Compilation of Quantum Programs with Control Flow Primitives in Superposition

Master Thesis

05.02.25

Sascha Thiemann

Supervisors: apl. Prof. Dr. Thomas Noll

Prof. Dr. rer. nat. Dominique Unruh

Outline

Introduction

- Idea of quantum control flow first used by [Altenkirch and Grattage, 2005].
- Later, it was defined by [Ying et al., 2012].
- Most languages focus on classical control flow.
- Only recently, Quantum Control Machine, proposed by [Yuan et al., 2024].
  - An instruction set architecture.
  - Allows for quantum control flow with an assembly-like syntax.
  - Low-level design of the architecture and syntax $\rightarrow$ complex programs.

$\Rightarrow$ Build upon the ideas, define a high-level quantum language with quantum control flow.

$\Rightarrow$ Plan to implement a compiler for it.

Quantum Control Flow

- First used by [Altenkirch and Grattage, 2005] to define functional programming language.
- For example, used to define Hadamard gate as function *had*:

$$\begin{aligned} had &: Q \to Q \\ had &: x \mapsto \ \texttt{if}^{\circ} x \\ & \qquad\quad \texttt{then} \ \{false \mid -true\} \\ & \qquad\quad \texttt{else} \ \ \{false \mid true\} \end{aligned}$$

- Later, formally defined by [Ying et al., 2012].
- Allows for the execution of functions based on values in superposition.
- Results in superposition of the results of individual executions.

Background

Limitations

- Mainly limited by two principles: *reversibility* and *synchronization*.
- Instructions required to be reversible, as they are unitary transformations.
  - $\Rightarrow$ Control flow, as implemented in classical computers, not possible.
  - – Landauer Embedding [Landauer, 1961] seems to offer solution.
  - – Can make non-reversible function reversible by outputting input value.
  - – However, output includes program history and result depends on the history $\rightarrow$ become entangled.
  - – Leads to disruptive entanglement, invalid results [Yuan et al., 2024].
- Program counter may become entangled with data.
  - $\Rightarrow$ Disruptive entanglement.
  - – The principle of synchronization: *Control flow must become independent from data*.
  - – Program may include padding operations to ensure synchronization.
  - $\Rightarrow$ Loops cannot depend solely on value in superposition.
  - $\Rightarrow$ Instead, a loop must be bounded by classical value [Yuan et al., 2024].

Quantum Control Machine

- Quantum Control Machine (QCM), proposed by [Yuan et al., 2024]
- Focused on quantum control flow.
- Syntax and logic similar to classical assembly language.
- Employs a branch control register *bcr* to enable reversible jumps.
- Instead of increasing IP by 1 after statement, increased by value of *bcr*.
- *bcr* can then be reversibly modified.
  - To jump by 5, the *bcr* increased by 5
  - At its destination, decreased by 5 again.

Instructions

- Here, some instructions of the QCM are listed.
- Reversed instruction for every instruction, e.g., `radd` is the subtraction operation.

Operation	Syntax	Semantics
No-op	`nop`	Only increases instruction pointer by the *bcr*.
Addition	`add` *ra rb*	Adds register *rb* to *ra*.
Multiplication	`mul` *ra rb*	Multiplies register *ra* by *rb*.
Jump	`jmp` *p*	Increases *bcr* by *p*.
Conditional Jumps	`jz` *p ra*	Increases *bcr* by *p* if *ra* is 0.
	`jne` *p ra rb*	Increases *bcr* by *p* if *ra* is not equal to *rb*.

An excerpt of the QCM instruction set with instructions used in later examples.

Background

(Non-) Reversible Example

- Example of a classical program and reversible equivalent.
- Both programs calculate x^y.
- Neither are synchronized because loop is not bound by classical value.

```
      add   res $1
      add   r1  y
l1:   jz    l2  r1
      mul   res x
      radd  r1  $1
      jmp   l1
l2:   nop
```

A non-reversible exponentiation algorithm.

```
      add   res $1
      add   r1  y
l1:   rjne  rl1 r1  y
rl2:  jz    l2  r1
      mul   res x
      radd  r1  $1
rl1:  jmp   l1
l2:   rjmp  l2
```

A reversible exponentiation algorithm.

Background

Reversible Synchronized Example

- Synchronized implementation, calculating $x^{\min(y, max)}$.
- *max* is a classical bound for loop.
- Line 9 includes padding instruction.

```
1            add   res  $1
2            add   r1   max
3  l1:       rjne  rl1  r1   max
4  rl2:      jz    l2   r1
5  rl3:      jg    l3   r1   y
6            mul   res  x
7  rl4:      jmp   l4
8  l3:       rjmp  rl3
9            nop                  ; padding
10 l4:       rjle  rl4  r1   y
11           radd  r1   $1
12 rl1:      jmp   l1
13 l2:       rjmp  rl2
```

A synchronized, reversible exponentiation algorithm.

Language Overview

- Idea for language: Provide high-level language with the capabilities of the QCM.
- Want to remove low-level concepts, add high-level ones.
- Jump instructions in superposition are removed $\rightarrow$ need to add other control flow primitives.
- Introduced concepts:
  - blocks and scopes,
  - different data types,
  - composite gates,
  - loop statements, unrolled at compile time, and
  - quantum if- and else-statements.
- Implicit measurements
- Translated to OpenQASM

Language

Syntax

- Define $\mathit{CFG}_{\mathit{Luie}}$ for our language.
- Start symbol is the program, consisting of gate declarations and a block.
- A block is a list of translatables, either statements or declarations.

$$
\begin{aligned}
\mathit{CFG}_{\mathit{Luie}} &= (V_{\mathit{Luie}}, \Sigma_{\mathit{Luie}}, R_{\mathit{Luie}}, \mathit{prg}_{\mathit{Luie}}) \\
V_{\mathit{Luie}} &= \{\mathit{exp}, \mathit{rExp}, \mathit{gate}, \mathit{prg}_{\mathit{Luie}}, \ldots\} \\
\Sigma_{\mathit{Luie}} &= \{\texttt{..}, \texttt{range}, \texttt{(}, \texttt{)}, \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Program} : \mathit{prg}_{\mathit{Luie}} \quad &::= \mathit{gDcl}_1 \ldots \mathit{gDcl}_n \; \mathit{blk} \mid \\
& \mathit{blk} \\
\mathit{Block} : \mathit{blk} \quad &::= t_1 \ldots t_n \mid \\
& \epsilon
\end{aligned}
$$

```
/* Gate Declaration */
gate c_h_reg(control, reg) do
   /* Block Start */

   qif control do
     for i in range(sizeof(reg)) do
       h reg[i];
     end
   end

   /* Block End */
end

/* Block Start */
const regSize : int = 3; // Decl.
qubit c;                 // Decl.
qubit[regSize] a;        // Decl.
c_h_reg c, a;            // Stm.
/* Block End */
```

Luie program with structure highlighted.

Language

Syntax

- Three different statements: quantum if-statement, a loop statement, gate application.
- To combine qubit or register access: qubit argument.
- Expressions for numeric values or ranges.
- Fix set of defined gates $\longrightarrow$ differentiate translations.

$$
\begin{aligned}
\textit{Statement} : \textit{stm} ::= {} & \texttt{qif}\ \ \textit{qArg}\ \ \texttt{do}\ \ \textit{blk}\ \ \texttt{end} \mid \\
& \texttt{for}\ \ \textit{id}\ \ \texttt{in}\ \ \textit{rExp}\ \ \texttt{do}\ \ \textit{blk}\ \ \texttt{end} \mid \\
& \textit{id}\ \textit{qArg}_1, \ldots, \textit{qArg}_n\texttt{;}
\end{aligned}
$$

$$
\textit{QubitArg} : \textit{qArg} ::= \textit{id} \mid \textit{id}[\textit{exp}]
$$

```
gate c_h_reg(control, reg) do
   /* If-Statement */
   qif control do
      /* Loop Stat. | ⬐ Expression */
      for i in range(sizeof(reg)) do
             /* ⬑ Range Exp. */
         h reg[i];
         /* ⬑ Qubit Argument */
      end
   end
end

const regSize : int = 3;
qubit c;
qubit[regSize] a;
/* Composite Gate Application */
c_h_reg c, a;
      /* ⬑  ⬑ Qubit Argument */
```

Luie program with statements and arguments highlighted.

Translation Functions

- Translation functions:
  - *trans*: translates program.
  - *bt*: translates code blocks.
  - *tt*: translates either declaration of statement.
- Symbol table saves symbol information for translation.
  - Information updated by *up* function.
  - Initial table st_ϵ contains no mapping.

$$\begin{aligned}
\mathit{trans}(\texttt{gate c_h_reg...}) &= \texttt{OPENQASM 3.0;} \\
&\ \texttt{include "stdgates.inc";} \\
&\ \mathit{bt}(\texttt{const regSize = 3;}\ldots, \mathit{up}(st_\epsilon, \texttt{gate c_h_reg...})) \\
st_1 = \mathit{up}(st_\epsilon, \texttt{gate c_h_reg...}) &= [\texttt{c_h_reg} \mapsto (\texttt{gate}, \texttt{qif control do ... end}, \\
&\qquad\qquad \texttt{control}, \texttt{reg})]
\end{aligned}$$

$$\begin{aligned}
\mathit{bt}(\texttt{const regSize = 3;}\ldots, st_1) = {} & tr_1 && \text{where } (tr_1, st_2) = \mathit{tt}(\texttt{const regSize = 3;}, st_1) \\
& tr_2 && \text{where } (tr_2, st_3) = \mathit{tt}(\texttt{qubit c;}, st_2) \\
& tr_3 && \text{where } (tr_3, st_4) = \mathit{tt}(\texttt{qubit[regSize] a;}, st_3) \\
& tr_4 && \text{where } (tr_4, _) = \mathit{tt}(\texttt{c_h_reg c, a;}, st_4)
\end{aligned}$$

Implementation

Overview

- Four compilation stages:
  1. the lexical and syntactic analysis,
  2. semantic analysis,
  3. code generation, and
  4. optimizations
- Process managed by a static compiler class.
  - Parses command line parameters.
  - Handles input and output of files.
  - Calls the different stages.
  - Handles logging and error messages.

Lexical and Syntactic Analysis

- First compilation stage is lexical and syntactic analysis.
- Both lexer and parser created with the ANTLR4 tool.
- Generates the source code based on a given grammar.
- Implementation of the grammar is more elaborate version of previously disused one.

```
parse      : mainblock EOF;

mainblock : gateDeclaration* (declaration | statement)*;

block      : (declaration | statement)*;
```

The basic structure of parsing rules for Luie.

Semantic Analysis

- Analyzes non-syntactic constraints of program, mainly:
  1. declaration analysis and
  2. type checking.
- Declaration analysis ensures all identifiers used were previously declared and all identifiers used in declarations are not already declared.
- Type checking ensures that symbols are used in the correct context.

```
qubit c;            /* ↰ Type error (cannot use qubits in arith. expression) */
const regSize : int = c + 2;
            /* ↰ Already declared */
qubit[regSize] c;
              /* ↰ Type error (expects a register or qubit) */
ccx c, a, regSize;
        /* ⬑ Undeclared identifier */
```

Luie program with semantic errors highlighted.

Implementation

Code Generation

- Parse tree is traversed and source code is translated to in-memory representation.
- Source code representation (SCR) is translated to target code representation (TCR).
- TCR can be translated directly to the textual OpenQASM code.
- Example code generation with program:

```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
        end
    end
end

qubit c;
qubit[3] a;
c_h_reg c, a;
```

An example Luie program to show the code generation process.

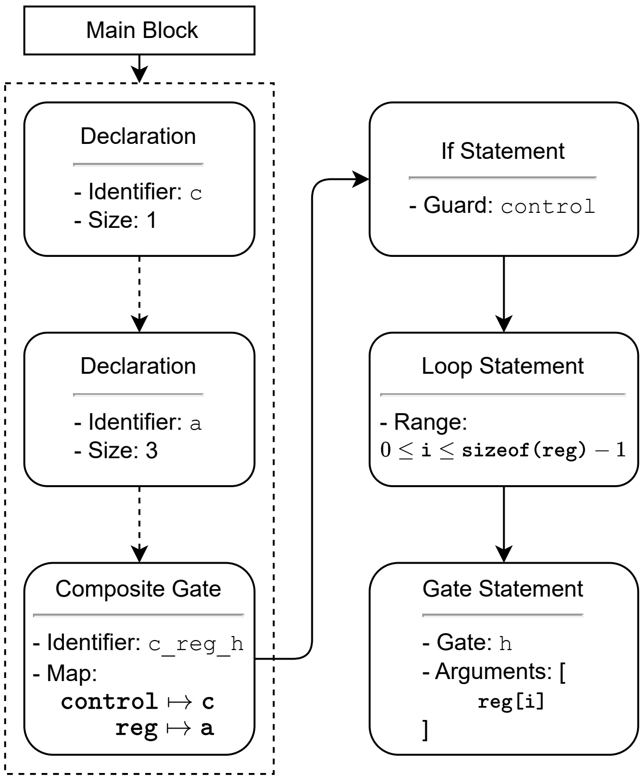
Source Code Representation

- Three main classes:
  1. Code block: containing list of translatables.
  2. Declaration: only register declaration.
  3. Statements: variety of statements.
- Example contains three translatables.
  - First two declarations, last is gate statement.
  - Gate's body ⊃ if-statement ⊃ loop statement ⊃ gate application.

```
gate c_h_reg(control, reg) do
    qif control do
        for i in range(sizeof(reg)) do
            h reg[i];
end end end

qubit c; qubit[3] a;
c_h_reg c, a;
```

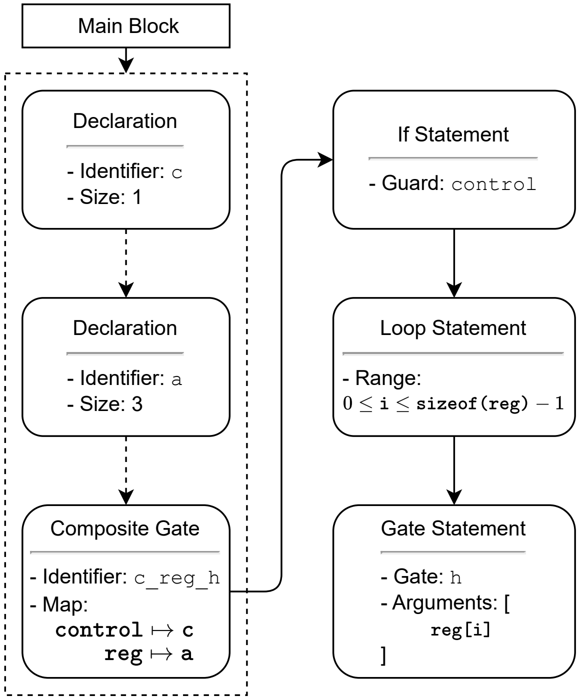
An example Luie program to show the code generation process.



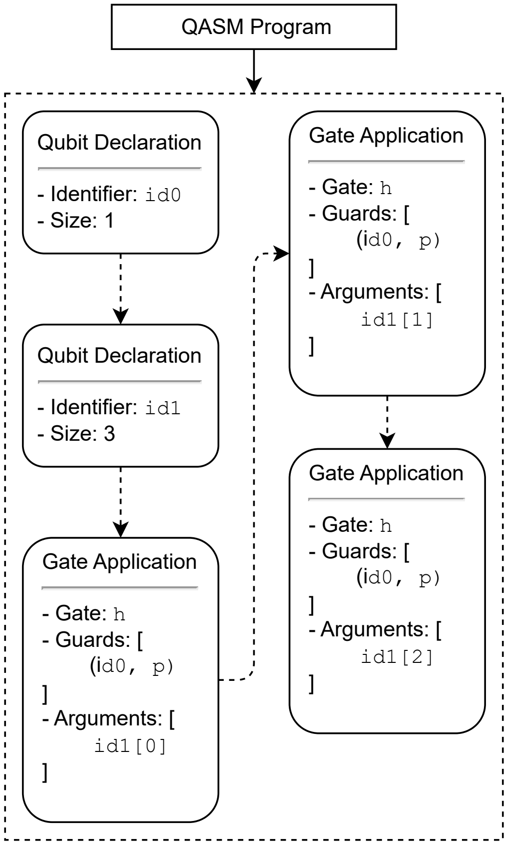
The SCR of the example program.

Target Code Representation

- TCR based on `QASMProgram` object.
- Contains list of `Code` objects, either gate or declaration.
- All SCR objects translated to list of code objects and appended to program object.
- Example:
  - Change identifiers to ensure uniqueness.
  - Loop is unrolled.
  - If-statement adds control to gate applications.
  - Composite gate is inlined.



The SCR of the example program.



The TCR of the example program.

Implementation

Translated Example Program

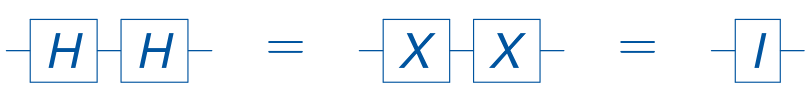
- TCR converted to OpenQASM program.
- Version string and include header prepended to the code.
- Add measurements.
- Additions performed after optimization.

```
OPENQASM 3.0;
include "stdgates.inc";
qubit id0;
qubit[3] id1;
ctrl(1) @ h id0, id1[0];
ctrl(1) @ h id0, id1[1];
ctrl(1) @ h id0, id1[2];
bit id0_measurement;
measure id0 -> id0_measurement;
bit[3] id1_measurement;
measure id1 -> id1_measurement;
```

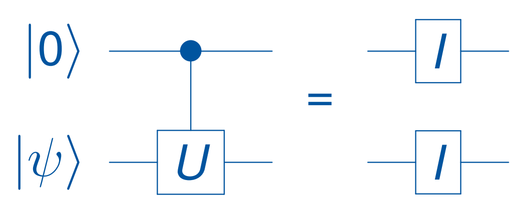
The OpenQASM translation of the example Luie program.

Optimization

- Compiler performs peephole optimizations based on rules presented by [Garcia-Escartin and Chamorro-Posada, 2011].
- Can be divided into four rules:
  1. Null gate rule.
  2. Peeping control rule.
  3. Hadamard reduction rule (omitted).
  4. Control reversal (omitted).
- Null gates are combinations of gates under specific conditions equivalent to I.
  - Simplest null gate version is twofold application of self-inverse gate.
- Our peeping control rules are special case of null gates.
  - Control is $|1\rangle \longrightarrow$ remove control, control is $|0\rangle \longrightarrow$ remove gate.



Null gates of self-inverse gates.



Null gates for gates in specific conditions.

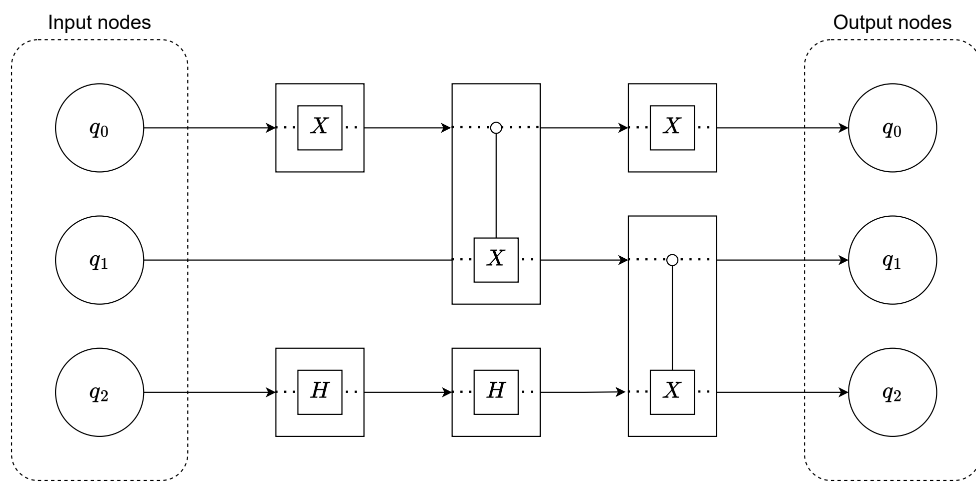
Implementation

Cricuit Graph

- Rules not directly applied to program, but to circuit graph.
- Circuit graph based on graph proposed by [Kreppel et al., 2023].
  - Acyclic and directed.
  - Nodes split into input, output, gate nodes.
  - Input, output nodes have only one outgoing, incoming edge.
  - Each input, output, edge belongs to qubit in program.
  - Qubit wire represented by path from input to output node over corresponding edges.

```
qubit[3] q;
x q[0];
cx q[0], q[1];
x q[0];
h q[2];
h q[2];
cx q[1], q[2];
```

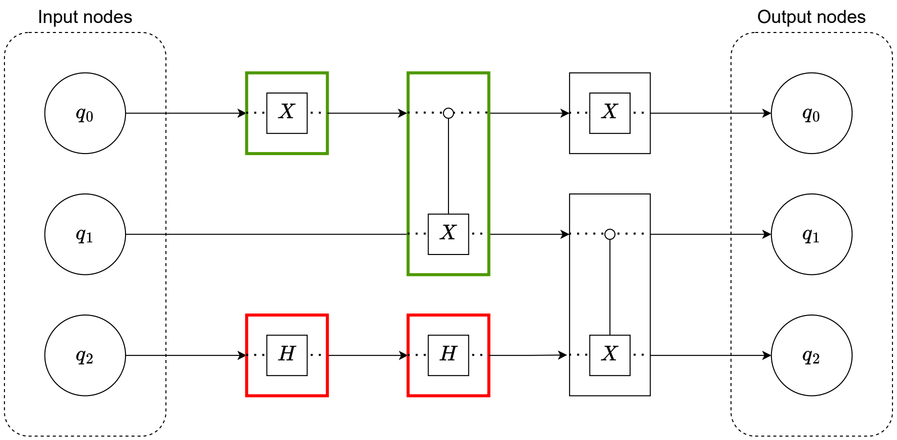
A simple, unoptimized program.



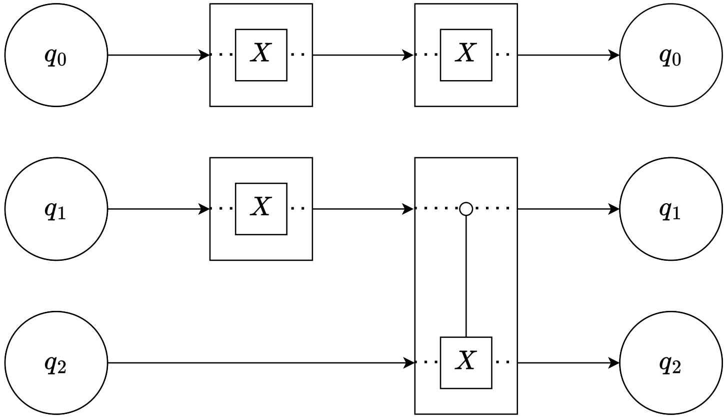
An example of a simple, unoptimized circuit graph.

Example Optimization Process I

- To optimize the graph, each qubit wire is iterated.
- All subpaths (up to max. length) checked for alternatives.
- Example:
  - First wire: Peeping control rule can be applied to first *CX* gate.
  - Third wire: *HH* null gate can be removed.



An example of a simple, unoptimized circuit graph.



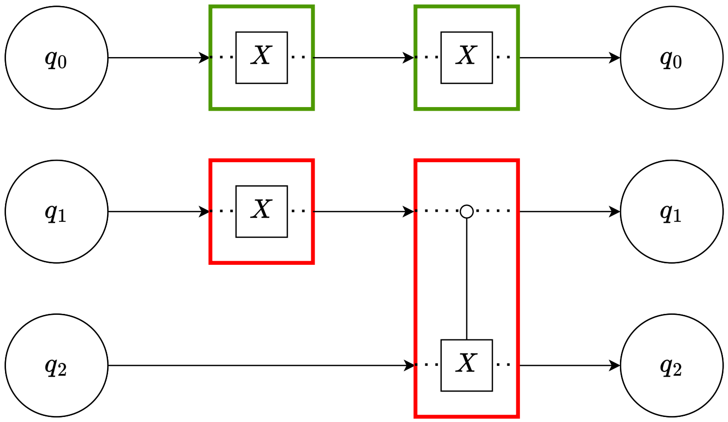
The circuit graph after the first optimization step.

Example Optimization Process II

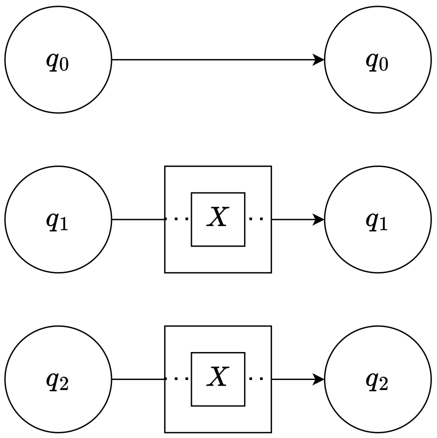
- While all qubit wires iterated, still possible optimizations.
- Applying optimizations may enable others!

⇒ Optimization repeated as long as previous iteration applied optimizations.

- Example:
  - First wire: *XX* null gate can be removed.
  - Second wire: Peeping control rule can be applied to *CX* gate.



The circuit graph after the first optimization step.



The completely optimized graph.

Evaluation

- Evaluation consists of two aspects:
  1. Optimizations performed.
  2. Execution time stages.
- Example program: quantum ripple-carry adder proposed by [Cuccaro et al., 2004].
  - Takes two registers *a* and *b*, two qubits *cin* and *cout*.
  - Adder adds *a* register to *b* register.
  - *cin* and *cout* qubits are used as input, output carry bits.
- Our implementation consists only of *CX* and *CCX*.

Optimization Evaluation

- For optimization evaluation, classical inputs and in superposition.
- First inputs: $a = |1\rangle$ and $b = |15\rangle$.
  - Classical inputs $\longrightarrow$ peeping control rules, null gate rules can be applied.
  - Optimized circuit: only gates that initialize the result.
  - Only two X gates remain.
  - First X flips $a[0]$ qubit $\longrightarrow a = |1\rangle$, second flips $cout$, indicating result of $|16\rangle$.
- Second inputs: $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$ and $b = |4\rangle$.
  - Inputs in superposition $\longrightarrow$ peeping control rules, null gate rules only applied *partially*.
  - Only $\frac{12}{25}$ gates can be optimized.
  - Other inputs in superposition: even fewer gates optimized.

Evaluation

Performance Evaluation

- Compiled adder with input of $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$ and $b = |15\rangle$ for different register sizes n.
- Program size does not change $\rightarrow$ execution times of the semantic analysis remain constant.
- Code generation stage: linear increase.
  - Unrolled loop $\rightarrow$ compiled program increases linearly.
- Optimization worst performance, approximate quadratic increase.

Register Size n	Execution Time of Stages in ms		
	Semantic Analysis	Code Generation	Optimization
64	27.3	47.8	711.6
128	26.3	50.4	2292.4
256	26.2	59.7	10755.7
512	25.8	74.9	60204.7
1024	26.1	109.1	405376.6

The execution times compiling a quantum ripple-carry adder with different register sizes.

Conclusion

- Multiple aspects that can be improved.
- Optimizations:
  - Focused on high-level optimizations $\rightarrow$ work best in tandem with other tools.
  - Addition of hardware-focused features and transpilation to concrete devices.
  - Improve performance of optimization stage by parallelization of wire traversals.
  - Increases complexity of program, e.g., race conditions need to be considered.
- General improvements:
  - Addition of type casting for named constants.
  - Expansion of predefined function; addition of constants, e.g., π or e.
  - Addition of explicit measurement.

References

Altenkirch, T. and Grattage, J. (2005).
A functional quantum programming language.
In 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05), pages 249–258. IEEE.

Cuccaro, S. A., Draper, T. G., Kutin, S. A., and Moulton, D. P. (2004).
A new quantum ripple-carry addition circuit.

Garcia-Escartin, J. C. and Chamorro-Posada, P. (2011).
Equivalent quantum circuits.

Kreppel, F., Melzer, C., Olvera Millán, D., Wagner, J., Hilder, J., Poschinger, U., Schmidt-Kaler, F., and Brinkmann, A. (2023).
Quantum circuit compiler for a shuttling-based trapped-ion quantum computer.
Quantum, 7:1176.

Landauer, R. (1961).

References

Irreversibility and heat generation in the computing process.
IBM Journal of Research and Development, 5(3):183–191.

Ying, M., Yu, N., and Feng, Y. (2012).
Defining quantum control flow.

Yuan, C., Villanyi, A., and Carbin, M. (2024).
Quantum control machine: The limits of control flow in quantum programming.
Proceedings of the ACM on Programming Languages, 8(OOPSLA1):1–28.