Automating NISQ Application Design w/ Meta Quantum Circuits with Constraints

Xiaodi Wu
QuICs & UMD

Joint work with Haowei Deng, Yuxiang Peng, and Mike Hicks
Features of NISQ Application Design

**NISQ machines:** very *restricted* hardware resources, where precisely controllable qubits are expensive, error-prone, and scarce.

**NISQ application design:** investigate the best balance of *trade-offs* among a large number of (potentially heterogeneous) factors specific to the targeted application and quantum hardware.

**Multi-Programming (MICRO 2019):**

- ** Competing Goals:**
  1. Fully leverage qubits & Shorten the total execution  
     => Multi-Programming
  2. High Reliability  
     => Sequentially Allocate Programs

**Solution:** A run-time trade-off between these competing goals.
Features of NISQ Application Design

Cross-talk:

Cross-Talk: Red Pairs of gates when executed simultaneously will cause much larger errors.

Competing Goals:
Circuit Depth (decoherence) vs Cross-Talk

Software Solutions:
(1) Circuit Reschedule - Xtalk - (ASPLOS 2020)
(2) Frequency-Aware Compilation (MICRO 2020)

IBMQ Boeblingen

Xtalk

(a) Machine  (b) Program IR  (c) Original Default Schedule  (d) High decoherence schedule  (e) Desired Schedule
Automating NISQ Application Design

Current implementation of NISQ application design are CASE by CASE. A unified and automatic framework for productivity?

Desiderata:

**Succinct Expression**
of different design choices

**Flexible Expression**
of different optimization goals

**Automation of Trade-offs**
of competing optimization goals

**High Reusability & Productivity**
of balancing different trade-offs
Meta Quantum Circuits with Constraints (MQCC)

Desiderata:

**Succinct Expression**
- of different design choices
  - MQCC with choice variables

**Flexible Expression**
- of different optimization goals
  - Flexible Attributes Expression

**Automation of Trade-offs**
- of competing optimization goals
  - Satisfiability Modulo Theories (SMT) Solver

**High Reusability & Productivity**
- of balancing different trade-offs
  - A Meta-Programming Framework
Meta Quantum Circuits with Constraints (MQCC)

```plaintext
\begin{verbatim}
\textbf{Register and variable declarations}
qreg q[10];
creg r[1];
fcho c1 = \{0, 1\};
fcho c2 = \{0, 1\};
\textbackslash{lcho} c = 1 - c1 \ast c2;

\textbf{Module define}
module Bell1(q1, q2){
    h(q1);
cnot(q1, q2);
}

module Bell2(q1, q2){
    \textbf{case} (r[0]){\textbackslash{verbatim}
    \textbf{1}: x(q1);
    \textbf{0}: \textbf{pass}\textbackslash{verbatim}
    \}
    h(q1);
cnot(q1, q2);
}

\textbf{Main part of the program}
choice (c1){
    \textbf{0}: Bell1(q[1], q[2]);
    \textbf{1}: Bell1(q[7], q[8]);
};

h(q[0]);
measure(q[0], r[0]);
choice (c2){
    \textbf{0}: Bell2(q[1], q[2]);
    \textbf{default}: Bell2(q[7], q[8]);
};
\end{verbatim}
```

A Sample Code of MQCC which shares many features with OpenQASM
Meta Quantum Circuits with Constraints (MQCC)

\[\text{Register and variable declarations}\]
\begin{verbatim}
qreg q[10];
creg r[1];
fcho c1 = [0, 1];
fcho c2 = [0, 1];
\end{verbatim}

\[\text{Module define}\]
\begin{verbatim}
module Bell1(q1, q2) {
    h(q1);
cnot(q1, q2);
}
module Bell2(q1, q2) {
    case (r[0]) {
        1: x(q1);
        0: pass
    };
h(q1);
cnot(q1, q2);
}
\end{verbatim}

\[\text{Main part of the program}\]
\begin{verbatim}
choice (c1) {
    0: Bell1(q[1], q[2]);
    1: Bell1(q[7], q[8]);
};
choice (c2) {
    0: Bell2(q[1], q[2]);
default: Bell2(q[7], q[8]);
};
\end{verbatim}

A Sample Code of MQCC which shares many features with OpenQASM

**Define CHOICE variables**

Free Choice (fcho) \(c_1, c_2 \in \mathbb{Z}\), in certain ranges

Limited Choice (lcho) \(c_1 = 1 - c_1 \cdot c_2 \in \mathbb{Z}\)

**Stitch Many Programs w/ choice variables**

 choice (c,v) \(\{ i : P_i \}\)

\(n \in \mathbb{N}\) \(i \in \mathbb{Z}\) \(r \in \mathbb{R}\) \(\text{var} \in \text{Vars}\)
\(\text{qreg} \in \text{Quantum reg.}\) \(\text{creg} \in \text{Classical reg.}\)
\(\text{reg} ::= \text{qreg} | \text{creg}\)

\(P \in \text{Program} ::= \overrightarrow{D} S\)

\(D \in \text{Declaration} ::= \text{RegDecl} | \text{VarDecl}\)
\(\text{RegDecl} ::= \text{qreg} \text{ qreg} | \text{creg} \text{ creg}\)
\(\text{VarDecl} ::= \text{Free} | \text{Limit}\)
\(\text{Free} ::= \text{fcho} \text{ var} = \{ \overrightarrow{r} \}; | \text{fcho} \text{ var} = [i_1, i_2];\)
\(\text{Limit} ::= \text{lcho} \text{ var} = E;\)
\(E \in \text{VarExp} ::= i | \text{var} | E + E | E - E | E \cdot E | E/E | (E)\)
\(S \in \text{Stmt} ::= \epsilon | O | \text{case} | \text{choice} | S; S\)

\(O \in \text{Operation} ::= x(\overrightarrow{r}, \overrightarrow{r});\)
\(\text{case} ::= \text{case}(\text{creg})\{ i : S_i \}\)
\(\text{choice} ::= \text{choice}(\text{var})\{ i : S_i \}\)
A Sample Code of MQCC which shares many features with OpenQASM

```cpp
// Register and variable declarations
qreg q[10];
creg r[1];
fcho c1 = {0, 1};
fcho c2 = [0, 1];
\|lcho c = 1 - c1 * c2;

// Module define
module Bell1(q1, q2){
    h(q1);
cnot(q1, q2);
}
module Bell2(q1, q2){
    case (r[0]){
        1: x(q1);
        0: pass
    }
    h(q1);
cnot(q1, q2);
}

// Main part of the program
choice (c1){
    0: Bell1(q[1], q[2]);
    1: Bell1(q[7], q[8]);
}
choice (c2){
    0: Bell2(q[1], q[2]);
    default: Bell2(q[7], q[8]);
}
```

Depth: \( 7\delta_1^0 \delta_2^0 + 5\delta_1^0 \delta_2^1 + 5\delta_1^1 \delta_2^0 + 7\delta_1^1 \delta_2^1 \)

Noise: \( 0.045\delta_1^0 + 0.066\delta_1^1 + 0.027\delta_2^0 + 0.043\delta_2^1 \)

where \( \delta_c^i = 1 \iff c = i; \text{ otherwise } 0 \)
Expressing the Constraints on Costs/Attributes

Express desired goals as objects called Attributes. Thus, any MQCC program is a transformer on attributes.

Precisely, any attribute $A$ is defined by a tuple $(T, \text{empty}, \text{op}, \text{case}, \text{value})$ s.t.:

- $T$ is a data type of the states. A state of type $T$ consists of information needed in the computation of the cost.
- $\text{empty} : T$ is the initial state at the beginning of the program.
- $\text{op} : T \times \text{string} \times \mathbb{R} \times \text{reg} \rightarrow T$ receives a state, an operation’s name and its arguments, and generates a new state that merges the old state and the information of the operation.
- $\text{case} : T \times T \rightarrow T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs’ information on the old state, and generates a new state merging the old state and the sub-programs’ states.
- $\text{value} : T \rightarrow \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

Program $S'$ attribute semantics $\left[ [S] \right] : (\text{Vars} \rightarrow \mathbb{Z}) \times T \rightarrow T$

$S = \text{opID}(\text{exp}, \text{reg})$
$[S](\sigma, s) = \text{op}(s, \text{opID}, \text{exp}, \text{reg})$
$[S_1 ; S_2](\sigma, s) = [S_2](\sigma, [S_1](\sigma, s))$

$S = \text{case}(\text{creg})\{i : S_i\}$
$[S](\sigma, s) = \text{case}(s', [S_i](\sigma, s')_i)$

$S = \text{choice}(\text{var})\{i : S_i\}$
$k = \sigma[\text{var}]$
$[S](\sigma, s) = [S_k](\sigma, s)$

how transformers evolve over programs

Express the constraints on the final $T$ as SMT instances (some optimization applied but details omitted)
Expressing the Constraints on Costs/Attributes

**Simple Examples of Attributes**

**Attribute** Noise:

T: 
empty():=  
value(s : T):=  
op (s : T, OpID : str, exps : \mathbb{R}, regs : \mathbb{Reg}):=  

s.noise := init s : T, s.noise = 0 
return s 
return s.noise 
s.noise += calNoise(OpId, exps, regs) 
return s 
s.noise = max \{n.noise| n \in group\} 
return s

**Attribute** Depth:

T: 
empty():=  
value(s : T):=  
op (s : T, OpID : str, exps : \mathbb{R}, regs : \mathbb{Reg}):=  

s.dep := Map of Reg \rightarrow \mathbb{N} 
init s : T, s.dep = \emptyset 
return s 
return (max s.dep.values) 
s деп = s.dep.keys \cap regs 
next = \max \{s.dep[i]| i \in share\} + 1 
for i \in regs: s.dep.update(i, next) 
return s 
s.dep = \{(k, \max \{n.dep[k] | n \in group\}) | k \in all\} 
return s

- **E. Examples of Attributes**
  - **Noise:**
    - T: noise : \mathbb{R}
    - empty():= init s : T, s.noise = 0
    - return s
    - value(s : T):= return s.noise
    - op (s : T, OpID : str, exps : \mathbb{R}, regs : \mathbb{Reg}):=
      s.noise += calNoise(OpId, exps, regs)
      return s
    - case (s : T, group : Vector of T):=
      s.noise = max \{n.noise| n \in group\}
      return s

  - **Depth:**
    - T: dep : Map of Reg \rightarrow \mathbb{N}
    - empty():= init s : T, s.dep = \emptyset
    - return s
    - value(s : T):= return (max s.dep.values)
    - op (s : T, OpID : str, exps : \mathbb{R}, regs : \mathbb{Reg}):=
      share = s.dep.keys \cap regs
      next = \max \{s.dep[i]| i \in share\} + 1
      for i \in regs: s.dep.update(i, next)
      return s
    - case (s : T, group : Vector of T):=
      all = \bigcup_{n \in group} n.dep.keys
      s.dep = \{(k, \max \{n.dep[k] | n \in group\}) | k \in all\}
      return s
We advocate multi-programming NISQ computers to improve the allocations of a 4-qubit program P1 and a 5-qubit program P2 on its capability to optimize for greater reliability. In order to understand the reliability of a program directly impacts its reliability; these error rates vary in time. Thus, the physical qubits allocated to the reliability of a NISQ application depends on the physical qubits mapped and executed reliably, they are compiled together using X and Y qubits each.

No trial in which all applications in the group give the correct ways: (1) isolated (single-programmed) as the baseline; and advantages and our evaluation in this section.

A. Multi-Program Quantum Computers

We use the same applications (Table I) and methods updates the depth of register used by the new application.

Compile refers to the corresponding value of map.keys

The depth of a program is the maximum depth of all branches. The depth of each register is its maximum value of map.keys

Example, the depth in all branches.

The rate of success - the probability of a successful trial

The framework provided by Bernstein-Vazirani [3]

Fig. 7: Layout of IBMQ Boeblingen.

Execute 8192 trials on IBMQ Rochester for each group. All experiments performed on IBMQ machines

Group A-B contains two applications A and B. Similarly A-B-C.

A success trial if all applications in the group are successful.

Execute 8192 trials on IBMQ Rochester for each group.

Comparative results to MICRO 2019 for this part.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Qubits</th>
<th># of gates</th>
<th># of CNOTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>bv3</td>
<td>Bernstein-Vazirani</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>bv4</td>
<td>Bernstein-Vazirani</td>
<td>4</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Toff3</td>
<td>Toffoli gate</td>
<td>3</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Fred3</td>
<td>Fredkin gate</td>
<td>3</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>Pere3</td>
<td>Peres gate</td>
<td>3</td>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>

Recover some essential ideas of MICRO 2019 while ignoring others.

```
qreg q[10];
creg r[1];

module Bell1(q1,q2){
    h(q1);
    cnot(q1, q2);
}

module Bell2(q1, q2){
    case (r[0]){           
      1: barrier(q1);
      0: x(q1);               
    }
    h(q1);
    cnot(q1,q2);
}

fcho c1 = {0, 1};
fcho c2 = {0, 1};

choice (c1){
  0: Bell1(q[1], q[2]);
}

choice (c2){
  0: Bell2(q[1], q[2]);
  1: Bell2(q[7], q[8]);
}
Case Study

Cross-talk: (Xtalk - ASPLOS 2020)

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

use “barrier” to control the order of the gates

Benchmark Test:

CNOT 15 8 = SWAP 15 16; SWAP 16 11; SWAP 8 7; SWAP 7 12; CNOT 11 12

Execute 8192 trials on IBMQ Boeblingen for SWAP circuits connecting a-b

- Seq: running all instructions serially
- Par: maximize the parallel execution, default in Qiskit

Benchmark over SWAP circuits connecting a-b on IBM Boeblingen
Case Study: Multi-Programming + Cross-Talk

Optimizing Goal: Noise + Decoherence + Crosstalk

Easy implementation in MQCC

- seq: Sequential, always high-quality qubits, but larger depth (decoherence)
- multi-p: Multi-programs without considering crosstalk, short depth, but large crosstalk errors
- multi-c: Multi-programs with crosstalk, short depth and large successful probability

More Optimization w/ MQCC

(a) Probability of Successful Trial. Here higher PST is better.

(b) Circuit Depth. Here lower circuit depth is better.
Case Study: Cost-Effective Uncomputation

Recovering one idea from SQUARE (ISCA 2020)

Strategic Quantum Ancilla Reuse for Modular Quantum Programs

Deciding the point to uncompute for ancilla reuse

```module foo(c,...){
...
 choice (c) {
  0: pass \No uncomputation
  1: uncomputation code; \Do uncomputation
    release(ancillas);
  }
}
```

```fcho c1,c2 = {0,1};
lcho ct = 1 - c1*ct;
foo1(c1,....);
foo2(c2,...);
choice (ct) {
  0: pass \No uncomputation
  1: uncompute code \Do uncomputation
}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Gate Number</th>
<th>Qubits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2of5</td>
<td>Output is 1 if number of 1s in its input equals two.</td>
<td>1528</td>
<td>8</td>
</tr>
<tr>
<td>6sym</td>
<td>Function with 6 inputs and 1 output.</td>
<td>1620</td>
<td>11</td>
</tr>
<tr>
<td>rd53</td>
<td>Input weight function with 5 inputs and 3 outputs.</td>
<td>1849</td>
<td>10</td>
</tr>
<tr>
<td>adder4</td>
<td>4-bit in-place controlled-addition.</td>
<td>1748</td>
<td>12</td>
</tr>
<tr>
<td>elsa</td>
<td>Heavy workload and shallowly nested synthetic function.</td>
<td>256</td>
<td>14</td>
</tr>
<tr>
<td>jasmine</td>
<td>Shallowly nested synthetic function.</td>
<td>604</td>
<td>11</td>
</tr>
<tr>
<td>belle</td>
<td>Light workload and deeply nested synthetic function.</td>
<td>768</td>
<td>9</td>
</tr>
</tbody>
</table>

(a) AQV of the benchmarks. (Lower AQV is better.)

(b) Realistic noise simulation using IBM Qiskit Aer simulator. (Lower total variation distance is better.)
Thank You!

MQCC:
- github/sqrta/MQCC