inspired by Chisel's approach: both set up a constraint problem whose solution selects instructions based on an optimized-for objective. However, MQCC is more general: With MQCC, users can easily define their own attributes and objective to optimize, whereas with Chisel both the attributes and objective are fixed. Moreover, in actual manifestation, we have used MQCC on many more, and various, applications than Chisel did in the classical realm.

Hung et al. [2019] and Tao et al. [2021] define logics of *quantum robustness* to assess the potential noise accumulation in quantum programs. These logics permit reasoning about noise, but provide no means to automatically compensate for it. The optimized quantities in these works [Hung et al. 2019; Tao et al. 2021] —reliability, noise, resources, etc.—are *additive attributes*, in our terminology. For our applications, we also crucially rely on the flexibility of general attributes provided by MQCC.

SMT solvers are widely used in programming language and architecture designs, e.g., as the basis for automation in program verification [Filliâtre and Paskevich 2013; Srivastava et al. 2009], and specification-based program synthesis [Gulwani et al. 2011; Srivastava et al. 2011, 2010]. The solver-aided host language Rosette [Torlak and Bodik 2013, 2014] has been designed to ease the construction of solver-aided domain-specific languages. SMT solvers have also been employed to design NISQ applications. In addition to the crosstalk example [Murali et al. 2020], one can also model the qubit mapping and gate scheduling problems as SMT instances [Murali et al. 2019a,b]. MQCC provides a flexible framework that leverages SMT solvers to automate NISQ designs.

2 PRELIMINARIES: QUANTUM PROGRAMMING

Principles of Quantum Computation. The state of a quantum system is made up of *qubits*. A qubit has two *basis* states, typically written as $|0\rangle$ and $|1\rangle$. Unlike classical bits, qubits may be in a *superposition* of these states, rather than just one or the other. This is written $\alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ are complex numbers called *amplitudes* satisfying $|\alpha|^2 + |\beta|^2 = 1$. The state with $\alpha = \beta = \frac{1}{\sqrt{2}}$ is written $|+\rangle$ and the state with $\alpha = \frac{1}{\sqrt{2}}$ and $\beta = -\frac{1}{\sqrt{2}}$ is written $|-\rangle$. Information is extracted from a quantum state via *measurement*. Measuring a single qubit returns 0 or 1 with probability of $|\alpha|^2$ and $|\beta|^2$, respectively.[2] Moreover, it collapses that qubit's state to $|0\rangle$ or $|1\rangle$, i.e., setting $\alpha = 0, \beta = 1$ or vice versa. A system with n qubits can, in general, exist in a superposition of 2^n possible states; e.g., a 2-qubit state will have basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. The exponential size of this superposition, and the ability of a quantum program to process it "in parallel," is a key reason for the potential quantum advantages over classical computation.

Algorithms in a quantum system are expressed as *circuits* of quantum *gates* which process qubits, represented as wires. Such processing *evolves* the qubits' amplitudes. For example, the single-qubit *not* gate, written `x`, swaps the amplitudes of the given qubit; e.g., it would evolve $\alpha\,|0\rangle + \beta\,|1\rangle$ to $\beta\,|0\rangle + \alpha\,|1\rangle$. Another gate is the *Hadamard* gate `h`, which evolves a qubit to $\frac{\alpha+\beta}{\sqrt{2}}\,|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}\,|1\rangle$. A common two-qubit gate is *controlled not*, or CNOT, which leaves the first qubit alone but transforms the second via X if the first qubit is 1. Measurement is also an operation, like a gate, with the key differences that measurement returns a classical result and collapses the state.

Quantum Assembly Language (QASM). A quantum circuit can be specified using the "quantum assembly language" (QASM) [Dousti et al. 2015; Svore et al. 2006]. QASM is a simple text language that describes quantum circuits as a sequence of gate operations on numbered qubits. OpenQASM [Cross et al. 2017] provides a bit more high-level structure, while still being compatible with modern hardware [IBM 2021]. Fig 1(a) is an example.

The only storage types of OpenQASM (version 2.0) are classical and quantum registers, which are one-dimensional arrays of bits and qubits, respectively. The statement `qreg name[size];` declares an array of qubits while the statement `creg name[size];` declares a `size`-bit classical register. The qubits are initialized as $|0\rangle$ and the classical bits are initialized to 0.

OpenQASM supports a built-in set of arbitrary single-qubit gates with CNOT (written `cx`) as the sole two-qubit gate. A programmer can define different gates using a subroutine-like mechanism

[2]This measurement is in the Z (*computational*) basis. Measurements can be in other bases as well, as discussed in Sec 3.2.

```
qreg q[2];  creg c[2];
gate cz a,b { h b; cx a,b; h b; }
x q[0];      cz q[0],q[1];
measure q[1] -> c[1];
if (c==2) x q[0];
```

(a)

```
1 CX r[0],r[1];
2 h q[0];
3 barrier r,q[0];
4 h r[1];
```

(b)

Fig. 1. OpenQASM Examples.

with keyword `gate`; the example code defines a new `cz` gate which consists of two Hadamard gates and one CNOT gate. The `measure` statement measures a qubit and stores the result in a classical bit. The `if` statement conditionally executes a quantum operation based on the value of a classical register. This register is interpreted as an integer, using the bit at index zero as the low order bit.

OpenQASM allows gate sequential control through a special instruction `barrier`, which prevents reordering gates across its source line. Consider the example in Fig 1(b). The instruction `h r[1]` has to wait until all gates on `r[0]`,`r[1]`,`q[0]` before line 3 are finished. In particular, it cannot be executed with `h q[0]` in parallel.

3 META QUANTUM CIRCUITS WITH CONSTRAINTS

This section describes MQCC, using the problem of fault-tolerant quantum error correction (FQEC) as an example [Das et al. 2019].

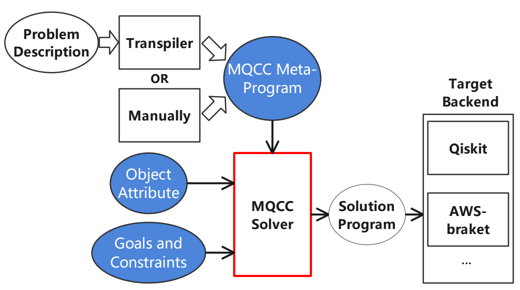
3.1 MQCC Overview

MQCC's architecture is shown in Fig 2(a). The core of MQCC is the MQCC solver, which takes three inputs: a quantum *meta-program*; the definitions of relevant *object attributes*; and an optimization *goal*.

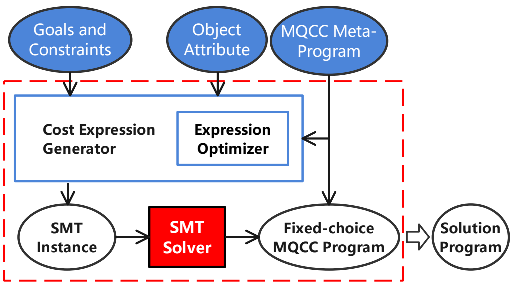
```
1 qreg q[1];
2 fcho c1 = {0, 1};
3 choice (c1) {
4     0: x(q[0]);
5     1: h(q[0]);
6 }
```

MQCC meta-programs. The syntax of the MQCC meta-program is essentially standard OpenQASM, but is extended to include *choice variables*. The valuation of the choice variables determines the actual program that will run on the quantum computer—different choice-variable valuations will yield different programs. Consider the example to the left. After declaring a quantum register, the program defines a *free choice variable* `c1` whose value can be either 0 or 1. `c1` is used in the subsequent `choice` statement. When the MQCC solver produces a solution to the choice variables, it replaces each `choice` statement with the branch corresponding the solution. So, if `c1` solved to 0, lines 3-6 would be replaced by `x(q[0]);` if it solved to 1, they would be replaced by `h(q[0]);`. After replacement, the program is normal OpenQASM and can run on quantum hardware.

The user of an optimization need not write the meta-program directly; as shown in Fig 2(a), an optimization-specific transpiler can produce it from a higher-level problem description. The insertion of choice variables for all applications in our paper is handled automatically by transpilers, whose design, however, requires domain knowledge of the corresponding application. A new transpiler is likely required for each new application, and its complexity will depend on the application. MQCC's transpilers often leverage Qiskit [Cross 2018] code for generating meta-programs.



(a) Overview of MQCC



(b) MQCC Solver

Fig. 2. Overview of MQCC

Object attributes. The second MQCC solver input is a set of relevant *object attributes*. An attribute is essentially a function from a quantum circuit to a numeric value (e.g., the count of qubits used by the circuit, the count of gates in the circuit).

Goal and constraints. The third MQCC solver input is the optimization goal and constraints that must be satisfied. The MQCC solver applies the provided attributes to the meta-program and thus generates a formula that expresses the attribute in terms of the program's choice variables. The solver then comes up with a solution for the choice variables such that these formulae satisfy the given constraints while meeting the stated goal (e.g., minimize the count of gates in the circuit while keeping the count of qubits used by the circuit under the given threshold).

3.2 Example: Fault-tolerant Quantum Error Correction

As an example of MQCC usage, we present how it can maximize fidelity when using fault-tolerant quantum error correction (FQEC) schemes, whose efficacy can depend on the target program and architectural constraints.

3.2.1 Background. Fault-tolerant quantum error correction protects quantum information from noise. Classical error correction based on error-correcting codes employs redundancy and extracts a *syndrome* to diagnose the error that corrupts an encoded state. Quantum error correction also employs syndrome extraction [Gottesman 2010]. Each syndrome is extracted by applying a specific

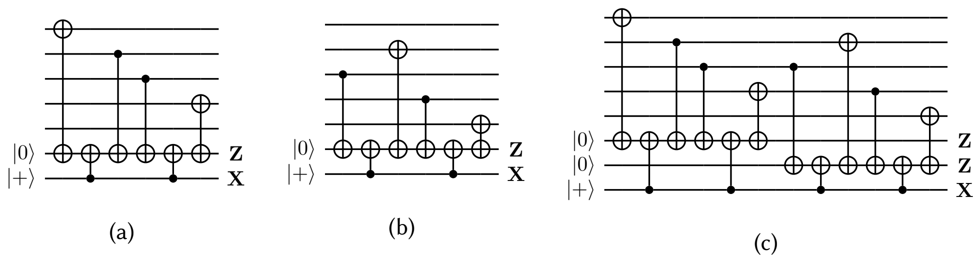
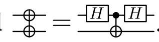


Fig. 3. Syndrome extraction circuit for the $[\![5,1,3]\!]$ code [Chao and Reichardt 2018; Reichardt 2020]. (a) Circuit to extract the $XZZXI$ syndrome. (b) Circuit to extract the $IXZZX$ syndrome. (c) Extract syndrome $XZZXI$ and $IXZZX$ in parallel.

circuit to the data qubits to extract their information into the ancilla qubits. Then these ancilla qubits are measured to retrieve the syndrome information and can be reused for the next syndrome extraction. Fig 3(a)(b) shows two examples where **Z** indicates a $|0\rangle, |1\rangle$ measurement, **X** indicates $|+\rangle, |-\rangle$ measurement and . In each circuit, the top five qubits are data qubits and others are ancilla qubits. The ancilla qubits measured with **Z** measurement are called syndrome qubits and their measurement results are used to decide whether the data qubits are corrupted. The data qubits are not corrupted only when all syndrome qubits are measured as $|0\rangle$. Otherwise, correction circuits will be applied to the data qubits based on the measurement results. The ancilla qubits measured with **X** measurement are called "flag" qubits and their measurement results are used to detect the error in the syndrome extraction circuits. The syndrome extraction circuits are correct only if all flag qubits are measured as $|+\rangle$.

As with classical error correction, many efficient FQEC codes are known [Bacon 2006; Fowler et al. 2012; Steane 1996]. One is the perfect $[\![5, 1, 3]\!]$ code [Laflamme et al. 1996]. For this code, there are four syndromes, named $XZZXI$, $IXZZX$, $XIXZZ$, and $ZXIXZ$. Fig 3(a)(b) respectively show the circuits that extract syndromes $XZZXI$ and $IXZZX$.

There are also many existing syndrome extraction strategies. For example, Shor-style syndrome extraction [DiVincenzo and Aliferis 2007] requires $w + 1$ or w ancilla qubits, where w is the largest weight of a stabilizer generator. Steane [1997, 2002] uses at least a full code block of extra qubits, while Knill [2005b] uses an encoded EPR state and thus at least two ancilla code blocks. Chao and Reichardt [2018] and Yoder and Kim [2017] propose syndrome extraction strategies based on flag qubits that use only two ancilla qubits.

For a large code with many syndromes, it can be inefficient to extract the syndromes one after the other. Reichardt [2020] introduce the method to extract multiple syndromes in parallel. The circuit in Fig 3(c) extracts $[\![5, 1, 3]\!]$ code's $XZZXI$ and $IXZZX$ syndromes at once. Compared to extracting two syndromes sequentially, extracting syndromes in parallel requires one more qubit (eight vs. seven) but one fewer qubit measurement (**ZZX** vs. **ZX**, **ZX**). There is a trade-off between the count of ancillary qubits and the number of measurements, so the strategy for extracting syndrome should be chosen carefully. Quantum error correction aims to protect quantum information from quantum noise and the trade-off should aim to increase the procedure's fidelity. Since error may occur during the qubit measurement, fewer qubit measurements can improve fidelity. On the other hand, using more ancilla qubits can harm it. In practice, different physical qubits have different error rates and the application will use the qubits with the highest fidelity first; more ancillary qubits mean that qubits with higher error rates might be used, decreasing the fidelity of the whole procedure.

3.2.2 Expressing the FQEC tradeoff with MQCC. We can use MQCC to maximize **Fidelity**—an attribute modeled by the probability that no error occurs in the error correction procedure—while satisfying the constraint **QubitCount** $< \theta$ where **QubitCount** is an attribute that estimates the count of ancilla qubits, and θ is a provided threshold. We give the definition of **Fidelity** and **QubitCount** below.

We model the probability of error using the standard depolarizing noise model [Knill 2005a; Nielsen and Chuang 2002]. The depolarizing noise model assumes that a quantum operation's error consists of random, independent applications of products of Pauli operators after the gate with probabilities determined by the gate. Suppose the "depolarizing" error for a qubit e_p: the probability that $|0\rangle$ ($|+\rangle$) state preparation erroneously produces $|1\rangle$ ($|-\rangle$). A binary (such as **Z** or **X**) measurement results in the wrong outcome with probability $e_m = e_p$; for a quantum gate, each qubit the gate is applied is modified by one of the three possible Pauli operators, each with probability e_p. With this noise model, given the e_p for each qubit, the probability P that no error occurs in a quantum circuit S can be estimated as the product of the probability of no error for

```
module dualCZ(q1, q2){
    h(q1);
    cnot(q1,q2);
    h(q1);
}

module Extract_XZZXI(data,anc,r){
    \\Extract sydrome XZZXI
    \\The same circuit as in Fig.1(a)
    reset(anc[0],anc[1]);
    h(anc[1]);
    dualCZ(data[0], anc[0]);
    cnot(anc[1], anc[0]);
    cnot(data[2], anc[0]);
    cnot(data[3], anc[0]);
    cnot(anc[1], anc[0]);
    dualCZ(data[3], anc[0]);
    measureZ(anc[0],r[0]);
    measureX(anc[1], r[1]);
}
module Extract_IXZZX(data,anc,r){...}
    \\The same circuit as in Fig.1(b)
\\Similar circuits to extract other syndromes
module Extract_XIXZZ(data,anc,r){...}
module Extract_ZXIXZ(data,anc,r){...}

module Extract_both_12(data,anc,r){
    \\Extract XZZXI and IXZZX in parallel
    \\The same circuit as in Fig.1(c)
    ...
}
module Extract_both_34(data,anc,r){
    \\Extract XIXZZ and ZXIXZ in parallel
    ...
}
```

(a)

```
qreg data[5];
qreg anc1[2];
creg r1[2], r2[2], r3[2], r4[2];
... \\Module declaration from part (a)
Extract_XZZXI(data, anc1, r1);
Extract_IXZZX(data, anc1, r2);
Extract_XIXZZ(data, anc1, r3);
Extract_IXZZX(data, anc1, r4);
```

(b)

```
\\Register and variable declarations
qreg data[5];
qreg anc1[2], anc2[3];
creg r1[2], r2[2], r3[2], r4[2];
creg r5[3], r6[3];
fcho c1,c2 = {0, 1};
... \\Module declaration from part (a)
\\Main part of the meta-program
\\Extract syndrome XZZXI and IXZZX
choice (c1){
    0:
    Extract_XZZXI(data, anc1, r1);
    Extract_IXZZX(data, anc1, r2);
    1:
    Extract_both_12(data, anc2, r5);
};

\\Extract syndrome XIXZZ and ZXIXZ
choice (c2){
    0:
    Extract_XIXZZ(data, anc1, r3);
    Extract_IXZZX(data, anc1, r4);
    1:
    Extract_both_34(data, anc2, r6);
};
```

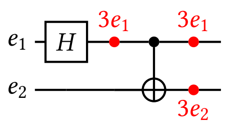
(c)

Fig. 4. (a) Predefined modules of the $[\![5,1,3]\!]$ code syndrome extraction circuits. (b) Fault-tolerant syndrome extraction for $[\![5,1,3]\!]$ code given by programmer. (c) MQCC meta-program produced by transpiler running on (b).

each operation:

$$P = \prod_{op \in S} (1 - E_{op}) \Longleftrightarrow \log(P) = \sum_{op \in S} \log(1 - E_{op}), \tag{3.1}$$

where E_{op} denotes the probability that the operation op is modified by Pauli operators. The corresponding logarithm term can change the probability expression into a linear expression. For example, consider the Bell state preparation circuit to the right. Given e_1, e_2 as the error rate for two qubits, depolarizing error may occur at the red points with



the corresponding probability. So the probability P that no error occurs in the circuit is the product $(1-3e_1)^2(1-3e_2)$.

Now we can define our two attributes, **Fidelity** and **QubitCount**. The **Fidelity** attribute applied to a syndrome extraction circuit S estimates $log(P)$ where P is the probability that no error occurs in S based on the standard depolarizing noise model [Knill 2005a] described in Section 3.2.1. The **QubitCount** attribute applied to S computes the total qubit count used by the circuit. Programmers can easily define their own attributes, and these can be reused for different programs and problems. Formal definitions of **Fidelity** and **QubitCount** are given in Section 4.

Suppose we want to optimize the program in Fig 4(b), which extracts all syndromes from the five-qubit array `data` in which the information has been pre-encoded. The program invokes syndrome extraction circuits `Extract_...`, defined in Fig 4(a) (and corresponding to those in Fig 3), one after the other. We want to allow for the possibility that some could be extracted in parallel depending on our optimization tradeoff. In MQCC, this possibility is expressed with choice variables. Fig 4(c) shows the meta program that corresponds to Fig 4(b). It differs in that it has introduced two choice variables whose solution may allow calling `Extract_both_12` and/or `Extract_both_34`, respectively, which invoke two extraction circuits in parallel, rather than the alternative invocation in sequence. We produce this meta-program automatically by running a transpiler on Fig 4(b). Transpilation is often easy to build, which will be discussed in Section 3.4.

Let's look more closely at the meta-program in Fig 4(c).

Line 2-5 define quantum and classical registers used in the program as usual, using the **qreg** and **creg** syntax.

Line 6 declares the program's choice variables. We use keyword **fcho** to define two *free choice variables* c_1, c_2 that choose value in {0,1}. A choice variable's value can be any integer within an enumeration $\{a_1, a_2, ..., a_n\}$ or an interval $[a_1, a_2]$ with $a_1 < a_2$. In this program, these two choice variables are used to decide the strategy of extracting four syndromes.

Line 7 is a placeholder for the syndrome extraction circuits given in Fig 4(a). A module in MQCC can be viewed as macro over its parameters.

Lines 10-25 contains the part of the meta-program that expresses the possible schedules. Two **choice** statements decide to extract syndromes sequentially or in parallel, based on the value of choice variables c_1, c_2. The **choice** statement on lines 10-16 says that extracting syndrome $XZZXI$ and $IXZZX$ sequentially with only two ancilla qubits if $c1 = 0$ or in parallel with three ancilla qubits if $c1 = 1$. The **choice** statement on lines 19-25 does similarly for syndrome $XIXZZ$ and $ZXIXZ$.

3.3 MQCC Solver

Now let us see how the MQCC solver works. Its operation is shown in Fig 2(b).

Cost Expression Generator. The MQCC solver's *Cost Expression Generator* (CEG) computes each input attribute for the meta-program to produce a formula that expresses that attribute's value in terms of the meta-program's choice variables. For the example in Fig 4, the CEG would produce the following formula for the **QubitCount** attribute:

$$\textbf{QubitCount}: \quad 7\delta^0_{c_1}\delta^0_{c_2} + 8\delta^0_{c_1}\delta^1_{c_2} + 8\delta^1_{c_1}\delta^0_{c_2} + 8\delta^1_{c_1}\delta^1_{c_2}. \tag{3.2}$$

Here, the term δ^i_c evaluates to 1 if the value of c equals i, and evaluates to 0 otherwise. Referring to Fig 4, we can see that if $c_1 = c_2 = 0$ then all syndromes are extracted sequentially and only need seven qubits in total (five data qubits + two ancilla qubits); otherwise, syndromes are extracted in parallel in at least one choice statement, so eight qubits are needed in total.

In the general case, the number of terms in a CEG-produced formula relates to the number of valuations of the choice variables. We see this in the formula for **QubitCount**, which has four terms. However, we identified an optimized cost generation algorithm for attributes we call *additive*, which means that the attribute value of a program S can be computed from a linear combination of its sub-programs. The resulting expression will be linear in the number of choice variables. As an example, the **Fidelity** attribute is additive. The probability that no error occurs after two operations is the product of the individual operations' probabilities; the **Fidelity** attribute evaluates the logarithm of the probability, translating the product to a sum. A programmer may specify when an attribute is additive, which will prompt the CEG to optimize the generation of its cost expression (indicated as *Expression Optimizer* in the figure).

Consider our example in Fig 4. Suppose the **Fidelity** when extracting syndromes $XZZXI$ and $IXZZX$ sequentially or in parallel is -0.011 and -0.012, respectively. Then the fidelity of the first **choice** statement (lines 10-16) is calculated as $-0.011\delta^0_{c_1} + (-0.012)\delta^1_{c_1}$. Similarly suppose the error of the second **choice** statement (lines 19-25) is calculated as $-0.013\delta^0_{c_2} + (-0.012)\delta^1_{c_2}$. Since the **Fidelity** attribute is additive, the CEG sums these two:

$$\textbf{Fidelity}: \quad -0.011\delta^0_{c_1} + (-0.012)\delta^1_{c_1} + (-0.013)\delta^0_{c_2} + (-0.012)\delta^1_{c_2}. \tag{3.3}$$

How this formula was computed is explained in Section 4.3.

Solution by SMT Encoding. With the cost expressions generated for each object, MQCC encodes them as SMT instances based on the user's goal and constraints. Then MQCC uses an SMT solver to assign values to choice variables. In FQEC case, if the **QubitCount** threshold is 7, MQCC will choose $c_1 = 0, c_2 = 0$, and extract all syndromes sequentially. If there is more allowed qubits, MQCC will choose $c_1 = 0, c_2 = 1$ to maximize the Equation 3.3.

Scalability. This example application demonstrates that some attributes are exponential in the choice space, but the choice space tends to be rather small in many applications. Equation 3.2 shows the formula of attribute QubitCount as an example. QubitCount is not an additive attribute. When composing several subprograms whose QubitCount is determined by a choice variable, evaluating the total QubitCount of the composed program has exponential complexity since each subprogram may share some qubits. But in FQEC applications, the number of choice variables depends on the count of syndromes to extract. The syndrome extraction program for the most commonly used quantum error correction code needs no more than ten syndromes (e.g., five in the perfect code, six in the Steane code, and eight in the Shor code). So MQCC has good scalability in this application.

3.4 Construction of MQCC Meta-programs

Ideally, an optimization designer will write a transpiler to automatically construct an appropriate MQCC meta-program given a normal, target program. This allows normal programmers to use MQCC in a push-button fashion: They specify the tradeoff they want optimize, provide their input program to that optimization's custom transpiler, and then invoke MQCC on the result, which produces the optimized program. Without a transpiler, e.g., perhaps when experimenting with a new optimization of different tradeoffs, a programmer can manually construct a meta program.

In our experience, writing transpilers is straightforward: We have done so for every optimization presented in this paper. For the FQEC example just presented, the transpiler works by first detecting the substitutable usage of FQEC modules, e.g., `Extract_IXZZX` immediately followed by `Extract_XIXZZ` on the same registers. When it finds one, it generates a corresponding `choice` variable (as on line 6 in Fig 4(c)) and introduces a `choice` statement which selects between the original usage and one that happens in parallel (as on lines 10-25 in Fig 4(c)). Building a transpiler requires modeling the design space of the target problem, which will then determine the use of choice variables: their locations and granularity, and pieces of alternative QASM programs that will be stitched together. In our experience, the modeling step is natural given the problem, and the engineering overhead of building a transpiler is minimal; it took us 1 or 2 hours on average, for each optimization.

3.5 Implementation of MQCC

We implement MQCC in Python using PLY [Beazley 2018], which provides lex and yacc parsing tools. We choose Python for its popularity, accessibility, and flexibility. In particular, it allows the developers to easily define various attributes with Python classes. The SMT optimization for MQCC uses the Z3 SMT solver [De Moura and Bjørner 2008] version 4.8.9.

4 FORMALIZATION OF MQCC

This section presents MQCC formally, including its meta-language, attribute definitions, and symbolic cost expressions (but not its app-specific transpilers). We prove that additive attributes' optimized cost procedure is correct.

4.1 Language Syntax

The formal syntax of MQCC meta-programs is shown in Figure 5. A meta-program P consists of a sequence of declarations D and a statement S. There are two kinds of declarations:[3] *RegDecl* declares classical and quantum registers, and *VarDecl* declares MQCC choice variables. A statement S can be empty, an operation O, a *case*, a *choice*, or a sequence of semicolon-separated statements.

- Operations O are primitive operations op over a list of registers $\overrightarrow{reg}$, according to a list of (optional) parameters $\overrightarrow{r}$. An operation could be a quantum gate, in which case op is the name of the gate and $\overrightarrow{reg}$ identifies input/output quantum registers, e.g., `cnot(q1,q2)`. An operation could also be a measurement, e.g., `measure(q1,c1)`, which measures `q1`'s contents and stores the result in `c1`. Operations could also be purely classical, e.g., `add(c1,c2)` to add `c1` to `c2` and store the result back in `c1`.
- A *case* statement is a classical conditional. It chooses a branch based on the value of the classical register *creg*. Similar to OpenQASM, *creg* is interpreted as an integer, using the bit at index zero as the low order bit.
- A *choice* statement chooses a candidate statement based on the valuation of choice variable *var*; a value i denotes statement S_i.

MQCC is a meta-language, in the sense that a meta-program P's semantics is determined by the quantum program that remains once its choice variables are decided. Let σ be a map from choice variables *var* to their values i. We can reduce P to a normal program by replacing each $\textbf{choice}(var)\{\overline{i:\ S_i}\}$ statement with (recursively reduced) branch S_k when $\sigma(var) = k$. The reduced program is trivially compiled to an equivalent OpenQASM program.

4.2 Attribute Semantics

An attribute A is particular characterization of a quantum program's execution. An attribute is defined according to a tuple (T, `empty`, `op`, `case`, `value`). Here, T is the type of the *state* of attribute A, and we can view a program statement S as an *attribute state transformer*: Given an initial state s and a valuation of choice variables to values σ, we say program statement S will produce attribute state s' when $[[S]]_A\ (\sigma, s) = s'$. Rules for computing the attribute state are given in Figure 6.

In the rules, we write $A.\mathtt{x}$ to refer to the `x` element of the attribute A's tuple. Each of these elements we define as follows:

[3]We omit **module** definitions and register arrays (e.g., **qreg** $q1\,[10]$) from the formal definition, which can be easily encoded.

$$
\begin{array}{rcl}
& & n \in \mathbb{N} \quad i \in \mathbb{Z} \quad r \in \mathbb{R} \quad var \in Vars \quad op \in OpID \\
& & qreg \in Quantum\ reg. \quad creg \in Classical\ reg. \\
reg & ::= & qreg \mid creg \\
P \in Program & ::= & \overrightarrow{D}\ S \\
D \in Declaration & ::= & RegDecl \mid VarDecl \\
RegDecl & ::= & \textbf{qreg}\ qreg; \mid \textbf{creg}\ creg; \\
VarDecl & ::= & \textbf{fcho}\ var = \{\ \overrightarrow{i}\ \}; \mid \textbf{fcho}\ var = [i_1, i_2]; \\
S \in Stmt & ::= & \epsilon \mid O \mid case \mid choice \mid S; S \\
O \in Operation & ::= & op(\overrightarrow{r}, \overrightarrow{reg}) \\
case & ::= & \textbf{case}(creg)\{\overline{i:\ S_i}\} \\
choice & ::= & \textbf{choice}(var)\{\overline{i:\ S_i}\}
\end{array}
$$

Fig. 5. Formal syntax of MQCC meta-programs.

$$
\frac{S = op(exps,\ regs)}{[\![S]\!]_A\,(\sigma, s) = A.\texttt{op}(s, op, exps, regs)}
\qquad
\frac{}{[\![S_1; S_2]\!]_A\,(\sigma, s) = [\![S_2]\!]_A\,(\sigma, [\![S_1]\!]_A\,(\sigma, s))}
$$

$$
\frac{S = \textbf{case}(creg)\{\overline{i:\ S_i}\}}{[\![S]\!]_A\,(\sigma, s) = A.\texttt{case}(s, creg, [\,[\![S_i]\!]_A\,(\sigma, s)]_i)}
\qquad
\frac{S = \textbf{choice}(var)\{\overline{i:\ S_i}\} \qquad k = \sigma[var]}{[\![S]\!]_A\,(\sigma, s) = [\![S_k]\!]_A\,(\sigma, s)}
$$

Fig. 6. The semantics of MQCC program as an attribute-transformer over the program's statement S, using attribute A. $\sigma[var]$ is the valuation of choice variable var.

- T is the type of an attribute's state used to compute the cost.
- `empty` : T is the initial (empty) state.
- `op` : $T \times (OpID \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg}) \to T$ takes a state and an operation (its name and arguments), and produces a new state. It is used in the first rule of Figure 6.
- `case` : $T \times reg \times \overrightarrow{T} \to T$ takes a state, the guard choice register, and a list of states corresponding to each case branch, and generates the new state. It is used in the third rule of Figure 6.
- `value` : $T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

We define the tuples of two example attributes, **Fidelity** and **QubitCount**, in Section 4.4.

An attribute A's cost for a particular valuation of choice variables σ is simply $A.\texttt{value}([\![S]\!]_A\,(\sigma, A.\texttt{empty}))$. We want to generate a formula that expresses all possible costs, so the SMT solver can decide what choice-variable valuation to use. We do so as follows. Let $\Sigma \subset (Vars \to \mathbb{Z})$ be the variables' possible valuations, then cost_A of attribute A is a function that maps an MQCC program into an expression over $Vars$:

$$\text{cost}_A(S) = \sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot A.\texttt{value}([\![S]\!]_A\,(\sigma, A.\texttt{empty})).$$

Here δ is a variant of the Kronecker delta function: $\delta_{Vars,\sigma} = \prod_{var \in Vars} \delta_{var}^{\sigma[var]}$, and δ_{var}^{i} is a unit expression that contains variable var, which equals 1 if var's value is i, and 0 otherwise. An example formula was given in Section 3.3, for attribute **QubitCount**. Each term in the formula is the depth for a different possible choice of σ—only one term will be non-zero for a given σ.

$$\frac{S = op(exps, regs)}{\text{cost}^+_A(S) = A.\texttt{value}(A.\texttt{op}(A.\texttt{empty}, op, exps, regs))} \qquad \frac{}{\text{cost}^+_A(S_1; S_2) = \text{cost}^+_A(S_1) + \text{cost}^+_A(S_2)}$$

$$\frac{S = \textbf{case}(creg)\{\overline{i:\ S_i}\} \quad S_i \text{ is } \textbf{choice}\text{-free}}{\begin{array}{c}\text{cost}^+_A(S) = \\ A.\texttt{value}(A.\texttt{case}(A.\texttt{empty}, creg, \left[[\![S_i]\!]_A\,(\sigma_\phi, A.\texttt{empty})\right]_i))\end{array}} \qquad \frac{S = \textbf{choice}(var)\{\overline{i:\ S_i}\}}{\text{cost}^+_A(S) = \sum_{i\in\bar{i}} \delta^i_{var}\, \text{cost}^+_A(S_i)}$$

Fig. 7. The cost expression of choice-in-case-free S for *additive* attributes. Here $\bar{i}$ is the set of enumerated values that variable *var* can take.

4.3 Additive Attributes

In general, the generated cost expression $\text{cost}_A(S)$ has a size exponential in the number of choice variables in S. We can use *additive attributes* to reduce this size.

Let σ_ϕ denote an arbitrary valuation of choice variables. Then an attribute A is additive if it satisfies two conditions:

(1) for any $s : T$, *op*, and valid *exps* and *regs*, we have

$$A.\texttt{value}(A.\texttt{op}(s, op, exps, regs)) = A.\texttt{value}(s) + A.\texttt{value}(A.\texttt{op}(A.\texttt{empty}, op, exps, regs));$$

(2) for any $s : T$ and **choice**-free statements S_i, we have

$$\begin{aligned} &A.\texttt{value}(A.\texttt{case}(s, creg, [[\![S_i]\!]_A\,(\sigma_\phi, s)]_i)) \\ &= A.\texttt{value}(s) + A.\texttt{value}(A.\texttt{case}(A.\texttt{empty}, creg, [[\![S_i]\!]_A\,(\sigma_\phi, A.\texttt{empty})]_i)) \end{aligned}$$

We directly derive the cost expression $\text{cost}^+_A(S)$ from the rules in Figure 7 for S that contain no **choice** statements inside branches of **case** (so as to meet the second condition). Let V be the maximal number of possibilities of a variables' values. Notice that $\text{cost}_A(S)$ has $O(V^d)$ terms where d is the number of choice variables, and $\text{cost}^+_A(S)$ has at most $O(|S| \cdot V)$ terms where $|S|$ is the number of constructs of S.

The following theorem shows the correctness of cost^+_A. Its proof is based on induction on S and provided in Appendix A.1.

THEOREM 4.1. *For a statement S such that there is no* ***choice*** *nested in* ***case****, we have $\text{cost}^+_A(S) = \text{cost}_A(S)$ for any valid valuation $\sigma \in Vars$.*

4.4 Examples of Attributes

Here we present two attributes, **Fidelity** and **QubitCount**, used in the FQEC problem in Section 3. We have developed five additional attributes in the case studies in Section 5. Here we use mathematical notation; in our implementation(Section 3.5), programmers use Python classes.

Fidelity. Here we define the **Fidelity** attribute to characterize a circuit program's chances of not producing an error.

```
T = ℝ
empty = 0.0
value(s:T) = s
op(s:T, Op:OpID, exps:ℝ⃗, regs:reg⃗) = s + log(1 - calNoise(Op,exps,regs))
case(s:T, creg:reg, sbs:T⃗) = min sbs
```

The type T of **Fidelity**'s state is $\mathbb{R}$, i.e., the state is a real number, representing the $log(1-P)$ where P is the probability any error occurs. The `empty` state is 0.0—an empty circuit program will never introduce error so $P = 0$ and $log(1 - P) = 0.0$. The **Fidelity** attribute's cost is the fidelity itself, so the `value` function simply returns its argument `s`. The remaining two elements, `op` and `case`, define the fidelity of the program's basic building blocks:

- The `op` function increases the program's total fidelity by the given operation's fidelity (which depends on the operation and the qubits it uses). This fidelity is calculated by the function `calNoise`, which can be implemented variously based on the target machine.
- For the `case` function, the parameter `sbs` refers to the fidelity computed for each branch of the **case**. Since we do not know which branch will be chosen in run-time, we conservatively use the min of these.

Fidelity is an additive attribute so MQCC can generate an optimized cost expression cost^+_A. We can see that for the example in Section 3.3.

QubitCount. The second attribute used in Section 3 was **QubitCount**, which characterizes the maximum count of operations applied to any qubit in a circuit program. Here is its formal definition:

```
T = Set[$Qubit$]
empty = $\emptyset$
value(s:$T$) = |s|
op(s:$T$, Op:$OpID$ exps:$\overrightarrow{\mathbb{R}}$, regs:$\overrightarrow{reg}$) = s$\cup\{q \mid q \in$ regs$, q$ is a qubit$\}$
case(s:$T$, creg:$reg$, sbs:$\overrightarrow{T}$) = $\bigcup_{a \in \text{sbs}} a$
```

The type T of **QubitCount**'s state is a set of qubits. The `empty` element of **QubitCount** is an empty set. The cost of **QubitCount** is the cardinality of the qubit set. The elements `op` and `case` are defined thus:

- The `op` function unions the original qubit set s with the set that contains the qubits used in the input operation Op.
- The `case` function unions the qubit sets of all branches; we do not know which branch will be chosen at run-time, so we must conservatively remember them all.

QubitCount is not an additive attribute, so we must enumerate all possible valuations when computing the cost; an example formula is shown in Section 3.3.

4.5 Limitations

MQCC is limited in the optimization problems it can express. In particular, MQCC restricts choice variables to a finite number of predefined options. Thus, it cannot represent tradeoffs that consider an infinite number of choices; e.g., it cannot decide the real-valued rotation angle of a parameterized gate. Moreover, non-additive attributes cannot be used when a large number of choice variables is involved, for scalability reasons. Despite these limitations, MQCC can express a variety of interesting problems useful for near-term architectures, as the next section shows. Moreover, programming attributes in Python afford a fair degree of flexibility; we implemented the eight attributes in this paper without any difficulty.

5 CASE STUDIES

We evaluate MQCC's utility by evaluating its use in five case studies. The first considers MQCC's performance on the FQEC problem. The second develops a new optimization that trades off accuracy for circuit volume in QFT and QPE implementations; we show that traditional by-hand approaches fare worse than MQCC's approach. The third and the fourth encode previously proposed optimizations for multi-programming and crosstalk mitigation [Das et al. 2019; Murali et al. 2020], while the fifth is a novel combination of the third and the fourth cases, showcasing how MQCC facilitates composition.

We apply these optimizations to a benchmark of NISQ-ready programs and find that MQCC runs quickly, and the optimized programs demonstrate the intended effects when run on real quantum hardware (matching or bettering previously reported results). We apply MQCC's optimizations to the middle-scale benchmark suite collected from QASMBench [Li et al. 2021], and a large scale QFT circuit; these programs are too big to run on today's quantum hardware. We find that even with these larger programs, MQCC scales well. Finally, we apply a sensitivity study for MQCC to analyze how the input noise parameters affect the programs generated by MQCC for CM and MP tasks.

5.1 Fault-tolerant Quantum Error Correction

We describe the fault-tolerant quantum error correction (FQEC) problem and MQCC's programmed solution in Section 3. The definitions of its relevant attributes **Fidelity** and **QubitCount** are given in Section 4.4.

Evaluation. Existing quantum hardware does not support a complete quantum error correction procedure and the common evaluation for quantum error correction is based on hypothetical machine errors and simulators. We follow the same evaluation methodology as Chao and Reichardt [2018]; Reichardt [2020], which simulate error correction using a standard depolarizing noise error model and collect the logical failure rate of different syndrome extraction circuits. The standard depolarizing noise model [Knill 2005a] is described in Section 3.2.2. In the evaluation, we assume the "depolarizing" error e_p of all qubits is in Gaussian Distribution $G(\mu, \sigma)$. We use the Qiskit noise simulator to simulate the error correction procedure using the $[\![5, 1, 3]\!]$ code for 10^5 rounds based on three syndrome extraction strategies: (1) extracting syndromes by Chao and Reichardt [2018]'s circuit (Fig 3(c)) sequentially; (2) extracting syndromes in parallel by Reichardt [2020]'s circuits (Fig 3(d)) and (3) extracting syndromes by circuits produced by MQCC.The error correction procedure of $[\![5, 1, 3]\!]$ code is a small-size circuit ($7 \sim 8$ qubits, < 100 gates) and MQCC solver gives its solution instantly.

Fig 8 shows the logical error rate of different strategies with various σ/μ. The result is intuitive: under the same μ, larger σ means that some qubits have very bad fidelity; the sequential strategy performs better since it uses fewer qubits and can avoid using them. MQCC also tends to generate programs close to the sequential strategy in this case. When σ is small, parallel strategy becomes better since it uses fewer qubit measurements, and MQCC tends to generate programs close to the parallel strategy. The evaluation shows that MQCC can always achieve the minimum logical error rate given various qubit error rates.

5.2 Trade-off between Accuracy and Resources

The accuracy of a numeric computation is limited by the resources devoted to computing it. We can use MQCC to balance the accuracy/resource tradeoff.

5.2.1 Approximate QFT. Quantum Fourier Transform is a crucial part of many quantum information processing algorithms. Consider the n-qubit Quantum Fourier Transform circuit shown in Fig 9. This circuit has $O(n^2)$ gates. However, there are many rotations by small angles that do not affect the final result very much. The standard way to implement an Approximate Quantum Fourier Transform (AQFT) is by pruning these small-angle rotation gates [Barenco et al. 1996]. Given a unitary U and its approximate one V, we use $||U - V||$ to estimate the standard distance between the exact circuit and the approximate one, where $|| \cdot ||$ is the spectral norm. We apply the union bound to upper bound the distance between circuits by the sum of the distance between corresponding gates. For each qubit q_j, if we prune the gates R_k, where $k > h_j$ for a threshold h_j, the approximation

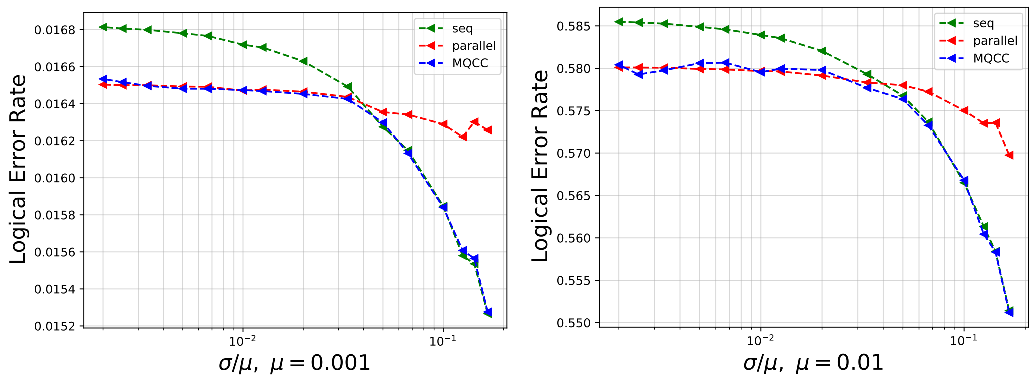
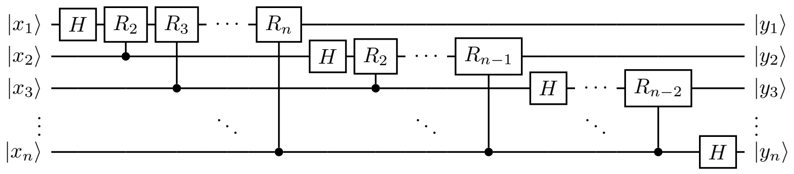


Fig. 8. Logical error rates for simulated error correction compared with previous strategies (Lower is better); using the parallel syndrome-extraction circuit of Fig 3(d), in red, the sequential syndrome-extraction circuit of Fig. 3(c), in green, and the syndrome-extraction produced by MQCC, in blue. Errors are from a standard depolarizing noise model [Knill 2005a], with the depolarizing error of all qubits in Gaussian distribution $G(\mu, \sigma)$.



$$R_k := \begin{pmatrix} 1 & 0 \\ 0 & \omega_k \end{pmatrix}$$

$$\omega_k = exp(2\pi i/2^k)$$

Fig. 9. Quantum circuit for QFT algorithm

error ϵ_j on q_j can be estimated by $\epsilon_j < \frac{1}{2^{h_j}}$. We aim to keep the total approximation error less than a threshold ϵ_{QFT}; i.e., $\sum_{i=1}^{n} \epsilon_i < \epsilon_{QFT}$.

Our MQCC goal is to minimize the gate count in the AQFT circuit while satisfying the accuracy bound ϵ_{QFT}. To do this, we allocate one choice variable for each qubit in AQFT to decide its threshold h_j. The MQCC program generated by the transpiler is

```
\\n-bit AQFT
\\controlled phase rotation gates for q[1]
choice({0,1,2,...}){
    0 : CRZN(h1_0, q[1]);
    1 : CRZN(h1_1, q[1]);
    2 : CRZN(h1_2, q[1]);
    ...
}
\\controlled phase rotation gates for q[2]
choice({0,1,2,...}){
    0 : CRZN(h2_0, q[2]);
    1 : CRZN(h2_1, q[2]);
    2 : CRZN(h2_2, q[2]);
    ...
} ...
```

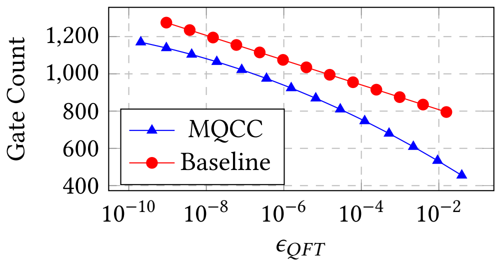


Fig. 10. Trade-off between the gate count and the accuracy for 50-qubit AQFT circuit. Lower gate count is better.

Qubits	Space-time Volume	Running time
20	1.84k	2.28s
30	4.08k	8.85s
40	7.04k	12.93s
50	11k	39.33s

Fig. 11. MQCC's running time on various size AQFT circuits.

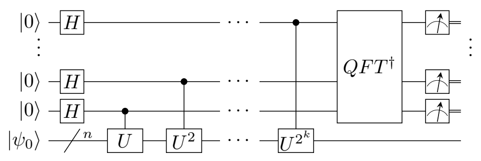


Fig. 12. Quantum circuit performing Quantum Phase Estimation on an n-qubit system with an accuracy of $k+1$ bits. U is the given oracle unitary.

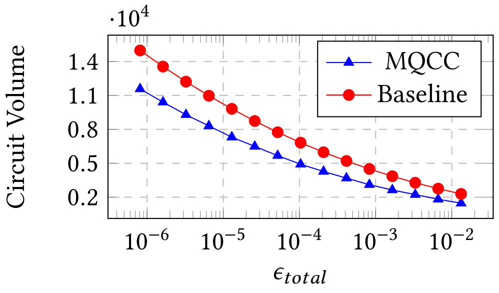


Fig. 13. Trade-off between circuit volume and the precision for QPE circuit. Here lower circuit volume is better.

Term `{0,1,2,...}` in the `choice` statement indicates an *anonymous* (undeclared) choice variable whose value ranges in `{0,1,2,...}`. Term `CRZN(h, q[j])` is a module representing the various controlled phase rotation gates on qubit `q[j]`, controlled by qubits `q[j+1],...,q[h]` (so small angle gates controlled by `q[h+1],...,q[n]` are pruned). The parameters `ha_b` (i.e., `h1_0`, `h1_1,...`) in the `CRZN` specify the threshold. We also define two new attributes, **Approximation** and **GateCount**, for MQCC to estimate program's approximation error and the total gate count, respectively. (Both attributes are additive, and similar to **Fidelity**.)

For evaluation, we use MQCC to minimize a 50-qubit AQFT's gate count, given various ϵ_{QFT}. We compare against a baseline result is produced by pruning a fixed number—call it h_s—of small angle gates for each qubit [Barenco et al. 1996]. We chose h_s by enumerating all possible values and choosing the one minimizes the circuit's gate count. Fig 10 graphs the result, which shows that MQCC cuts down more gates than the naive pruning strategy since MQCC adjusts the pruning threshold for each qubit independently.

5.2.2 Quantum Phase Estimation. Quantum Phase Estimation (QPE) [Nielsen and Chuang 2002] is a direct application of QFT. It estimates the eigenphases of an oracle unitary transformation. Consider the implementation in Fig 12. The top k qubits yield a k-bit approximation error to the phase. The value of k to choose for QPE depends on the desired accuracy [Meuli et al. 2020]. To make the success probability of QPE reach 50%, the desired accuracy bound ϵ_{QPE} can be bounded by k as $\epsilon_{QPE} \leq 16\pi/(2^k - 1)$. QPE requires an AQFT in the final step. Therefore, to achieve an overall target accuracy bound ϵ_{total}, we need to keep $\epsilon_{QPE} + \epsilon_{QFT} < \epsilon_{total}$.

Our MQCC goal is to minimize the circuit's volume but ensure that the circuit's approximation error does not exceed the desired accuracy bound. To achieve this, a choice variable is used to decide the value of k, and the choice of various k can be encoded by the following MQCC program

```
choice({0,1,...}){
    0:  Control_U(k0);
        AQFT(k0);
    1:  Control_U(k1);
        AQFT(k1);
    ... }
```

Here, `Control_U(k)` represents the list of controlled oracle gates in the QPE (controlled-U, U^2, ..., U^{2^k} in Fig 12). The input parameter `k` is an integer and represents the number of controlled qubits used in the QPE circuit. `k0,k1,...` are specified choice for k. This program reuses the code in the AQFT examples as the `AQFT` module so that MQCC can figure out how many qubits are required and how many small angle rotation gates will be removed from the AQFT simultaneously. We reuse the **Approximation** attribute defined in the AQFT example to calculate the circuit's approximation error. We also define a **Volume** attribute for MQCC to calculate the circuit's volume.

For evaluation, we use MQCC to minimize a QPE circuit's volume given various ϵ_{total}. The number of qubits in the QPE ranges in $15 \sim 30$, and Fig 13 shows the result. The baseline result is produced by the natural optimization idea for a circuit with multiple parts: divide ϵ_{total} into $\gamma\epsilon_{total} + (1-\gamma)\epsilon_{total}$ with some appropriate ratio $\gamma \in (0,1)$, then use $\gamma\epsilon_{total}$ and $(1-\gamma)\epsilon_{total}$ as thresholds to optimize the controlled unitary part and the AQFT part in QPE separately. We decide γ by enumerating $\gamma \in (0,1)$ and the one that minimizes the circuit's volume is chosen. The experiment shows that MQCC can cut down more circuit volume, especially in small ϵ_{total} cases. We also measure MQCC's running time on AQFT circuits with large volume. A circuit's space-time volume is defined as the multiplication of its depth and qubit count [Fowler et al. 2012]. Fig 11 shows the result.

5.3 Multi-Programming Quantum Computers

Quantum computer multi programming was proposed by Das et al. [2019]. The idea is simple: Given two quantum circuits A and B, instead of running A to completion and then B, we can create a combined circuit $A + B$ that allocates A and B to distinct qubits so they can be run in parallel. Doing so better utilizes the computer but may decrease the quality of the result. This is because different qubits of a NISQ computer have different error rates; running A and B serially on the highest-fidelity qubits will reduce overall error.

The goal of the *multi-programming* (MP) problem is to maximize utilization while keeping the fidelity above a stated threshold θ. Whether to run in serial or in parallel depends on the programs A and B, the noise characteristics of the hardware, and θ. Das et al. develop a custom solver for this problem; we can program it using the MQCC framework.

Solution of MQCC. In this case, we define the relevant attribute **Depth** and reuse the attribute **Fidelity** defined in Section 4.4. The **Depth** attribute applied to S computes, for each qubit, the length of the sequence of S's gates operating on that qubit, and then returns the maximum over all qubits. In the multi-programming problem, minimizing a circuit's depth will minimize the time that the quantum chip needs to finish executing all programs. In this problem, given a specific group of programs that need to run, the workload of the quantum chip is fixed and minimizing depth is equivalent to maximizing utilization of the chip. MQCC's goal is to minimize **Depth** and the constraint is keeping **Fidelity** above threshold θ.

```
module Bell1(q1,q2, r){
    reset(q1,q2);
    h(q1);
    cnot(q1, q2);
    measure(q1,r[0]);
    measure(q2,r[1]);
}

module Bell2(q1, q2, r){
    reset(q1,q2);
    x(q1);
    h(q1);
    cnot(q1, q2);
    measure(q1,r[0]);
    measure(q2,r[1]);
}
```

```
\\Register and variable declarations
qreg q[10];
creg r1[2], r2[2];
fcho c1 = {0, 1}, c2 = [0, 1];
module Bell1(q1,q2){ ... } \\See Left
module Bell2(q1,q2){ ... } \\See Left
\\Main part of the meta-program
choice (c1){
    0:  Bell1(q[1], q[2],r1);
    1:  Bell1(q[7], q[8],r1);
};
choice (c2){
    0:  Bell2(q[1], q[2],r2);
    1:  Bell2(q[7], q[8],r2);
};
```

Fig. 14. MQCC meta-program for multi-programming two Bell state quantum applications.

We assign a choice variable for each application to decide which qubits are allocated to this application. Different solutions for these choice variables trade off noise for utilization. For example, Fig 14 shows the code of two quantum applications to multi-program. `Bell1` prepares the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, while `Bell2` prepares $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. Both applications need two qubits; we suppose our target computer can schedule them on either qubits {q[1],q[2]} or qubits {q[7],q[8]}. With this information the MQCC transpiler (which uses Qiskit's qubit allocation and mapping library to find low-error areas and routing paths) will produce the meta-program in Fig 4. Then, taking this meta-program, the **Depth** and **Fidelity** attributes, and the optimization goal and constraint ("minimize depth with bounded fidelity"), MQCC will solve for the choice variables to produce a final program that schedules `Bell1` and `Bell2`. Suppose the MQCC solver's *Cost Expression Generator* produce the following formula for the **Depth** and **Fidelity** attribute.

$$\begin{aligned}\textbf{Depth}: &\quad 7\delta^0_{c_1}\delta^0_{c_2} + 4\delta^0_{c_1}\delta^1_{c_2} + 4\delta^1_{c_1}\delta^0_{c_2} + 7\delta^1_{c_1}\delta^1_{c_2}\\ \textbf{Fidelity}: &\quad (-0.045)\delta^0_{c_1} + (-0.066)\delta^1_{c_1} + (-0.027)\delta^0_{c_2} + (-0.043)\delta^1_{c_2}\end{aligned}$$

Here, the term δ^i_c evaluates to 1 if the value of c equals i, and evaluates to 0 otherwise. Referring to Fig 14, we can see that if $c_1 = c_2$ then the two applications will run in sequence, yielding a total depth of 7; otherwise they can run in parallel, and the longest sequence is the max of the two, which is 4. When $c_1 = 0$ and $c_2 = 1$ (running in parallel), **Depth** is 3 while **Fidelity** = $(-0.045) + (-0.043) = -0.088$. When $c_1 = 0$ and $c_2 = 0$ (running in sequence), **Depth** is 5 while **Fidelity** is higher: $(-0.045) + (-0.027) = -0.072$. The result is intuitive: compared to running two applications sequentially on the low error-rate area {q[1],q[2]}, running them in parallel yields a higher error rate but a faster execution time.

Evaluation. We use the same applications (Table 1) and follow the same evaluation methodology as Das et al. [2019]. We package multiple applications as a group and generate several groups. Applications in each group are then executed in three ways: (1) in parallel, as the baseline; (2) in sequential, as the baseline; and (3) multi-programmed by MQCC. Das et al. [2019] does not provide their source code so we do not compare with them directly. The error probability data used by MQCC is collected from the target machine's daily calibration. Prior work [Das et al.

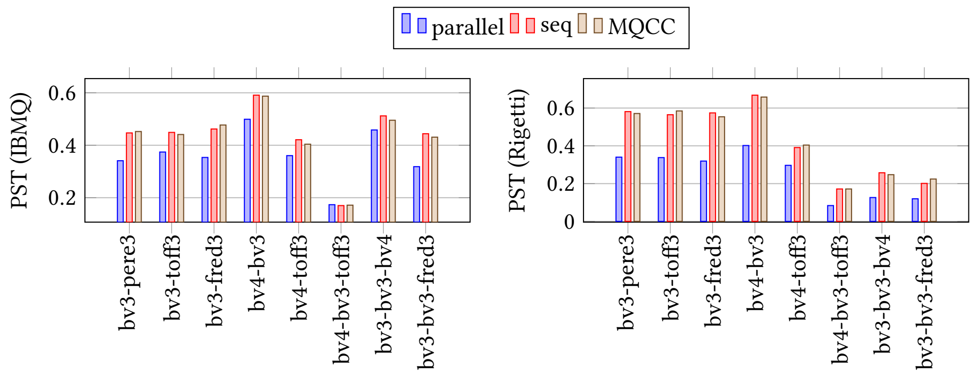


Fig. 15. PST under isolated or multi-programmed execution for each group on IBMQ (left) and Rigetti (right) quantum machines. Group name *A-B* means the group contains two applications *A* and *B*. Similarly for the name *A-B-C*.

2019] multi-programs exactly two applications but MQCC can naturally accept any number; we demonstrate several three-application cases.

For each group, we run 8192 trials on IBMQ Boelingen (20 qubits) and Rigetti Aspen-9 (32 qubits). A trial in which all applications in the group give the correct result is regarded as *successful*. The rate of success - the *Probability of a Successful Trial* (PST) - is used to evaluate the reliability. As Figure 15 demonstrates, since we set a small error threshold in the evaluation, solutions based on MQCC successfully maintain higher reliability of all groups compared to running applications in parallel and are closed to running application in sequential. If we use a larger error threshold, MQCC will produce a solution with PST close to running applications in parallel but with less circuit depth compare to in sequential.

5.4 Circuit Reschedule for Crosstalk Mitigation

Machine-dependent *crosstalk* arises when certain gates are executed in parallel. It is a major source of noise in NISQ systems [Harper et al. 2020; Murali et al. 2020]. Figure 16 shows the layout of the IBMQ Boelingen machine with links between *high crosstalk* gate pairs. Sarovar et al. [2020] propose a protocol for detecting and localizing the crosstalk in hardware. Niu and Todri-Sanial [2021] report several protocols for characterizing crosstalk in NISQ hardware, and discuss different crosstalk mitigation methods in both hardware and software. In software, crosstalk can be avoided by running problematic gates in sequence, but doing so increases circuit depth, which increases the chance of decoherence errors. Murali et al. [2020] propose a software-based method to balance this

Application	**Description**	**Qubits**	**Gates**	**CNOTs**
bv3	Bernstein-Vazirani [Bernstein and Vazirani 1997]	3	8	2
bv4	Bernstein-Vazirani [Bernstein and Vazirani 1997]	4	11	3
h3	Hamiltonian Simulation	3	11	4
h4	Hamiltonian Simulation	4	15	6
Toff3	Toffoli gate	3	15	6
Fred3	Fredkin gate	3	17	8
Pere3	Peres gate	3	16	7

Table 1. Applications used by Das et al. [2019].

tradeoff. They employ an SMT-based scheduler that judiciously decides whether gates should be executed in parallel or serially. The schedule of each gate is encoded by two variables—the gate's start time and its duration—and these in turn are included in an SMT instance that encodes the crosstalk along with other constraints based on features of each gate. Ding et al. [2020] proposes a software solution to alleviate crosstalk by systematically tuning qubit frequencies according to input programs. However, current quantum hardware often does not allow dynamical qubit frequency adjustment; e.g., IBM's current quantum platforms are built with fixed qubit frequencies.

Solution of MQCC. With MQCC We can *program* a solution like Murali et al. [2020]. As with multiprogramming, a transpiler encodes different gate schedules via choice variables. It makes use of the OpenQASM `barrier` operation described in Section 2, which is also used in Murali et al..

For example, consider the following module for a CNOT gate:

```
module cnotb(c,q1,q2){
    choice (c){
        0:  cnot(q1,q2);
        1:  barrier(q);  cnot(q1,q2);
}}
```

When $c = 0$, the gate is applied normally (maximum parallel); Otherwise, suppose the program declares quantum registers with **qreg** `q[n]`. The `barrier(q)` forces the CNOT to be executed sequentially only after all gates on `q` are finished. We encode all those `CNOT`s that use different qubits from the their precursor `CNOT` into this form.

Our goal is to minimize both decoherence error and crosstalk. We define attributes **Crosstalk** and **Decoherence** which take into account the expected appearance of the `barrier` operations in the meta-program.

Evaluation. We follow the evaluation methodology of Murali et al. In particular, we use the same meet-in-the-middle SWAP sequences as their benchmarks. The reason this is a sensible benchmark is that in superconducting QC systems, CNOTs are permitted only between adjacent qubits. To apply a CNOT between two far-away qubits, compilers usually insert a sequence of SWAP operations that move two qubits into adjacent locations through exchanges. For example, in IBMQ Boeblingen, `CNOT 15 8` can be implemented as `SWAP 15 16; SWAP 16 11; SWAP 8 7; SWAP 7 12; CNOT 11 12`.

The IBMQ Poughkeepsie and Johannesburg machines used in the evaluation by Murali et al. are currently unavailable, so we use similar SWAP sequences based on IBMQ Boeblingen. We run 8192 trials for each SWAP sequence and consider those with desired outputs to be successful. We compare the PST of four scheduling strategies: running all instructions serially (`Seq`); running all instructions maximally in parallel (`Par`) which is the default strategy used by Qiskit; the strategy produced by Murali et al. (`Xtalk`); and the strategy produced by MQCC. Figure 17 shows the result. Circuits generated by MQCC always have higher PST than `Seq` and `Par`. They have performance similar to `Xtalk`.

5.5 Multi-programming with Crosstalk Mitigation

In this section, we combine problems from Section 5.3 and Section 5.4 to show MQCC's ability to handle multiple optimization tradeoffs simultaneously.

Motivation. Crosstalk can happen within a single application, but also across different applications that are multi-programmed to be in parallel. MQCC can be used to directly combine the previous crosstalk mitigation and multiprogramming applications (thus informing scheduling decisions by crosstalk-induced noise, but can also include other methods for crosstalk mitigation as well: e.g., transforming circuits into crosstalk-resistant forms.

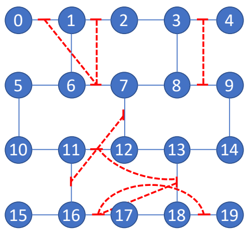


Fig. 16. Layout of IBMQ Boeblingen [Murali et al. 2019a]. Red dashed edges indicate *high crosstalk* gate pairs (e.g., the pair of `CNOT 0 1` and `CNOT 6 7`), where the error caused by their simultaneous execution is much higher than their independent gate error.

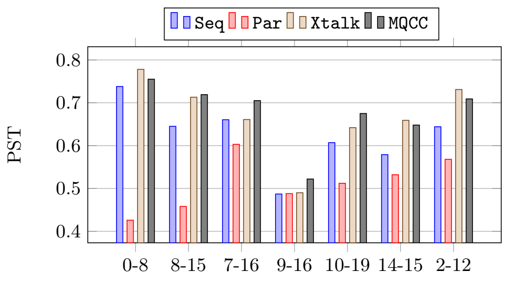


Fig. 17. The measured PST for SWAP circuits on IBMQ Boeblingen using the four schedulers. Higher PST is better. *a*-*b* refers a SWAP circuit connecting qubit *a* and *b*.

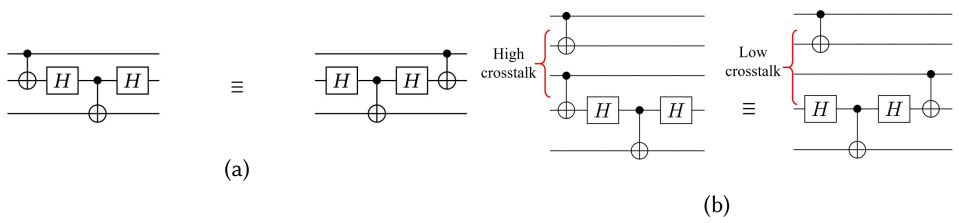
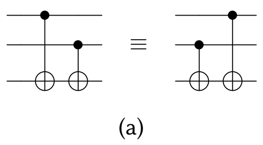


Fig. 18. (a) Two equivalent circuits. (b) The choice of equivalent circuit affects crosstalk with nearby gates.

Figure 18(a) shows two equivalent circuits. In the presence of another CNOT gate, as shown in Figure 18(b), the first structure may introduce much higher crosstalk than the second one since the crosstalk between two CNOT gates is much greater than the crosstalk between a CNOT and a single-qubit gate. One can encode choices between these equivalent forms by MQCC choice variables, to automatically determine the best one to use.

Solution of MQCC. Our goal is to minimize **Depth** while keeping for **Crosstalk** under an upper bound and the **Fidelity** above a lower bound[4], instead of **Crosstalk** only.

Consider equivalent circuit structures as shown in Figure 19(a), the choice of which can be encoded as Fig 19(b):



```
module twoCnot(c,q1,q2,q3){
    choice (c){
        0: cnot(q1,q3); cnot(q2,q3);
        1: cnot(q2,q3); cnot(q1,q3);
    }}
```

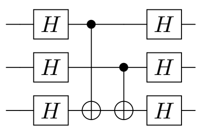
(b)

Fig. 19. (a) Equivalent circuit. (b) Encoding as an MQCC program.

This circuit is a common part of many quantum applications such as Bernstein-Vazirani algorithm (BV) (Figure 20(a)) and Hamiltonian Simulation (HS). For example, the 3-qubit BV circuit in Fig 20(a)

[4]We do not add the **Decoherence** attribute because is effectively represented by **Depth**, which we are minimizing.

can be coded as Fig 20(b). We similarly encode Hamiltonian Simulation applications involving structure in Figure 19(a) into MQCC.



(a)

```
module BV3(c,q1,q2,q3){
    h(q1,q2,q3);
    twoCnot(c,q1,q2,q3);
    h(q1,q2,q3);
}
```

(b)

Fig. 20. (a) 3-qubit BV circuit. (b) MQCC meta-program for the 3-qubit BV circuit.

We then reuse the MQCC setup in Section 5.3 to multi-program these applications. MQCC will determine which form to use for each `twoCnot` in addition to the choice variables for multi-programming.

Evaluation. We use the instances of BV and HS shown in Table 1 for evaluation. This size-restriction is due to the size limit of IBMQ Boebligen. We compare the PST and circuit depth among three scheduling strategies: running benchmarks in serial (`seq`), multi-programmed by MQCC without considering crosstalk (`multi-p`), and when considering crosstalk (`multi-c`). As shown in Fig. 21(b), workloads scheduled by `multi-p` and `multi-c` have lower circuit depth than `seq`. However, as shown in Fig. 21(a), the PST of `multi-p` is lower than both `seq` and `multi-c` because of high crosstalk,[5] where the difference is more significant when multi-programming more applications. In contrast, `multi-c` could always maintain a comparable PST to `seq` while reducing the circuit depth significantly.

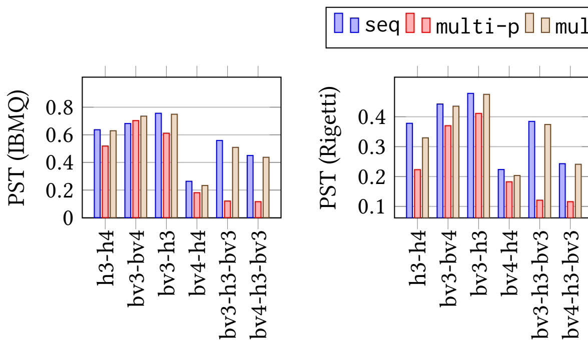
5.6 Scalability Study of MQCC for MP and CM tasks

The running time of MQCC on the small-scale benchmarks in Table 1 is less than 0.01s for all problems. To demonstrate MQCC's scalability, we measure MQCC's running time on some larger benchmarks, which cannot run on current real machines due to hardware constraints. These benchmarks are collected QASMBench [Li et al. 2021], an open-source OpenQASM benchmark suite. QASMBench [Li et al. 2021] collects commonly seen quantum algorithms and routines from a variety of domains with distinct properties. These experiments are carried on Intel Core i7-5960X in Ubuntu 20.10 environment. The running time of each benchmark is the average of three trials.

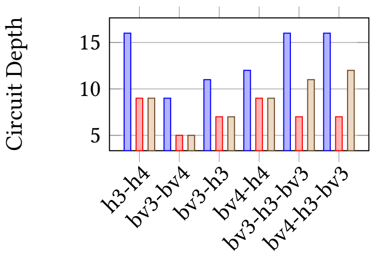
Table 2 compares MQCC's running time with Xtalk's [Cross 2018] running time for mitigating the crosstalk in all QASMBench middle-scale benchmarks. Both running times are measured on the same hardware. MQCC's running time depends more on the circuit's structure rather than its size. For example, the "vqe_uccsd8" is the longest circuit. But most CNOTs in it have to be executed in serial due to topological constraints and cannot cause any crosstalk. So MQCC can find a solution for "vqe_uccsd8" in a reasonable time. On the contrary, Xtalk assigns SMT variables and generates SMT instances for every gate in the circuit, even though some of the gates can never cause crosstalk due to topological constraints. So Xtalk's running time highly depends on the number of gates in the circuit, and it times out when the circuit grows large.

We use MQCC to multi-program QASMBench's middle-scale benchmarks and the left tabular in Table 3 shows MQCC's running time. The complexity of solving MP with MQCC depends on the number of applications to multi-program. Since these benchmarks are too big to run on today's

[5]There is one exception with the case of bv3-bv4 and it might be caused by the fluctuation of the quantum machine.



(a) Probability of Successful Trial (PST) IBMQ Boebligen and Rigetti Aspen-9 . Here higher PST is better.



(b) Circuit Depth. Here lower circuit depth is better.

Fig. 21. Multi-Programming with Crosstalk Mitigation.

quantum hardware, we use the architecture information from qiskit's noise simulator. Then we choose those benchmarks for which MQCC needs more than 0.1s to mitigate their crosstalk (i.e., benchmark 1,3,4,5,6,7,12,13,14,16,17) and group them in pairs to test MQCC's MP-CM running time. We discard benchmark 9 since MQCC already times out($>$ 60min) for mitigating its crosstalk. The right tabular in Table 3 shows this result.

ID	Benchmarks	#Qubits	#Gates	#CNOT	Time (s): MQCC	Time (s): Xtalk
1	adder	10	142	65	**0.1120**	0.2645
2	bv	14	41	13	0.0251	**0.0143**
3	seca	11	216	84	**1.983**	3.567
4	ising	10	480	90	**1.932**	237.8
5	multiply	13	98	40	0.2670	**0.1121**
6	qf21	15	311	115	**0.1097**	22.18
7	qft	15	540	210	**34.98**	12 min
8	qpe	9	123	43	**0.0422**	0.3512
9	sat	11	679	252	Timeout	Timeout
10	cc	12	22	11	0.0231	**0.00312**
11	simons	6	44	14	0.0341	**0.00512**
12	vqe_uccsd6	6	2282	1052	**1.847**	Timeout
13	vqe_uccsd8	8	10808	5488	**15.58**	Timeout
14	qaoa	6	270	54	**0.202**	13.21
15	bb84	8	27	0	0.0092	**0.0021**
16	dnn	8	1008	192	**5.674**	Timeout
17	multiplier	15	574	246	**16 min**	46 min

Table 2. Running time of solving CM by MQCC for all middle-scale circuits in QASMBench. The benchmarks are described in Li et al. [2021]. Timeout is 60 minutes, shortest times in **bold**.

5.7 Sensitivity Study of MQCC for MP and CM tasks

In this section, we discuss how the input parameters for MP and CM tasks affect the structure of output circuits from MQCC.

Fig 22 shows the *Depth Growth Rate* of output circuits from MQCC for benchmarks *vqe_uccsd8, vqe_uccsd6* and *dnn* (the largest three in middle QASMBench, see Table 2) for CM task under different error proportions. In CM task, MQCC receives an input circuit from the user and uses

ID	MP Running time (s)
1-5	0.212
1-8	1.89
1-11	12.6
1-14	101.2
1-17	15min

ID	MP-CM Running time (s)
1,3	2.61
4,5	21.27
6,7	65.12
12,13	36.92
14,16	16.59
16,17	Timeout

Table 3. **Left table:** Running time of multi-programming the benchmarks from a to b in Table 2 by MQCC. **Right table:** Running time of MQCC for handling multi-programming and crosstalk mitigation simultaneously.

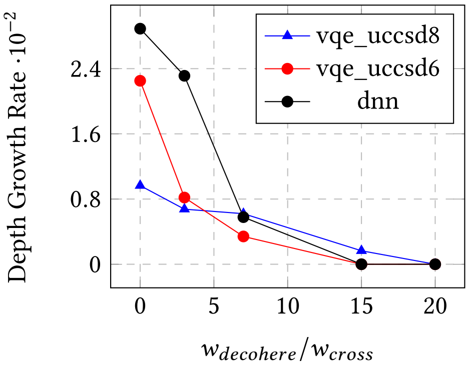


Fig. 22. Depth growth rate of the circuit generated by MQCC for CM task under different error proportion.

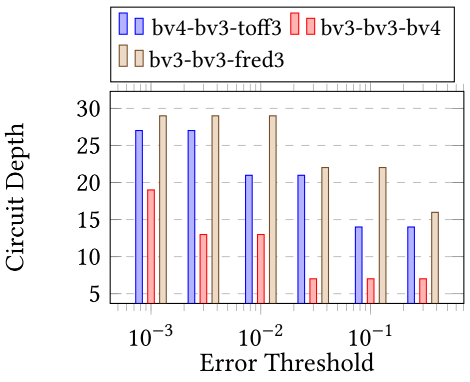


Fig. 23. Circuit Depth under different error threshold for MP task.

`barrier` operations to serialize gates to mitigate crosstalk; these operations increase the circuit's depth. So we define depth growth rate of the circuit C generated by MQCC as $(d_C - d_i)/d_i$, where d_C is C's depth with barriers added and d_i is the depth of the circuit MQCC receives as input. Thus, higher circuit depth growth rate indicates MQCC serializes more gates. $w_{decohere}/w_{cross}$ indicates the proportion of decoherence to crosstalk in the final error: When $w_{decohere}/w_{cross}$ is 0, crosstalk is the only error and MQCC tries to avoid any crosstalk by serializing some gates, which increases the circuit's depth. When $w_{decohere}/w_{cross}$ is 20, decoherence is the dominant error and MQCC decides to run the circuit in maximum parallel to minimize the decoherence, which makes the depth growth rate 0.

Fig 23 shows the depth of circuits generated by MQCC for MP task under different error thresholds. The benchmark is described in Section 5.3. We suppose measurement and qubit reset operations add one unit to the depth. When the input error threshold is 10^{-3} (used for the experiments in Fig 15), MQCC uses qubits with the highest fidelity to run all applications sequentially, which leads to the deepest circuit. When the error threshold becomes larger, MQCC utilizes qubits with lower fidelity to run applications in parallel, leading to circuits with less depth.

6 CONCLUSION

We present MQCC, a meta-programming framework, to assist NISQ application designers to identify the best balance of trade-offs among heterogeneous factors specific to the targeted application and quantum hardware in an automatic way. We also demonstrate MQCC 's expressiveness through an extensive case study, where we showcase MQCC programs that easily implement ideas from previous examples of NISQ application design, either theoretical or with one-off automation, which produces comparable results and exhibits certain advantages.

MQCC constitutes the first step toward a fully automatic design framework for NISQ applications. Ideally, it would be desirable to develop more automation in leveraging MQCC framework from domain problems as well as to improve the scalability of MQCC with more efficient cost expression generations.

ACKNOWLEDGEMENT

This project was partially funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040 and Accelerated Research in Quantum Computing under Award Number DE-SC0020273.

CODE AVAILABILITY

The code for MQCC and the experiments we carried out is available at https://github.com/sqrta/MQCC.

REFERENCES

Dave Bacon. 2006. Operator quantum error-correcting subsystems for self-correcting quantum memories. *Physical Review A* 73, 1 (2006), 012340.

Adriano Barenco, Artur Ekert, Kalle-Antti Suominen, and Päivi Törmä. 1996. Approximate quantum Fourier transform and decoherence. *Physical Review A* 54, 1 (1996), 139.

David Beazley. 2018. PLY (Python Lex-Yacc). https://www.dabeaz.com/ply/.

Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM Journal on computing* 26, 5 (1997), 1411–1473.

A. R. Calderbank and Peter W. Shor. 1996. Good quantum error-correcting codes exist. *Phys. Rev. A* 54 (Aug 1996), 1098–1105. Issue 2. https://doi.org/10.1103/PhysRevA.54.1098

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. *SIGPLAN Not.* 48, 10 (Oct. 2013), 33–52. https://doi.org/10.1145/2544173.2509546

Rui Chao and Ben W Reichardt. 2018. Quantum error correction with only two extra qubits. *Physical review letters* 121, 5 (2018), 050502.

Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS March Meeting Abstracts*, Vol. 2018. L58–003.

Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).

Poulami Das, Swamit S Tannu, Prashant J Nair, and Moinuddin Qureshi. 2019. A case for multi-programming quantum computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 291–303.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

Yongshan Ding, Pranav Gokhale, Sophia Fuhui Lin, Richard Rines, Thomas Propson, and Frederic T Chong. 2020. Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 201–214.

David P DiVincenzo and Panos Aliferis. 2007. Effective fault-tolerant quantum computation with slow measurements. *Physical review letters* 98, 2 (2007), 020501.

Mohammad Javad Dousti, Alireza Shafaei, and Massoud Pedram. 2015. Squash 2: a hierarchical scalable quantum mapper considering ancilla sharing. *arXiv preprint arXiv:1512.07402* (2015).

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13).* Springer-Verlag, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A* 86, 3 (2012), 032324.

Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction.* Ph.D. Dissertation.

Daniel Gottesman. 2010. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, Vol. 68. 13–58.

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *PLDI'11, June 4-8, 2011, San Jose, California, USA* (pldi'11, june 4–8, 2011, san jose, california, usa ed.). https://www.microsoft.com/en-us/research/publication/synthesis-loop-free-programs/

Robin Harper, Steven T Flammia, and Joel J Wallman. 2020. Efficient learning of quantum noise. *Nature Physics* 16, 12 (2020), 1184–1188.

Adam Holmes, Yongshan Ding, Ali Javadi-Abhari, Diana Franklin, Margaret Martonosi, and Frederic T Chong. 2019. Resource optimized quantum architectures for surface code implementations of magic-state distillation. *Microprocessors and Microsystems* 67 (2019), 56–70.

Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative robustness analysis of quantum programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

IBM. 2021. IBM Q Device Information. https://quantum-computing.ibm.com/docs/manage/backends/.

Emanuel Knill. 2005a. Quantum computing with realistically noisy devices. *Nature* 434, 7029 (2005), 39–44.

Emanuel Knill. 2005b. Scalable quantum computing in the presence of large detected-error rates. *Physical Review A* 71, 4 (2005), 042322.

Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. 1996. Perfect quantum error correcting code. *Physical Review Letters* 77, 1 (1996), 198.

Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2021. QASMBench: A Low-level QASM Benchmark Suite for NISQ Evaluation and Simulation. *arXiv preprint arXiv:2005.13018* (2021).

Giulia Meuli, Mathias Soeken, Martin Roetteler, and Thomas Häner. 2020. Automatic accuracy management of quantum programs via (near-) symbolic resource estimation. *arXiv preprint arXiv:2003.08408* (2020).

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 309–328. https://doi.org/10.1145/2660193.2660231

Prakash Murali, Jonathan Baker, Ali Javadi-Abhari, Frederic Chong, and Margaret Martonosi. 2019a. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. 1015–1029. https://doi.org/10.1145/3297858.3304075

Prakash Murali, Ali Javadi-Abhari, Frederic Chong, and Margaret Martonosi. 2019b. Formal Constraint-based Compilation for Noisy Intermediate-Scale Quantum Systems. *Microprocessors and Microsystems* 66 (02 2019). https://doi.org/10.1016/j.micpro.2019.02.005

Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1016.

Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information.

Siyuan Niu and Aida Todri-Sanial. 2021. Analyzing crosstalk error in the NISQ era. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 428–430.

Ben W Reichardt. 2020. Fault-tolerant quantum error correction for steane's seven-qubit color code with few or no extra qubits. *Quantum Science and Technology* 6, 1 (2020), 015007.

Mohan Sarovar, Timothy Proctor, Kenneth Rudinger, Kevin Young, Erik Nielsen, and Robin Blume-Kohout. 2020. Detecting crosstalk errors in quantum information processors. *Quantum* 4 (2020), 321.

Peter W. Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A* 52 (Oct 1995), R2493–R2496. Issue 4. https://doi.org/10.1103/PhysRevA.52.R2493

Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1484–1509. https://doi.org/10.1137/S0097539795293172

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-based Inductive Synthesis for Program Inversion. In *PLDI'11, June 4-8, 2011, San Jose, California, USA* (pldi'11, june 4–8, 2011, san jose, california, usa ed.). https://www.microsoft.com/en-us/research/publication/path-based-inductive-synthesis-program-inversion/

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2009. VS3: SMT Solvers for Program Verification. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 702–708.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *POPL'10, January 17-23, 2010, Madrid, Spain* (popl'10, january 17–23, 2010, madrid, spain ed.). https://www.microsoft.com/en-us/research/publication/program-verification-program-synthesis/

A. M. Steane. 1996. Error Correcting Codes in Quantum Theory. *Phys. Rev. Lett.* 77 (Jul 1996), 793–797. Issue 5. https://doi.org/10.1103/PhysRevLett.77.793

Andrew M Steane. 1997. Active stabilization, quantum computation, and quantum state synthesis. *Physical Review Letters* 78, 11 (1997), 2252.

Andrew M Steane. 2002. Fast fault-tolerant filtering of quantum codewords. *arXiv preprint quant-ph/0202036* (2002).

Andrew M Steane. 2006. A tutorial on quantum error correction. *Quantum Computers, Algorithms and Chaos* (2006), 1–32.

Krysta M Svore, Alfred V Aho, Andrew W Cross, Isaac Chuang, and Igor L Markov. 2006. A layered software architecture for quantum computing design tools. *Computer* 39, 1 (2006), 74–83.

Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T. Chong, and Ronghui Gu. 2021. Gleipnir: Toward Practical Error Analysis for Quantum Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/3453483.3454029

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *SIGPLAN Not.* 49, 6 (June 2014), 530–541. https://doi.org/10.1145/2666356.2594340

Theodore J Yoder and Isaac H Kim. 2017. The surface code with a twist. *Quantum* 1 (2017), 2.

A APPENDIX

A.1 Proof of Theorem 4.1

Proof. Before we start, we show that for any statement S, $\sigma \in \Sigma$ and state $s : T$, where no **choice** is nested inside **case**, there is

$$A.\mathtt{value}(\llbracket S \rrbracket_A(\sigma, s)) = A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S \rrbracket_A(\sigma, A.\mathtt{empty})).$$

Notice that by choosing $s = A.\mathtt{empty}$ in above equation, we have $A.\mathtt{value}(A.\mathtt{empty}) = 0$.

This can be proved by induction on S. For the case that S is an operation or a **case** clause, it is by the additive properties of the attribute. For $S = S_1; S_2$, notice that

$$\begin{aligned}
&A.\mathtt{value}(\llbracket S_1;S_2 \rrbracket_A(\sigma, s)) \\
=\, &A.\mathtt{value}(\llbracket S_2 \rrbracket_A(\sigma, \llbracket S_1 \rrbracket_A(\sigma, s))) \\
=\, &A.\mathtt{value}(\llbracket S_1 \rrbracket_A(\sigma, s)) + A.\mathtt{value}(\llbracket S_2 \rrbracket_A(\sigma, A.\mathtt{empty})) \\
=\, &A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S_1 \rrbracket_A(\sigma, A.\mathtt{empty})) \\
&\qquad + A.\mathtt{value}(\llbracket S_2 \rrbracket_A(\sigma, A.\mathtt{empty})) \\
=\, &A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S_2 \rrbracket_A(\sigma, \llbracket S_1 \rrbracket_A(\sigma, A.\mathtt{empty}))) \\
=\, &A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S_1;S_2 \rrbracket_A(\sigma, A.\mathtt{empty})).
\end{aligned}$$

For $S = \mathbf{choice}(var)\{\overline{i \longrightarrow S_i}\}$, let k be $\sigma[var]$. We equates:

$$\begin{aligned}
&A.\mathtt{value}(\llbracket S \rrbracket_A(\sigma, s)) \\
=\, &A.\mathtt{value}(\llbracket S_k \rrbracket_A(\sigma, s)) \\
=\, &A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S_k \rrbracket_A(\sigma, A.\mathtt{empty})) \\
=\, &A.\mathtt{value}(s) + A.\mathtt{value}(\llbracket S \rrbracket_A(\sigma, A.\mathtt{empty})).
\end{aligned}$$

We now prove the theorem by induction on S. Notice that $\sum_{i \in \Sigma_{var}} \delta^i_{var} = 1$, so we have $\sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot r = r$ for any constant $r \in \mathbb{R}$.

For the base case where S = *opID(exps, regs)*, and $S = \mathbf{case}(creg)\{\overline{i \longrightarrow S_i}\}$, it is true by the above equation.

To show the target for $S = S_1; S_2$, we have

$$\begin{aligned}
&\mathrm{cost}^+_A(S_1;S_2) \\
=\, &\mathrm{cost}^+_A(S_1) + \mathrm{cost}^+_A(S_2) \\
=\, &\mathrm{cost}_A(S_1) + \mathrm{cost}_A(S_2) \\
=\, &\sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot (A.\mathtt{value}(A.\mathtt{empty}) + \mathtt{value}(\llbracket S_1 \rrbracket_A(\sigma, A.\mathtt{empty})) \\
&\qquad\qquad + A/\mathtt{value}(\llbracket S_2 \rrbracket_A(\sigma, A.\mathtt{empty}))) \\
=\, &\sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot \mathtt{value}(\llbracket S_1;S_2 \rrbracket_A(\sigma, \mathtt{empty})).
\end{aligned}$$

Notice that $\sum_{i \in \bar{i}} \delta^i_{var} \delta_{Var,\sigma} = \delta_{Vars,\sigma}$ since

$$\delta^i_{var} \delta_{Var,\sigma} = \delta^i_{var} \delta^{\sigma[var]}_{var} \prod_{u \in Vars \setminus \{var\}} \delta^{\sigma[u]}_u$$

is non-zero only when $i = \sigma[var]$. So for $S = \textbf{choice}(var)\{\overline{i \longrightarrow S_i}\}$ we have

$$\begin{aligned}
&\text{cost}^+_A(S) \\
&= \sum_{i \in \bar{i}} \delta^i_{var} \text{cost}^+_A(S_k) \\
&= \sum_{i \in \bar{i}} \delta^i_{var} \sum_{\sigma \in \Sigma} \delta_{Var,\sigma} A.\texttt{value}(\left[\!\left[S_{\sigma[var]} \right]\!\right]_A (\sigma, A.\texttt{empty})) \\
&= \sum_{\sigma \in \Sigma} \delta_{Var,\sigma} A.\texttt{value}(\left[\!\left[S_{\sigma[var]} \right]\!\right]_A (\sigma, A.\texttt{empty})) \\
&= \text{cost}_A(S).
\end{aligned}$$

□