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

Anonymous Author(s)

Abstract

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 demonstrate MQCC's generality by easily encoding prior one-off NISQ application designs —multi-programming (MP), crosstalk mitigation (CM)— and building new ones, including a combined MP-CM, and an approach to writing approximate quantum Fourier transformation and quantum phase estimation that smoothly trades off accuracy and resource use.

1 Introduction

Quantum computers offer potentially significant performance advantages over classical ones for important classes of problem [15, 27]. 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 [5, 14, 26, 31] consume a substantial number of qubits, so they are not a practical remedy.

Recent work has explored ways to optimize the efficiency and/or reliability of programs run on NISQ platforms by balancing competing tradeoffs. Here are two examples.

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. [9] propose a technique to automatically balance the tradeoff.

Crosstalk mitigation (CM): 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. [24] propose a layout algorithm to balance the tradeoff.

These works both offer mechanisms to relax a program's output fidelity so as to optimize some other attribute of its performance. In this sense, they each support *approximate computing* [6, 18, 21], which aims to make best use of imperfect, noisy hardware.

Unfortunately, these prior optimizations are designed to work in isolation; composing them is not easy, due to conflicting tradeoffs. For example, removing "parallelism" to mitigate crosstalk reduces utilization MP hopes to gain, but also adds a new source of error MP should consider. How can these concerns be integrated? What is needed 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.

As a step toward this goal, we present **Meta Quantum Circuits** with **Constraints** (MQCC), the first general-purpose approximate computing framework for quantum programs.

To use MQCC, developers start by designing various **attributes** of quantum programs that are the basis of optimization. We have implemented seven attributes so far, among them *crosstalk*, *error probability*, *circuit depth*, and *quantum circuit space-time volume* [13, 17]. Then programmers write the program they want to optimize as a collection of normal quantum circuits stitched together by meta-level **choice variables**, which are introduced either manually or automatically. In essence, a program with choice variables is a *family* of programs, and a compile-time valuation of those variables identifies one member of the family. That member is chosen automatically: The MQCC framework analyzes the program with respect to the attributes of interest and a stated optimization goal. In so doing, it generates symbolic expressions over the choice variables that express the program's attributes' values with respect to the goal. MQCC encodes these as SMT formulae and solves for the choice variables, selecting a final program to run on the actual platform. In the worst case, formula sizes are exponential in the number of choice variables (independent of the number of qubits), but we have identified a special case of *additive* attributes whose formulae are linear in the number of choice variables.

We demonstrate MQCC's generality by using it to implement several case studies. First, we implement both the MP

PL'18, January 01–03, 2018, New York, NY, USA
2018.

and CM optimizations listed above; each involves a tradeoff of two attributes. A transpiler automatically introduces choice variables, and the solver produces the final result. We also implement a novel composition of MP and CM; this optimization 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 and implement.

We applied these optimizations to a benchmark of quantum programs, and demonstrated their benefits by running the programs on actual NISQ machines; for CM and MP, we match or improve previously reported results. The new optimization, i.e., a combination of both MP and CM attributes enabled by MQCC, 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 second 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 [19]. MQCC is able to generate the solution for most test cases within a few seconds, with some exceptions in a few minutes.

Finally, we implemented a new optimization to automatically trade accuracy for savings of circuit volume in implementations Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE). Both of these are important subroutines in quantum computing. MQCC's automation is able to identify more efficient gate pruning strategies than typical ones [3] for the entire parameter range. MQCC can generate the solution for QFT/QPE instaces with 50 qubits and 11k space-time volume [13] in 40 seconds.

MQCC can be easily integrated into the existing eco-system of quantum computing tool-chains. MQCC builds on top of OpenQASM [8] 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. [6] 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. [21] made one step further with Chisel, automatically optimizing the tradeoff between reliability and accuracy via integer linear programs. MQCC is inspired by Chisel, in light of its ability to select instructions based on resource consumption. Errors in quantum computing emerge naturally since quantum operations are inherently noisy. Hunget al. [18] and Tao et al. [33] use a logic of quantum robustness to analyze the noise accumulation in quantum programs.

The optimized quantities in these works—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 [12, 29], and specification-based program synthesis [16, 28, 30]. The solver-aided host language Rosette [34, 35] has been designed to ease the construction of solver-aided domain specific languages. Powerful SMT solvers have also been employed to design NISQ applications. In addition to the crosstalk example [24], one can also model the qubit mapping and gate scheduling problems as SMT instances [22, 23]. 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 $|\alpha|^2 + |\beta|^2 = 1$. 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. 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 for 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 "quantum assembly language" (QASM) [11, 32]. QASM is a simple text language that describes quantum circuits as a sequence of gate operations on

numbered qubits. OpenQASM [8] provides a bit more high-level structure, while still being compatible with modern hardware[1]. Here is an example.

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

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 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 this example.

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

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 multi-programming a quantum computer as an example [9].

3.1 Introduction to multi-programming

Quantum computer multi programming was proposed by Das et al. [9]. 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 noise under 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.

3.2 MQCC Overview

MQCC's architecture is shown in Fig 1(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*.

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. Below is an example.

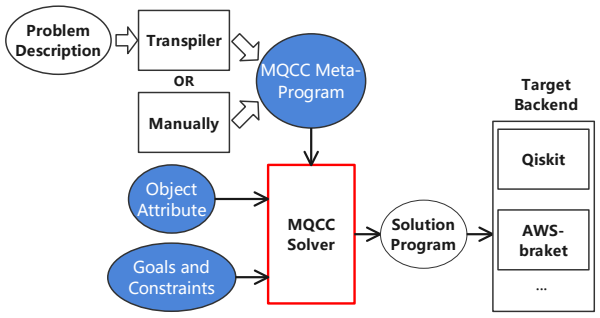
```
1 qreg q[1];
2 fcho c1 = {0, 1};
3 choice (c1) {
4     0: x(q[0]);
5     1: h(q[0]);
6 }
```

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.

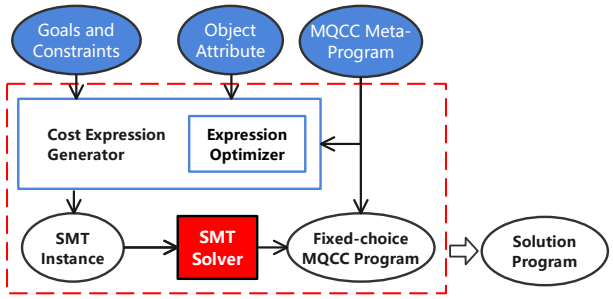
The programmer need not write the meta-program directly; as shown in Fig 1(a), a problem-specific transpiler can produce it from a higher-level problem description. For example, for the MP problem, MQCC implements a transpiler that takes the two programs S and T and produces an $S+T$ program with choice variables that identify the different scheduling choices. MQCC's transpilers often leverage Qiskit [7] code for generating meta-programs.

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. The relevant attributes for the MP problem are **Noise** and **Depth**. The **Noise** attribute applied to a circuit S sums the measured noise on S's gates—noise n is a real number with $0 \le n \le 1$ (lower is better). 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. Programmers can easily define their own attributes, and these can be reused for different programs and problems. Definitions of **Noise** and **Depth** are given in Section 4.

Goal and constraints. The third MQCC solver input is the optimization goal and constraints that must be satisfied.



(a) Overview of MQCC



(b) MQCC Solver

Figure 1. Overview of MQCC

```
1 module Bell1(q1,q2)
2 {
3     h(q1);
4     cnot(q1, q2);
5 }
```

```
1 module Bell2(q1, q2){
2     creg r[1];
3     case (r[0]){
4         1:  x(q1);
5         0:  pass
6     };
7     h(q1); cnot(q1,q2);
8 }
```

Figure 2. Two Bell state quantum applications.

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. For the MP problem, the goal is to minimize **Depth** (thus maximizing utilization) and the constraint is keeping **Noise** below threshold θ.

3.3 Full MP Example

Fig 2 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 $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ or $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ depending on

```
1  \\Register and variable declarations
2  qreg q[10];
3  fcho c1 = {0, 1}, c2 = [0, 1];
4  \\lcho c = 1 - c1 * c2;
5  module Bell1(q1,q2){ ... } \\See Fig. 2
6  module Bell2(q1,q2){ ... } \\See Fig. 2
7  \\Main part of the meta-program
8  choice (c1){
9      0:  Bell1(q[1], q[2]);
10     1:  Bell1(q[7], q[8]);
11 };
12 choice (c2){
13     0:  Bell2(q[1], q[2]);
14     default:  Bell2(q[7], q[8]);
15 };
```

Figure 3. MQCC meta-program for multi-programming

the classical register $r[0]$. 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 3. Then, taking this meta-program, the **Depth** and **Noise** attributes, and the optimization goal and constraint ("minimize depth with bounded error"), MQCC will solve for the choice variables to produce a final program that schedules `Bell1` and `Bell2`.

Let's look more closely at the meta-program. It consists of three parts: registers and variables declarations (lines 1-5); module definitions (lines 7-8); and the main part of the program (lines 10-18).

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

Lines 3-4 declare 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 where to run the two applications `Bell1` and `Bell2` . Apart from free choice variables, sometimes we need to define a choice variable whose value depends on the value of other choice variables. We call such a variable a *limited choice variable*, and it can be defined using keyword **lcho**. The comments on Line 5 show an example. The value of a limited variable c depends on the expression $1 - c1 * c2$. We write "choice variable" for free choice variables, and write "limited" explicitly.

Lines 5-6 designate the two circuit components `Bell1` and `Bell2`, taken from Fig 2. A module in MQCC can be viewed as macro over its parameters.

Lines 8-15 contain the part of the meta-program that expresses the possible schedules. Two **choice** statements decide where to run `Bell1` and `Bell2`, based on the value of choice variables c_1, c_2. The **choice** statement on lines 11-14 says that `Bell1` should run on {q[1],q[2]} if $c1 = 0$ or on {q[7],q[8]} if $c1 = 1$. The **choice** statement on lines 15-18 does similarly for `Bell2`. It is easy to see that when $c_1 = c_2$, `Bell1` and `Bell2` are executed in serial since they are instantiated to the same area. Otherwise they are executed on distinct areas in parallel.

3.4 MQCC Solver

Now let us see how the MQCC solver works. Its operation is shown in Fig 1(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 3, the CEG would produce the following formula for the **Depth** attribute.

$$\textbf{Depth}: \quad 5\delta^0_{c_1}\delta^0_{c_2} + 3\delta^0_{c_1}\delta^1_{c_2} + 3\delta^1_{c_1}\delta^0_{c_2} + 5\delta^1_{c_1}\delta^1_{c_2}$$

Here, the term δ^i_c evaluates to 1 if the value of c equals i, and evaluates to 0 otherwise. Referring to Fig 2, we can see that if $c_1 = c_2$ then the two applications will run in sequence, yielding a total depth of 5; otherwise they can run in parallel, and the longest sequence is the max of the two, which is 3.

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 **Depth**. 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 **Noise** attribute is additive: the noise due to two operations can be estimated as the sum of the operations' individual noise [20]. 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 3, suppose the noise when running `Bell1` on {q[1],q[2]} is 0.045 and the noise on {q[7],q[8]} is 0.066. Then the total noise of the first **choice** statement (lines 11-14) is calculated as $0.045\delta^0_{c_1} + 0.066\delta^1_{c_1}$. Similarly suppose the error of the second **choice** statement (lines 15-18) is calculated as $0.027\delta^0_{c_2} + 0.043\delta^1_{c_2}$. Since the **Noise** attribute is additive, the CEG sums these two:

$$\textbf{Noise}: \quad 0.045\delta^0_{c_1} + 0.066\delta^1_{c_1} + 0.027\delta^0_{c_2} + 0.043\delta^1_{c_2}$$

(How this formula was computed is explained in Section 4.3.) Different solutions for c_1 and c_2 in these formulas trade off noise for utilization. When $c_1 = 0$ and $c_2 = 1$ (running in parallel), **Depth** is 3 while **Noise** $= 0.045 + 0.043 = 0.088$. When $c_1 = 0$ and $c_2 = 0$ (running in sequence), **Depth** is 5 while **Noise** is lower—$0.045 + 0.027 = 0.072$. 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. Which solution is chosen depends on the threshold θ.

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 (Z3 [10]) to assign values to choice variables. In multi-programming case, if the noise threshold is $\theta = 0.090$, MQCC will choose $c_1 = 0, c_2 = 1$, and run the modules in parallel. If the noise threshold is lower, say $\theta = 0.080$, MQCC will choose $c_1 = 0, c_2 = 0$ and run them in sequence.

Scalability of MQCC. The time MQCC takes to generate and solve SMT formulae will depend on the size of the search space, which in turn depends on the number of choice variables, and whether the attributes are additive; non-additive attributes produce formulae that are exponential in the number choice variables, while additive ones are linear. The actual running time of the MQCC solver (including CEG and the SMT solver) is very reasonable for all instances demonstrated in Section 5. In particular, MQCC can easily handle a large number of qubits or gates as long as the number of choice variables is bounded.

3.5 Implementation of MQCC

We implement MQCC in Python using `PLY`[2], 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[10] version 4.8.9. The implementation of MQCC contains about 2K LOC including all of the applications in Section 5. Our code is available at *LINK REDACTED FOR BLIND REVIEW*.

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 4. A meta-program P consists of a sequence of declarations D and a statement S. There are two kinds of declarations:[1] *RegDecl* declares classical and quantum registers, and $VarDecl$ declares MQCC choice variables. A statement S

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

$$n \in \mathbb{N} \quad i \in \mathbb{Z} \quad r \in \mathbb{R} \quad var \in Vars \quad op \in OpID$$
$$qreg \in Quantum\ reg. \quad creg \in Classical\ reg.$$
$$reg ::= qreg \mid creg$$
$$P \in Program ::= \overrightarrow{D}\ S$$
$$D \in Declaration ::= RegDecl \mid VarDecl$$
$$RegDecl ::= \textbf{qreg}\ qreg; \mid \textbf{creg}\ creg;$$
$$VarDecl ::= Free \mid Limit$$
$$Free ::= \textbf{fcho}\ var = \{\overrightarrow{i}\}; \mid \textbf{fcho}\ var = [i_1, i_2];$$
$$Limit ::= \textbf{lcho}\ var = E;$$
$$E \in VarExp ::= i \mid var \mid E + E \mid E - E \mid E * E \mid E/E \mid (E)$$
$$S \in Stmt ::= \epsilon \mid O \mid case \mid choice \mid S; S$$
$$O \in Operation ::= op(\overrightarrow{r}, \overrightarrow{reg})$$
$$case ::= \textbf{case}(creg)\{\overline{i : S_i}\}$$
$$choice ::= \textbf{choice}(var)\{\overline{i : S_i}\}$$

Figure 4. Formal syntax of MQCC meta-programs.

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 $(T, \texttt{empty}, \texttt{op}, \texttt{case}, \texttt{value})$. Here, T is the type of the

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

$$\frac{}{[\![S_1; S_2]\!]_A(\sigma, s) = [\![S_2]\!]_A(\sigma, [\![S_1]\!]_A(\sigma, s))}$$

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

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

Figure 5. 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.

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 5.

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

- T is the type of an attribute's state, used to compute the cost.
- $\texttt{empty} : T$ is the initial (empty) state.
- $\texttt{op} : T \times (OpID \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg}) \to T$ takes a state and an operation (its name and arguments), and produces a new state. It is used in the first rule of Figure 5.
- $\texttt{case} : T \times reg \times \overrightarrow{T} \to T$ 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 5.
- $\texttt{value} : T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

We define the tuples of two example attributes, **Noise** and **Depth**, 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.

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

$$\frac{}{\text{cost}^+_A(S_1; S_2) = \text{cost}^+_A(S_1) + \text{cost}^+_A(S_2)}$$

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

$$\frac{S = \textbf{choice}(var)\{\overline{i :\ S_i}\}}{\text{cost}^+_A(S) = \sum_{i \in \bar{i}} \delta^i_{var}\ \text{cost}^+_A(S_i)}$$

Figure 6. 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.

An example formula was given in Section 3.4, for attribute **Depth**. We can see that 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.

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, and optimize SMT solving performance.

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

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

We directly derive the cost expression $\text{cost}^+_A(S)$ from the rules in Figure 6 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, **Noise** and **Depth**, used in the multi-programming 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.

Noise. Recall that we defined the **Noise** attribute to characterize a circuit program's noise. It is defined thus.

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

The type T of **Noise**'s state is $\mathbb{R}$, i.e., the state is a real number, representing the noise. The `empty` state is 0.0—an empty circuit program has no noise. The **Noise** attribute's cost is the noise itself, so the `value` function simply returns its argument `s`. The remaining two elements, `op` and `case`, define the noise of the program's basic building blocks:

- The `op` function increases the program's total noise by the given operation's noise (which depends on the operation and the qubits it uses). This noise 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 noise computed for each branch of the **case**. Since we do not know which branch will be chosen in run-time, we conservatively use the max of these.

Noise is an additive attribute so MQCC can generate an optimized cost expression cost^+_A. We can see that for the example in Section 3.4.

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

```
T = Map[AllReg,ℕ]
empty = {r ↦ 0 | r ∈ AllReg}
value(s:T) = max values(s)
op(s:T, op:OpID exps:ℝ⃗, regs:reg⃗) =
      s += {r ↦ max {s[a]+1 | a ∈regs} | r ∈regs}
case(s:T, creg:reg, sbs:T⃗) =
      {r ↦ max {t[r] | t ∈sbs} | r ∈ AllReg}
```

The type T of **Depth**'s state is a map from the program's qubits (in the set *AllReg*) to natural numbers $\mathbb{N}$, which count gates applied to that qubit. The `empty` element of **Depth** is a map from any key to 0. The cost of **Depth** is the maximum depth of any qubit in the circuit. Thus the `value` element of **Depth** is the maximum of `values(s)`, which is the map s's range, i.e., its distinct mapped-to natural numbers. The elements `op` and `case` are defined thus:

- The `op` element updates the depth for all registers used in the given operation. We write s+={...} to mean a map that is the same as s but is updated with bindings in {...}. Here, those bindings are for `regs` used in the operation, and their depth is the max depth of all registers involved, after adding 1.
- The `case` element constructs a fresh map that conservatively maps each r to the max depth it has in any of the branch maps `sbs`. (The current map s will have been considered when constructing `sbs` per the **case** rule, Figure 5.)

Depth is not an additive attribute, so we must enumerate all possible valuations when computing the cost; an example formula is shown in Section 3.4.

5 Case Studies

We evaluate MQCC's utility by demonstrating its use in four case studies. The first two encode previously proposed optimizations [9, 24], while the third is a novel combination of the first two, 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 also develop a new optimization the trades off accuracy for circuit volume in QFT and QPE implementations; we show that traditional by-hand approaches fare worse than MQCC's approach. Finally, we apply MQCC's optimizations to the middle scale benchmark suite collected from QASMBench[19] and large scale QFT circuit. All these benchmarks are too big to run on today's quantum hardware. We find that even with larger programs, MQCC scales well.

5.1 Multi-Programming Quantum Computers

We described the multi-programming (MP) problem and MQCC's programmed solution in Section 3. The definitions of its relevant attributes **Depth** and **Noise** are given in Section 4.4. Prior work [9] multi-programs exactly two applications but MQCC can naturally accept any number; we demonstrate several three-application cases.

Evaluation. We use the same applications (Table 1) and follow the same evaluation methodology as Das et al. [9]. We package multiple applications as a group and generate several groups. Applications in each group are then executed in two ways: (1) in parallel, as the baseline; and (2) multi-programmed by MQCC. The error probability data used by MQCC is collected from the target machine's daily calibration.

For each group, we run 8192 trials on IBMQ Rochester and Rigetti Aspen-9. 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 7 demonstrates, solutions based on MQCC successfully maintain higher reliability of all groups compared to running applications in parallel.

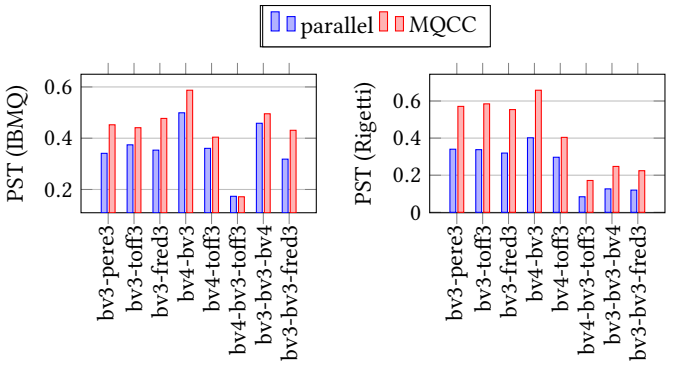


Figure 7. PST under isolated or multi-programmed execution for each group on IBMQ (left) and Rigetti (right) quantum machines. Group name *A-B* means the group contains two applications *A* and *B*. Similarly for the name *A-B-C*.

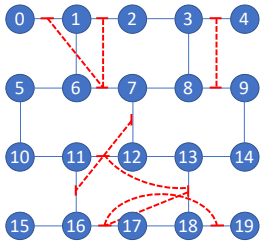


Figure 8. Layout of IBMQ Boeblingen [22]. Red dashed edges indicate *high crosstalk* gate pairs (e.g., the pair of `CNOT 0 1` and `CNOT 6 7`), where the error caused by their simultaneous execution is much higher than their independent gate error.

5.2 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 [24]. Figure 8 shows the layout of the IBMQ Boelingen machine with links between *high crosstalk* gate pairs.

Application	Description	Qubits	Gates	CNOTs
bv3	Bernstein-Vazirani[4]	3	8	2
bv4	Bernstein-Vazirani[4]	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. [9].

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. [24] propose a software-based method to balance this 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.

Solution of MQCC. With MQCC We can *program* a solution like Murali et al.. 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 9 shows the result.

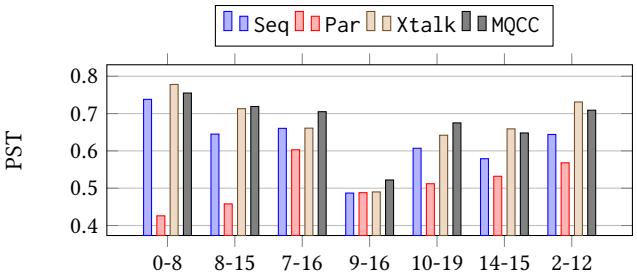


Figure 9. The measured PST for SWAP circuits on IBMQ Boeblingen using the four schedulers. Higher PST is better. *a-b* refers a SWAP circuit connecting qubit *a* and *b*.

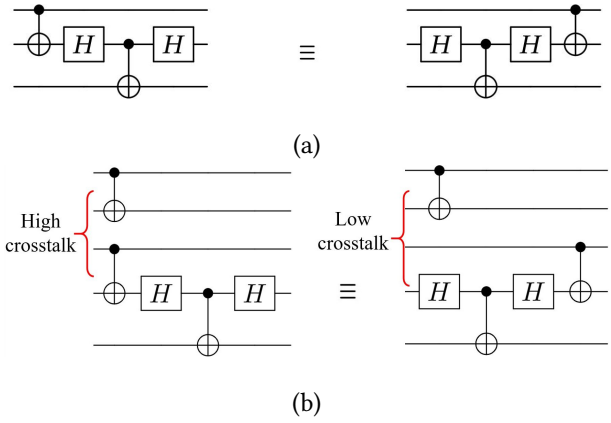


Figure 10. (a) Two equivalent circuits. (b) The choice of equivalent circuit affects crosstalk with nearby gates.

Circuits generated by MQCC always have higher PST than `Seq` and `Par`. They have performance similar to `Xtalk`.

5.3 Multi-programming with Crosstalk Mitigation

In this section, we combine problems from Section 5.1 and Section 5.2 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.

Figure 10(a) shows two equivalent circuits. In the presence of another CNOT gate, as shown in Figure 10(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.

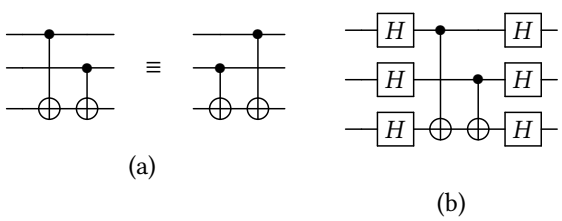


Figure 11. (a) Equivalent Circuits; (b) 3-qubit BV circuit.

Solution of MQCC. Our goal is to minimize **Depth** while bounding error **Crosstalk + Noise**,[2] instead of **Noise** only.

Consider equivalent circuit structures as shown in Figure 11(a), the choice of which can be encoded as follows:

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

This circuit is a common part of many quantum applications such as Bernstein-Vazirani algorithm (BV) (Figure 11(b)) and Hamiltonian Simulation (HS). For example, the 3-qubit BV circuit in Figure 11(b) can be coded as follows:

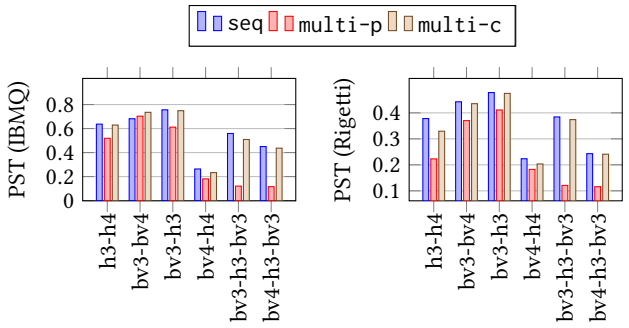
```
module BV3(c,q1,q2,q3){
    h(q1,q2,q3);
    twoCnot(c,q1,q2,q3);
    h(q1,q2,q3);
}
```

We similarly encode Hamiltonian Simulation applications involving structure in Figure 11(a) into MQCC. We then reuse the MQCC setup in Section 5.1 to multi-program these applications. MQCC will determine which form to use for each `twoCnot` in addition to the choice variables for multi-programming.

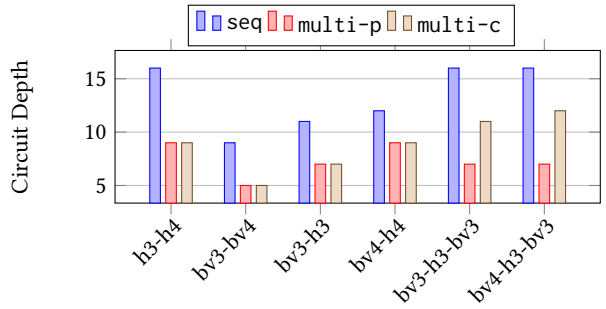
Evaluation. We use the instances of BV and HS shown in Table 1 for evaluation. This size-restriction is due to the size limit of IBMQ Boebligen. We compare the PST and circuit depth among three scheduling strategies: running benchmarks in serial (`seq`), multi-programmed by MQCC without considering crosstalk (`multi-p`), and when considering crosstalk (`multi-c`). As shown in Fig. 12(b), workloads scheduled by `multi-p` and `multi-c` have lower circuit depth than `seq`. However, as shown in Fig. 12(a), the PST of `multi-p` is lower than both `seq` and `multi-c` because of high crosstalk,[3] 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.

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

[3]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.

Figure 12. Multi-Programming with Crosstalk Mitigation.

5.4 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 [19]an open-source OpenQASM benchmark suite. QASMBench [19] 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 runtime of each benchmark is the average of three trials.

Table 2 shows MQCC's runtime for mitigating the crosstalk in all QASMBench middle-scale benchmarks. The runtime 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.

We use MQCC to multi-program QASMBench's middle-scale benchmarks and the left tabular in Table 3 shows MQCC's runtime. The complexity f solving MP with MQCC depends on the number of applications to multi-program. Since these benchmarks are too big to run on today's 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 runtime. We discard benchmark 9 since MQCC is already timeout for mitigating its crosstalk. The right tabular in Table 3 shows this result.

ID	Benchmarks	Qubits	Gates	CNOT	Runtime(s)
1	adder	10	142	65	0.1120
2	bv	14	41	13	0.0251
3	seca	11	216	84	1.983
4	ising	10	480	90	1.932
5	multiply	13	98	40	0.2670
6	qf21	15	311	115	0.10971
7	qft	15	540	210	34.98
8	qpe	9	123	43	0.0422
9	sat	11	679	252	Timeout
10	cc	12	22	11	0.0231
11	simons	6	44	14	0.0341
12	vqe_uccsd6	6	2282	1052	1.847
13	vqe_uccsd8	8	10808	5488	15.58
14	qaoa	6	270	54	0.202
15	bb84	8	27	0	0.0092
16	dnn	8	1008	192	5.674
17	multiplier	15	574	246	16 min

Table 2. Runtime of solving CM by MQCC for all middle-scale circuits in QASMBench. The descriptions for these benchmarks are shown in QASMBench [19].

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

ID	MP-CM Runtime(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 tabular: Runtime of multi-programming the benchmarks from *a* to *b* in Table 2 by MQCC. **Right tabular:** Runtime of MQCC for handling multi-programming and crosstalk mitigation simultaneously.

5.5 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.

5.5.1 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 13. 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 [3]. 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 accuracy ϵ_j on q_j can be estimated by $\epsilon_j < \frac{1}{2^{h_j}}$. We aim to keep the total accuracy 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]);
    ...
} ...
```

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, **Accuracy** and **GateCount**, for MQCC to estimate program's accuracy and the total gate count, respectively. (Both attributes are additive, and similar to **Noise**.)

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 [3]. We chose h_s by enumerating all possible values and choosing the one minimizes the circuit's gate count. Fig 14 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.

5.5.2 Quantum Phase Estimation. Quantum Phase Estimation (QPE) [25] is a direct application of QFT. It estimates the eigenphases of an oracle unitary transformation. Consider the implementation in Fig 15. The top k qubits yield a k-bit approximation to the phase. The value of k to choose for QPE depends on the desired accuracy [20]. To make the success probability of QPE reach 50%, the desired accuracy ϵ_{QPE}

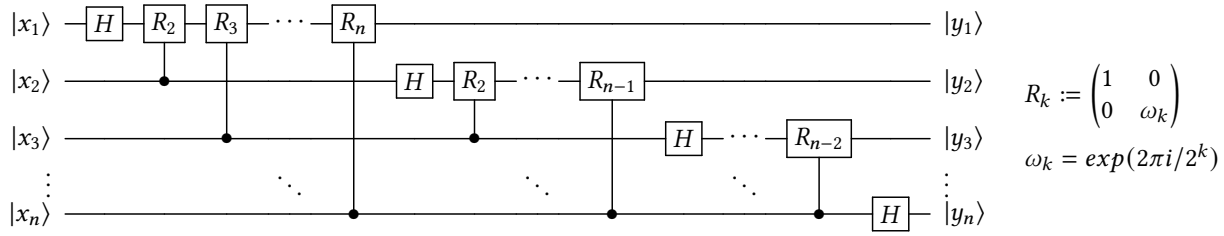


Figure 13. Quantum circuit for QFT algorithm

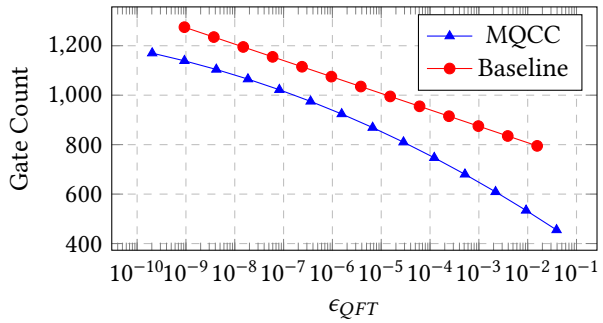


Figure 14. Trade-off between the gate count and the accuracy for 50-qubit AQFT circuit. Lower gate count is better.

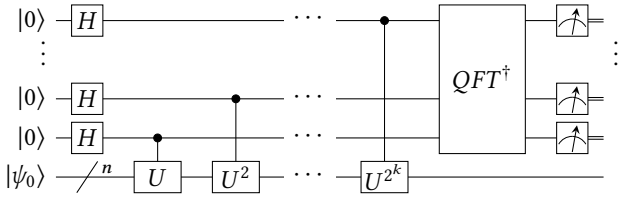


Figure 15. Quantum circuit performing Quantum Phase Estimation on an n-qubit system with an accuracy of $k+1$ bits. U is the given oracle unitary.

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 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 accuracy does not exceed the desired 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 15). The input parameter `k` is an integer and represents the number of controlled qubits used in the QPE circuit. `k0,k1,...` are

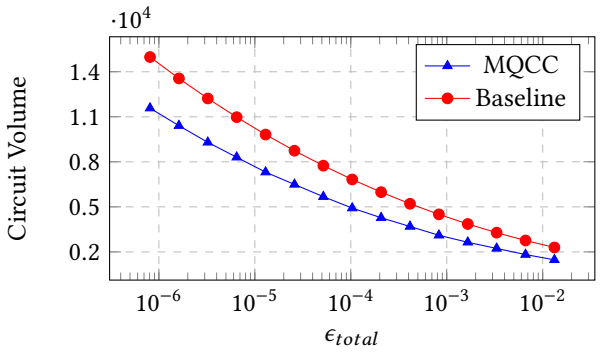


Figure 16. Trade-off between circuit volume and the precision for QPE circuit. Here lower circuit volume is better.

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

Table 4. MQCC's runtime on various size AQFT circuits.

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 **Accuracy** attribute defined in the AQFT example to calculate the circuit's accuracy. We also define a **Volume** attribute for MQCC to calculate the circuit's volume. **Volume** is defined similarly to **Depth**.

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 16 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[13]. Tab 4 shows the result.

References

[1] [n.d.]. IBM Q Device Information. https://quantum-computing.ibm.com/docs/manage/backends/.

[2] [n.d.]. PLY (Python Lex-Yacc). https://www.dabeaz.com/ply/.

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

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

[5] 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

[6] 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

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

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

[9] 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.

[10] 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.

[11] 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).

[12] 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

[13] 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.

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

[15] 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

[16] 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/

[17] 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.

[18] 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.

[19] 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).

[20] 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).

[21] 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

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

[23] Prakash Murali, Ali Javadi-Abhari, Frederic Chong, and Margaret Martonosi. 2019. 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

[24] 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.

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

[26] 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

[27] 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

[28] 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/

[29] 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.

[30] 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/

[31] 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

[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.

[33] 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

[34] 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

[35] 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

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

$$A.\texttt{value}([\![S]\!]_A(\sigma, s)) = A.\texttt{value}(s) + A.\texttt{value}([\![S]\!]_A(\sigma, A.\texttt{empty})).$$

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

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

$$\begin{aligned}
&A.\texttt{value}([\![S_1; S_2]\!]_A(\sigma, s)) \\
&= A.\texttt{value}([\![S_2]\!]_A(\sigma, [\![S_1]\!]_A(\sigma, s))) \\
&= A.\texttt{value}([\![S_1]\!]_A(\sigma, s)) + A.\texttt{value}([\![S_2]\!]_A(\sigma, A.\texttt{empty})) \\
&= A.\texttt{value}(s) + A.\texttt{value}([\![S_1]\!]_A(\sigma, A.\texttt{empty})) \\
&\qquad + A.\texttt{value}([\![S_2]\!]_A(\sigma, A.\texttt{empty})) \\
&= A.\texttt{value}(s) + A.\texttt{value}([\![S_2]\!]_A(\sigma, [\![S_1]\!]_A(\sigma, A.\texttt{empty}))) \\
&= A.\texttt{value}(s) + A.\texttt{value}([\![S_1; S_2]\!]_A(\sigma, A.\texttt{empty})).
\end{aligned}$$

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

$$\begin{aligned}
&A.\texttt{value}([\![S]\!]_A(\sigma, s)) \\
&= A.\texttt{value}([\![S_k]\!]_A(\sigma, s)) \\
&= A.\texttt{value}(s) + A.\texttt{value}([\![S_k]\!]_A(\sigma, A.\texttt{empty})) \\
&= A.\texttt{value}(s) + A.\texttt{value}([\![S]\!]_A(\sigma, A.\texttt{empty})).
\end{aligned}$$

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 $S = \textbf{case}(creg)\{\overline{i \to S_i}\}$, it is true by the above equation.

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

$$\begin{aligned}
&\text{cost}^+_A(S_1; S_2) \\
&= \text{cost}^+_A(S_1) + \text{cost}^+_A(S_2) \\
&= \text{cost}_A(S_1) + \text{cost}_A(S_2) \\
&= \sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot (A.\texttt{value}(A.\texttt{empty}) + \texttt{value}([\![S_1]\!]_A(\sigma, A.\texttt{empty})) \\
&\qquad + A/\texttt{value}([\![S_2]\!]_A(\sigma, A.\texttt{empty}))) \\
&= \sum_{\sigma \in \Sigma} \delta_{Vars,\sigma} \cdot \texttt{value}([\![S_1; S_2]\!]_A(\sigma, \texttt{empty})).
\end{aligned}$$

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 $S = \textbf{choice}(var)\{\overline{i \to S_i}\}$ we have

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

□