CTQr: Control and Timing-Aware Qubit Routing

Ching-Yao Huang
National Tsing Hua University
Hsinchu, Taiwan
rabit66119929@gapp.nthu.edu.tw

Wai-Kei Mak
National Tsing Hua University
Hsinchu, Taiwan
wkmak@cs.nthu.edu.tw

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473795

***Abstract*—To execute a quantum program, it has to be compiled for execution on the target quantum processor. The program is first converted into a logical circuit composed of elementary gates supported by the target processor. Most often the logical circuit cannot be executed directly on the quantum processor due to the limited connectivity between the physical qubits of the processor. So, a quantum compiler needs to perform qubit routing by inserting auxiliary gates to execute operations like SWAP, MOVE, and BRIDGE in order to satisfy the connectivity constraint. Qubit routing yields a physical circuit that can be executed on the target processor. Finally, the physical circuit still has to be scheduled considering the gate delays and the control constraints imposed by the shared classical control electronics of the quantum processor. For noisy intermediate-scale quantum processors, it is important to minimize the latency of the final scheduled physical circuit. However, solving qubit routing without considering gate delays and control constraints will inevitably lead to suboptimal final results. Here we propose a control and timing-aware qubit routing algorithm, CTQr, considering gate delays and control constraints. Moreover, CTQr performs gate merging on the fly in order to minimize the final circuit latency. The experimental results show that CTQr outperforms the state-of-art approach with 11.2%, 8.8%, and 54.6% average reduction in the circuit latency, number of additional gates, and execution time, respectively.**

I. Introduction

Several quantum systems have been released[7][1][19]. Besides, researchers can execute quantum programs on real quantum devices through the cloud[8]. In the Noisy Intermediate-Scale Quantum era, quantum architectures are not ideal. They only support a limited set of elementary gates with relatively high fidelity and it is not possible to establish direct interaction between any pairs of qubits due to limited connectivity between the physical qubits. Furthermore, they use classical control electronics with channels that are shared among several qubits [19][3][5] since using a dedicated instrument for each qubit is highly resource-intensive and is not scalable.

A quantum program is designed under the ideal assumption that it will execute on an ideal quantum architecture where gates can be applied to any pairs of qubits without any control limitation and qubits have unlimited coherence time and zero error rate. In practice, to execute a quantum program, it is necessary to compile it for execution on the target quantum processor. First, a quantum program composed of high-level operations is decomposed into a logical circuit composed of elementary gates supported by the target architecture. Second, logical qubits are mapped to physical qubits and auxiliary operations such as SWAP, MOVE, and BRIDGE are inserted into the logical circuit to transform it to a physical circuit subject to the connectivity constraint of the target architecture so that it can be executed on the target quantum processor. Finally, the physical circuit has to be scheduled considering the gate delays and the control constraints imposed by the shared classical control electronics of the quantum processor. For noisy intermediate-scale quantum processors, since qubits are error-prone and decohere over time, it is important to minimize the latency of the final scheduled physical circuit.

The physical synthesis stage of compilation consists of two steps, initial mapping and qubit routing. Initial mapping, also known as qubit placement, is to decide a mapping from logical qubits to physical qubits in the beginning. Qubit routing is to insert additional gates such as SWAP, MOVE, and BRIDGE operations into the logical circuit to satisfy the connectivity constraint of the quantum architecture. Various qubit routing approaches have been proposed aiming at different objectives like the minimization of additional gate count [6][11], circuit latency [20][18][17][10], or circuit error rate [15].

Most of the previous works except [17][10] perform qubit routing without considering gate delays and control constraints. Though some works [20][18] tried to reduce latency by considering the gate delays, they ignored the control constraints. Performing qubit routing without considering gate delays and control constraints inevitably leads to suboptimal results after scheduling to satisfy the control constraints.

We propose a control and timing-aware qubit routing algorithm, CTQr, to perform qubit routing considering gate delays and control constraints. Moreover, CTQr considers gate optimization by gate merging[14] on the fly which leads to better routing decisions [12]. Our method iteratively inserts useful auxiliary operations into the logical circuit which can be scheduled with other gates in the logical circuit in parallel until all gates are scheduled.

[17] is the state-of-the-art work for solving the control and timing-aware qubit routing problem. Different from [17] which limits the number of auxiliary operations inserted in each iteration to one, we insert multiple auxiliary operations which can be scheduled in parallel to reduce the timing overhead. Unlike [17] which performs gate merging on the whole circuit before and after routing, we do gate merging on the fly when we determine and insert the auxiliary operations during routing.

Our method was evaluated with various benchmarks on the Surface-17[19] processor. Experimental results show that our method significantly outperformed the state-of-the-art work [17] in all metrics. The circuit latency is reduced by 11.2%, the number of additional gates is reduced by 8.8%, and the runtime is reduced by 54.6% on average.

The rest of the paper is organized as follows. The background of quantum computing is introduced in section II and the control and timing-aware qubit routing problem is formulated in section III. Section IV introduces our method. The experimental results are reported in section V and we conclude this article in section VI.

This work is supported in part by the National Science and Technology Council under grant 111-2221-E-007-119-MY3.

979-8-3503-9354-5/24/$31.00 ©2024 IEEE

II. Background

In this section, we introduce some relevant basic concepts.

Qubit A qubit has two basis states, $|0\rangle$ and $|1\rangle$, and it can be in the superposition state represented as a linear combination of two basis states, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.

Quantum Gate A single-qubit gate operates on one target qubit. For instance, the RX$_\theta$ (Fig. 1(a)) and RY$_\theta$(Fig. 1(b)) gates rotate the qubit along the X- and Y- axes for $\theta°$, respectively. A two-qubit gate operates on two qubits simultaneously. For example, a Controlled-Z (CZ) gate (Fig. 1(c)) performs controlled phase shift. In this work, the elementary gates are RX, RY, and CZ gates and the delays of RX, RY, and CZ gates are defined as one, one, and two timesteps, respectively. All the high-level operations[1] can be expressed as a combination of elementary gates and the decomposition for high-level operation like CX is shown in Fig. 1(d).

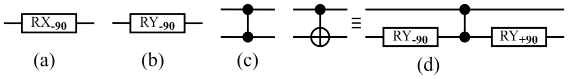

Fig. 1: (a) A RX gate rotates the target qubit along X-Axis for 90°. (b) A RY gate rotates the target qubit along Y-Axis for 90°. (c) A CZ gate. (d) A CX operation and its implementation.

Quantum Circuit A quantum circuit can be represented by a quantum circuit diagram as shown in Fig. 2. Each logical qubit is represented by a horizontal line and the gates are placed from left to right on the lines according to their execution order. Fig. 2 shows a circuit with seven logical qubits and six gates. g_1, g_3, g_4, g_6 are single-qubit gates applied to qubits q_2, q_4, q_6, and q_7, respectively. g_2 and g_5 are two-qubit gates applied to $\{q_3, q_6\}$ and $\{q_3, q_4\}$, respectively.

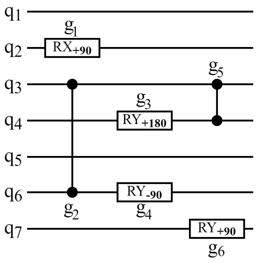

Fig. 2: A quantum circuit.

Quantum Device Real-world quantum device can be described by a connected undirected graph called the architecture graph. Fig. 3 shows the architecture graph for the scalable quantum processors, Surface-17[19]. Each vertex on the graph represents a physical qubit and a two-qubit gate can only be applied to a pair of adjacent physical qubits.

A. Connectivity Constraint and Control Constraints

A single qubit gate can be performed on any physical qubit of a quantum processor. However, a two-qubit gate may only be performed between adjacent physical qubits of a quantum processor which is referred to as the ***connectivity constraint***. Usually, no initial mapping of logical qubits of a given logical circuit to the physical qubits of a quantum processor can satisfy the connectivity constraint for all two-qubit gates. We use auxiliary operations such as SWAP, MOVE, and BRIDGE to resolve the connectivity issue. A SWAP operation (Fig. 4(a)) swaps the mapping of two logical qubits on the processor, and can be implemented with six RY gates and three CZ gates. A MOVE operation (Fig. 4(b)) moves the mapping of a logical qubit from one physical qubit to another physical qubit and requires the state of the destined physical qubit to be $|0\rangle$ before moving, and a MOVE operation can be implemented with four RY gates and two CZ gates. A BRIDGE operation (Fig. 4(c)) emulates a CZ gate on two physical qubits at distance two on the architecture without modifying the mapping.

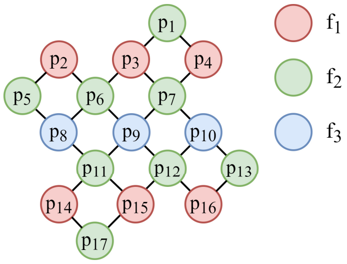

Fig. 3: The architecture of Surface-17 processor. The physical qubits are divided into frequency groups.

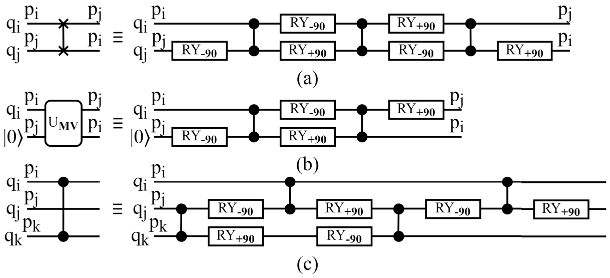

Fig. 4: (a) SWAP operation and its implementation. (b) MOVE operation and its implementation. (c) BRIDGE operation and its implementation.

In scalable quantum processors, shared classical control electronics are used to control multiple qubits and constrain the parallel execution of quantum gates. Here we describe the ***control constraints*** imposed by the shared classical control electronics. Single-qubit gates are realized by using microwave pulses and single-qubit gates applied to red, green, and blue colored physical qubits in Fig. 3 are performed at frequencies f_1, f_2, and f_3, respectively, where $f_1 > f_2 > f_3$. In this work, we assume that one microwave source drives all or some of the physical qubits that have the same frequency at each timestep [19][3]. Therefore, the same type of single-qubit gates can be performed on all or some of the physical qubits that share the same frequency but no more than one type of single-qubit gate can be performed on the physical qubits that share the same frequency at the same time.

An edge between two adjacent physical qubits connect qubits with frequencies f_1 and f_2, or qubits with frequencies f_2 and f_3. For a two-qubit gate, we call the operating physical qubit with higher frequency p_{high} (the frequency of p_{high} is f_{high}) and the operating physical qubit with lower frequency p_{low} (the frequency of p_{low} is f_{low}). A CZ gate is realized by lowering the frequency of p_{high} to an interacting frequency f_{high}^{int} close to f_{low}. Moreover, to prevent any undesired interaction between the operating physical qubit and its adjacent physical qubits

[1] To distinguish the elementary gate and high-level operations which is implemented by elementary gates, we call the former the *gate* and latter the *operation*.

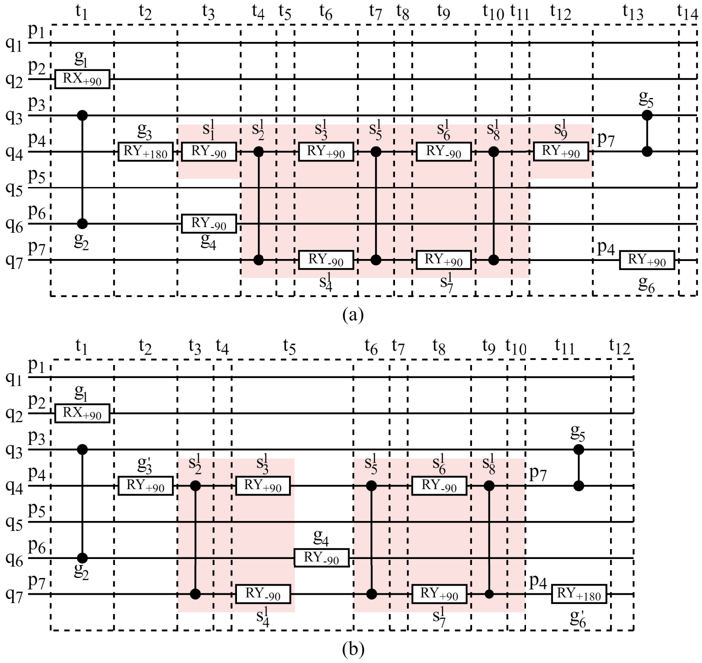
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14
q1 q2 q3 q4 q5 q6 q7
p1 p2 p3 p4 p5 p6 p7
g1 RX+90
g2 g3 RY+180 g4 RY-90 g5 g6 RY+90
s11 RY-90 s12 s13 RY+90 s14 RY-90 s15 s16 RY-90 s17 RY+90 s18 s19 RY+90
p7 p4
(a)
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
g1 RX+90 g2 g'3 RY+90 g4 RY-90 g5 g'6 RY+180
s12 s13 RY+90 s14 RY-90 s15 s16 RY-90 s17 RY+90 s18
p7 p4
(b)

Fig. 5: (a) A scheduled physical circuit for the logical circuit in Fig. 2 by inserting one SWAP operation without gate merging. (b) A scheduled physical circuit for the logical circuit in Fig. 2 by inserting one SWAP operation with gate merging.

other than p_{low}, physical qubits adjacent to p_{high} with frequency f_{low} except p_{low} are detuned to a parking frequency f_{low}^{park}. Similarly, physical qubits adjacent to p_{low} with frequency f_{high} except p_{high} need to stay at f_{high} and cannot be detuned to other frequencies. Note that physical qubits in the parking frequency cannot engage in any single-qubit or two-qubit gates. If two gates cannot execute in parallel due to some control constraints or they operate on the same qubit(s), then we say they are in conflict with each other.

Consider the quantum circuit composed of six gates in Fig. 2 and the architecture in Fig. 3. Given an initial mapping that maps q_i to p_i for $i = 1, 2, \ldots, 7$. We can schedule $\{g_1, g_2\}$, $\{g_3\}$, and $\{g_4\}$ in timestep t_1, t_2, and t_3, respectively, since g_2 satisfies the connectivity constraint under the initial mapping. But then we cannot schedule g_3 in timestep t_1 due to the control constraints. If g_3 is scheduled in timestep t_1, then two different types of single-qubit gates are performed on the physical qubits that share the same frequency at the same time. Gate $g_5\{q_3, q_4\}$ is not executable since q_3 and q_4 are mapped to p_3 and p_4 which are not adjacent on the architecture. To execute $g_5\{q_3, q_4\}$, we may insert $SWAP(q_7, q_4)$ to swap the physical qubits assigned to q_7 and q_4 at timestep t_3. Then, the mapping is updated to $\{q_1 \mapsto p_1, q_2 \mapsto p_2, q_3 \mapsto p_3, q_4 \mapsto p_7, q_5 \mapsto p_5, q_6 \mapsto p_6, q_7 \mapsto p_4\}$ which allows $g_5\{q_3, q_4\}$ to be executed subsequently. And we also schedule g_6 in the same timestep as g_5. Fig. 5(a) shows the final scheduled physical circuit where all gates are scheduled in 14 timesteps and a SWAP operation corresponding to the gate sequence $(s_1^1, s_2^1, \ldots, s_9^1)$ is inserted.

III. Problem Formulation

The control and timing-aware qubit routing problem is to determine a scheduled physical circuit with SWAP, MOVE, and BRIDGE operations insertion in order to execute all the gates in the original logical quantum circuit starting from an initial mapping with minimal increase in circuit latency. We formally define the control and timing-aware qubit routing problem below.

Definition. *(Control and timing-aware qubit routing problem) Given the connectivity constraint and control constraints of a quantum processor, a logical circuit, and an initial mapping of logical qubits to physical qubits, compute a scheduled physical circuit by inserting SWAP, MOVE, and BRIDGE operations such that the total latency of the final circuit is minimized and the gates that execute in each timestep satisfy the connectivity constraint and control constraints of the quantum processor.*

IV. Control and Timing-Aware Qubit Routing

We propose a new control and timing-aware qubit routing algorithm, CTQr. In each timestep, we formulate a maximum weighted independent set problem to determine a set of conflict-free[2] auxiliary operations to insert considering their costs and benefits. We note that the auxiliary operations (SWAP/MOVE/BRIDGE) begin or end with RY gates as shown in Fig. 4, so auxiliary operation insertion may result in contiguous RY gates. In our algorithm, auxiliary operations are inserted into the circuit with *gate merging* on the fly so that contiguous rotate gates of the same type that act on the same qubit are replaced by a single rotate gate with a rotation angle equal to sum of the rotation angles of the original gates, and we take this into account when the costs and benefits are computed.

Different from [17] which limits the number of auxiliary operations inserted in each timestep to one, we insert multiple auxiliary operations which can be scheduled in parallel to satisfy the connectivity constraint in a shorter time. Unlike [17] which performs gate merging on the whole circuit before and after routing, we do gate merging when we determine and insert the auxiliary operations during routing. Considering the effect of gate merging in the qubit routing process leads to better routing decisions [12]. Consider the architecture in Fig. 3 and the physical circuit in Fig. 5(a). We can replace g_3 and s_1^1 in Fig. 5(a) by a rotate gate g_3' as in Fig. 5(b), and replace s_9^1 and g_6 in Fig. 5(a) by a rotate gate g_6' as in Fig. 5(b) when we insert $SWAP(q_7, q_4)$ at timestep t_3. (g_6' is performed on p_4 since s_9^1 and g_6 are performed on p_4.). Then the delay of $SWAP(q_7, q_4)$ is reduced to 8 timesteps (t_3 to t_{10}).

The pseudo-code of our method is shown in Algorithm 1. In line 1, we perform gate merging on the input logical circuit LC_{in} to get LC. Initially, all gates are unscheduled. In lines 5 to 14, LC is the current logical circuit composed of all unscheduled gates, PC is the current physical circuit composed of all scheduled gates and it satisfies both connectivity constraint and control constraints, Π is the current mapping of logical qubits to physical qubits, and T is the current timestep. In each timestep, we call Procedure 1 to determine a desirable set of auxiliary operations to insert (line 7), and call Procedure 2 to insert the determined auxiliary operations with gate merging at the front of LC (line 8). Then, we use Procedure 3 to determine a suitable set of gates at the *front layer* (i.e. the gates that do not have any unscheduled predecessor) of LC to schedule in the current timestep T (line 9), remove them from LC (line 10), and insert them into PC (line 11). The gates we determine to schedule in the current timestep must satisfy the connectivity constraint

[2] If any of the 2-qubit elementary gates of the first auxiliary operation is in conflict with a 2-qubit elementary gate of the second auxiliary operation, then we say these two auxiliary operations are in conflict with each other.

Algorithm 1: Control and Timing-Aware Qubit Routing

```
Input  : Logical circuit $LC_{in}$, initial mapping $\Pi_{in}$
Output: A scheduled physical circuit $PC$ with initial mapping $\Pi_{in}$
        equivalent to $LC_{in}$ and satisfies the connectivity & control
        constraints
1  $LC \leftarrow$ Gate_Merging($LC_{in}$)
2  $PC \leftarrow$ empty circuit
3  $\Pi \leftarrow \Pi_{in}$
4  $T \leftarrow 1$
5  while $LC$ is not empty do
6      if $\exists$ $g$ in $Front\_Layer(LC)$: $g$ does not satisfy connectivity
         constraint under $\Pi$ then
7          $Selected\_Aux \leftarrow$ Select_Aux_Operations_to_Insert($LC$,
             $\Pi$)
8          $PC, LC \leftarrow$ Insert_with_Gate_Merging($PC$, $LC$,
             $Selected\_Aux$, $T$)
9      $Selected\_Gates \leftarrow$ Select_Gates_to_Schedule($LC$, $\Pi$)
10     $LC \leftarrow LC$ with gates in $Selected\_Gates$ removed
11     $PC \leftarrow PC$ concatenated with $Selected\_Gates$
12     Update $\Pi$ according to $Selected\_Gates$
13     Schedule all gates in $Selected\_Gates$ at timestep $T$
14     $T \leftarrow T + 1$
15 return $PC$
```

Procedure 1: Select_Aux_Operations_to_Insert

```
Input  : Logical circuit $LC$, current mapping $\Pi$
Output: A set of auxiliary operations $Selected\_Aux$ selected to be
        inserted
1  Find all helpful auxiliary operations that do not conflict with any
     two-qubit gate in $Front\_Layer(LC)$ which satisfies the
     connectivity constraint
2  Construct conflict graph $G_{aux} = (V, E)$ for the operations found
     above
3  $V_{sol} \leftarrow \phi$
4  while $G_{aux}$ is not empty do
5      Sort $V$ in lexicographical decreasing order of
         $(-Circuit\_Latency\_Overhead, \frac{Gain}{Delay})$
6      $V^{'} \leftarrow$ set of first $\lambda$ vertices in $V$
7      Construct $G^{'}_{aux} = (V^{'}, E^{'})$ where $(v_i, v_j) \in E^{'}$ if $v_i, v_j \in V^{'}$
         and $(v_i, v_j) \in E$
8      $V^{'}_{sol} \leftarrow$ Solve_MIS_Problem($G^{'}_{aux}$)
9      $V_{sol} \leftarrow V_{sol} \cup V^{'}_{sol}$
10     $G_{aux} \leftarrow G_{aux} \backslash (V^{'}_{sol} \cup Neighbors(V^{'}_{sol}))$
11 $Selected\_Aux \leftarrow$ Auxiliary operations corresponding to $V_{sol}$
12 return $Selected\_Aux$
```

and control constraints, and not conflict with any two-qubit gate that started execution at the previous timestep.

We elaborate on the details of Procedure 1 in subsection IV-A, Procedure 2 in subsection IV-B, and Procedure 3 in subsection IV-C.

A. Selection of Auxiliary Operations to Insert

Procedure 1 determines a set of conflict-free auxiliary operations to insert in parallel considering their costs and benefits. In line 1, we find all helpful auxiliary operations that do not conflict with any two-qubit gate in the front layer of LC which satisfies the connectivity constraint since we want these two-qubit gates to be scheduled immediately or in a short time. The helpful auxiliary operations are the SWAP and MOVE operations that bring closer the logical qubits of at least one two-qubit gate in the front layer of LC which violates the connectivity constraint in the architecture, and the BRIDGE operations that emulate a two-qubit gate in the front layer of LC on physical qubits at distance two.

When we insert an auxiliary operation, it is undesirable for the auxiliary operation to increase the final circuit latency, so we set the cost of an auxiliary operation as the *circuit latency overhead* which can be computed as follows.

$$Circuit\_latency\_overhead(Aux\_operation, LC) = \max_{g \in LC_{aux}} height(g) - \max_{g \in LC} height(g) \tag{1a}$$

$$height(g) = g_{delay} + \max_{g' \in succ(g)} height(g') \tag{1b}$$

where LC_{aux} is the circuit LC with the auxiliary operation inserted in the front with gate merging and $succ(g)$ represents the set of gates which depend on g.

The benefit of an auxiliary operation is defined as the gain obtained by inserting the operation divided by the delay of the operation. The *gain* is the total cost reduction of the gates in the front layer and the gates near the front layer due to the insertion of the auxiliary operation. The gain of a BRIDGE operation is set to 1. The gain of a SWAP operation, $Gain_{SWAP}$, and the gain of a MOVE operation, $Gain_{MOVE}$, are computed as follows.

$$Gain_{SWAP}(SWAP\{q_i, q_j\}) = \sum_{n=1}^{|LC_{q_i}|} \frac{D(\Pi, LC_{q_i}[n]) - D(\Pi', LC_{q_i}[n])}{n} + \sum_{n=1}^{|LC_{q_j}|} \frac{D(\Pi, LC_{q_j}[n]) - D(\Pi', LC_{q_j}[n])}{n} \tag{2a}$$

$$Gain_{MOVE}(MOVE\{q_i\}) = \sum_{n=1}^{|LC_{q_i}|} \frac{D(\Pi, LC_{q_i}[n]) - D(\Pi', LC_{q_i}[n])}{n} \tag{2b}$$

where LC_q is the first γ two-qubit gates[3] in the logical circuit LC that operates on logical qubit q, $LC_q[n]$ is the n-th gate in LC_q, Π is current mapping, Π' is the mapping after applying the SWAP/MOVE operation, and $D(\Pi, g)$ is the shortest distance between two logical qubits of g on the processor under mapping Π. So, the auxiliary operation that reduces more gate cost results in larger gain and the gates farther from the front layer have smaller effect on the gain. We also consider the operation *delay* since different auxiliary operations exhibit different delays. The delays of SWAP, MOVE, and BRIDGE operations are 10, 7, and 12 timesteps without gate merging, respectively. If we can apply gate merging on the inserted operation, then the delay is discounted accordingly.

In lines 2 to 11 of Procedure 1, we determine a set of conflict-free auxiliary operations to insert. In line 2, we construct a conflict graph $G_{aux} = (V, E)$ for the auxiliary operations. Each vertex of the conflict graph corresponds to an auxiliary operation. If auxiliary operation aux_i and aux_j are in conflict with each other, then there is an undirected edge between the vertices for aux_i and aux_j.

Afterwards, in lines 4 to 10, we determine an independent set (V_{sol} in line 3) on the conflict graph considering their cost and benefit. In each iteration, we extract λ vertices[4] with the highest priorities from the conflict graph (lines 5 to 6) to form a subgraph (line 7), we solve the maximum independent set problem on the subgraph (line 8) and add the subgraph solution to the final independent set (line 9), and update the conflict graph (line 10). If the conflict graph is not empty, then we start a new iteration with an updated conflict graph. Otherwise, the final independent set is a maximal independent set which

[3] γ is set to 5 in our implementation

[4] λ is set to 4 in our implementation

Procedure 2: Insert_with_Gate_Merging

```
Input  : Physical circuit $PC$, logical circuit $LC$, a set of auxiliary
         operations $Selected\_Aux$ to insert, current timestep $T$
Output: Updated physical circuit $PC_{update}$, updated logical circuit
         $LC_{aux}$
1  $PC_{update} \leftarrow PC$
2  $LC_{update} \leftarrow LC$
3  for $aux \in Selected\_Aux$ do
4      $Ele\_gates \leftarrow$ sequence of gates which implement $aux$
5      $g_{front} \leftarrow$ first gate in $Ele\_gates$
6      $g_{back} \leftarrow$ last gate in $Ele\_gates$
7      if $\exists\ g \in Last\_Layer(PC_{update})$: g and $g_{front}$ are mergeable
         then
8          Remove $g_{front}$ from $Ele\_gates$
9          $g_{merge} \leftarrow$ Gate_Merging($g$, $g_{front}$)
10         $timestep_{g_{merge}} \leftarrow timestep_g$
11         Replace $g$ in $PC_{update}$ by $g_{merge}$
12     if $\exists\ g \in Front\_Layer(LC_{update})$: $g_{back}$ and g are mergeable
         then
13         Remove $g_{back}$ from $Ele\_gates$
14         $g_{merge} \leftarrow$ Gate_Merging($g_{back}$, $g$)
15         Replace $g$ in $LC_{update}$ by $g_{merge}$
16     if $aux$ is BRIDGE then
17         $LC_{update} \leftarrow LC_{update}$ with the gate emulated by $aux$
             removed
18     for $g \in Ele\_gates$ do
19         if $g$ can be scheduled in $PC_{update}$ at timestep $T^{'}$ where
             $T^{'} < T$ then
20             Remove $g$ from $Ele\_gates$.
21             $PC_{update} \leftarrow PC_{update}$ concatenated with $g$
22             Schedule $g$ at timestep $T^{'}$
23         else
24             break
25     $LC_{update} \leftarrow LC_{update}$ with $Ele\_gates$ inserted in the front
26 return $PC_{update}$, $LC_{update}$
```

contains the vertices for the auxiliary operations selected. When we extract the vertices from the conflict graph, we assign higher priority to the auxiliary operation with smaller circuit latency overhead. When two vertices have the same circuit latency overhead, we give higher priority to the vertex with larger gain per unit delay.

B. Auxiliary Operation Insertion with Gate Merging

Procedure 2 inserts auxiliary operations with gate merging at the front of LC and schedule their constituent gates at the earliest possible timestep. For each auxiliary operation, let Ele_gates be the sequence of elementary gates that implement the auxiliary operation (line 4), g_{front} be the first gate in Ele_gates (line 5), and g_{back} be the last gate in Ele_gates (line 6). If there exists a gate g in the *last layer* (i.e. the gates that do not have any scheduled successor) of PC which can be merged with g_{front} (line 7), we remove g_{front} from Ele_gates (line 8), merge g_{front} and g to get g_{merge} (line 9), set the execution timestep of g_{merge} to be the same as g (line 10), and replace g in PC by g_{merge} (line 11). For example, when we insert SWAP operation at timestep t_3, g_3 and s_1^1 in Fig. 5(a) can be regarded as g and g_{front}, respectively, and g_3' in Fig. 5(b) can be regarded as g_{merge}. Similarly, if there exists a gate g in the front layer of LC which can be merged with g_{back} (line 12), we remove g_{back} from Ele_gates (line 13), merge g_{back} and g to get g_{merge} (line 14), and replace g in LC by g_{merge} (line 15). For example, when we insert SWAP operation at timestep t_3, s_9^1 and g_6 in Fig. 5(a) can be regarded as g_{back} and g, respectively, and g_6' in Fig. 5(b) can be regarded as g_{merge}. If the auxiliary operation is a BRIDGE operation (line 16), then we remove the gate in LC which is emulated by the BRIDGE operation (line 17).

Procedure 3: Select_Gates_to_Schedule

```
Input  : Circuit $LC$, current mapping $\Pi$
Output: A set of gates $Selected\_Gate$ selected to be scheduled
         together.
1  Find all gates in $Front\_Layer(LC)$ which are either single-qubit
     gates or two-qubit gates that satisfy connectivity constraint, and do
     not conflict with any two-qubit gate that started execution at the
     previous timestep
2  Construct conflict graph $G_{schedule}$ for the gates computed above
3  $Selected\_Gates \leftarrow Solve\_MWIS\_Problem(G_{schedule})$
4  return $Selected\_Gates$
```

It is possible that some gates in Ele_gates can be scheduled at a timestep before T which is desirable for reducing the circuit latency. So, for each gate g in Ele_gates (line 18), we try to find a timestep $T^{'}$ to schedule g in PC where $T^{'} < T$ and $T^{'}$ is as small as possible (line 19). If we find one, then we remove g from Ele_gates (line 20), add g into PC (line 21), and schedule g at timestep $T^{'}$ (line 22). Otherwise, we stop scheduling the remaining gates in Ele_gates before timestep T (line 24). Finally, we insert the remaining elementary gates in Ele_gates at the front of LC (line 25).

C. Selection of Gates to Schedule Concurrently

Procedure 3 determines a set of conflict-free gates to schedule together. First, we find all schedulable gates in line 1. A gate is schedulable if it is in the front layer of LC, it is a single-qubit gate or it is a two-qubit gate that satisfies the connectivity constraint, and it does not conflict with any two-qubit gate that started execution at the previous timestep.

But some of the schedulable gates are in conflict with one another due to control constraints or because they operate on the same logical qubit. We want to get a maximal subset of the schedulable gates to schedule together that will minimize the final circuit latency. So, we formulate a maximum weighted independent set problem for this purpose. In line 2, we construct a conflict graph for the schedulable gates. Each vertex of the conflict graph corresponds to a schedulable gate. If gates g_i and g_j are in conflict with each other, then there is an undirected edge between the vertices for g_i and g_j. The weight of a vertex is set equal to the height of the corresponding gate in the current logical circuit since the smaller the height of a gate, the less likely that it will affect the final circuit latency and so can be given less priority.

The maximum weighted independent set problem is a well-known NP-hard problem. A common strategy to solve the maximum weighted independent set problem is to iteratively extract a subgraph of large weight and solve the problem on the subgraph[13][2][4]. So, in line 3, we adopt a similar method like lines 4 to 10 of Procedure 1 to solve the maximum weighted independent set problem with the difference being that we sort the vertices in non-increasing order of the vertex weight. Afterwards, we select the gates corresponding to the vertices in the computed independent set to be scheduled together.

V. Experimental results

We implemented our method in C++. We chose 51 circuits from the IBM-QX circuit set [9] to be the benchmarks. In the experiment, we compared our method with [17] on the Surface-17 processor. We downloaded the source code of [17] from [16][5]

[5] In the implementation of [17], when a gate g_i is in conflict with a two-qubit gate g_j and g_j is scheduled earlier than g_i, it allows g_i to be scheduled in the second timestep of g_j's execution. Since the execution time of two-qubit gate is two timesteps, g_i should be scheduled at least two timesteps later then g_j, so we fixed this issue in the implementation.

TABLE I: Comparison of our qubit routing results with those by [17].

	[17]			CTQr				[17]			CTQr		
Name	$Latency$	$\#g$	$Runtime(s)$	$Latency$	$\#g$	$Runtime(s)$	Name	$Latency$	$\#g$	$Runtime(s)$	$Latency$	$\#g$	$Runtime(s)$
graycode6_47	**16**	**0**	<**0.001**	**16**	**0**	<**0.001**	pm1_249	6287	3539	0.029	**5458**	**2635**	**0.013**
xor5_254	**18**	**7**	<**0.001**	**18**	**7**	<**0.001**	cm42a_207	6287	3539	0.029	**5458**	**2635**	**0.013**
ham3_102	**56**	**14**	<**0.001**	**56**	22	<**0.001**	dc1_220	7131	3909	0.032	**6234**	**3544**	**0.015**
rd32-v0_66	106	37	<**0.001**	**97**	**36**	<**0.001**	squar5_261	7354	4356	0.036	**6266**	**3671**	**0.016**
alu-v0_27	127	**58**	<**0.001**	**110**	**58**	<**0.001**	sqrt8_260	11526	6695	0.053	**10207**	**5992**	**0.025**
miller_11	175	79	<**0.001**	**162**	**66**	<**0.001**	z4_268	11041	6199	0.051	**9721**	**5602**	**0.023**
4mod5-bdd_287	215	**81**	<**0.001**	**211**	82	<**0.001**	radd_250	11914	6749	0.055	**10913**	**6337**	**0.034**
one_two_three	213	**82**	<**0.001**	**195**	88	<**0.001**	adr4_197	12656	7313	0.059	**10622**	**5968**	**0.029**
decod24-bdd_294	209	**73**	<**0.001**	**206**	79	<**0.001**	sym6_145	15501	9097	0.073	**12089**	**5929**	**0.027**
alu-bdd_288	260	**95**	<**0.001**	**258**	134	<**0.001**	misex1_241	17604	9981	0.084	**15219**	**8116**	**0.038**
mini_alu_305	608	**366**	0.003	**575**	398	**0.002**	cycle10_2_110	22646	12679	0.103	**19863**	**11260**	**0.048**
sys6-v0_111	715	493	0.004	**686**	**438**	**0.002**	square_root_7	27998	17122	0.14	**22622**	**13750**	**0.06**
4gt12-v1_89	795	425	0.004	**705**	**333**	**0.002**	ham15_107	32870	18476	0.162	**27465**	**15113**	**0.07**
rd73_140	772	**443**	0.004	**736**	478	**0.002**	dc2_222	34977	19921	0.168	**30622**	**17594**	**0.077**
4gt4-v0_72	907	478	0.004	**847**	**473**	**0.002**	sqn_258	37729	21220	0.186	**32906**	**18764**	**0.079**
rd53_311	1016	650	0.005	**834**	**554**	**0.002**	cm85a_209	42471	23773	0.199	**37768**	**22070**	**0.095**
mini-alu_167	1152	624	0.005	**814**	**369**	**0.002**	root_255	59886	33384	0.305	**54517**	**32756**	**0.143**
sym9_146	1195	740	0.006	**927**	**541**	**0.002**	co14_215	63532	**39685**	0.37	**60004**	43219	**0.173**
decod24-enable_126	1188	599	0.005	**1068**	**507**	**0.002**	sym9_148	77921	42261	0.366	**66401**	**34660**	**0.156**
rd84_142	1127	819	0.007	**928**	**707**	**0.003**	life_238	79933	**43307**	0.382	**73624**	43658	**0.181**
mod8-10_177	1592	835	0.007	**1412**	**699**	**0.003**	max46_240	96931	54272	0.494	**85768**	**49298**	**0.213**
cnt3-5_180	1587	858	0.008	**1428**	**802**	**0.004**	clip_206	121615	68922	0.627	**111111**	**68768**	**0.295**
mod5adder_127	2466	1511	0.011	**1735**	**892**	**0.004**	9symml_195	128270	72507	0.693	**112682**	**64695**	**0.281**
sf_274	2828	1487	0.012	**2562**	**1305**	**0.006**	dist_223	134825	77801	0.745	**124220**	**76596**	**0.331**
wim_266	3457	1951	0.017	**2966**	**1578**	**0.007**	sym10_262	233892	129963	1.29	**212909**	**127101**	**0.553**
cm152a_212	4702	2624	0.021	**3850**	**1935**	**0.009**	Norm.	1	1	1	0.888	0.912	0.454

which was implemented in C++. Both our method and [17] were executed on a 64-bit Ubuntu server with AMD ThreadRipper 3970X CPU and 256GB memory. We used the same initial mapping as [17] for each testcase. The experimental results are shown in Table I. We report the latency and the number of additional elementary gates ($\#g$) of each final scheduled physical circuit, and the routing runtime.

Table I shows that CTQr can solve the control and timing-aware qubit routing problem with less execution time and better solution quality compared to [17]. The latency of the final scheduled physical circuit computed by our method is always smaller than or equal to that by [17], and we obtained smaller latency in 48 out of 51 benchmarks. The scheduled physical circuits produced by our method requires 11.2% fewer timesteps than [17] on average. Though minimizing the number of additional gates is not our primary objective, the number of additional gates in the final circuit produced by our method is 8.8% fewer than [17]. The routing runtime of CTQr is 54.6% less on average.

VI. Conclusion

In this paper, we proposed CTQr for solving the control and timing-aware qubit routing problem. In each timestep, we insert more than one auxiliary operations which can be scheduled simultaneously to satisfy the connectivity constraint in a shorter time. In addition, we consider gate merging when we select and insert auxiliary operations to make better routing decisions that lead to smaller final circuit latency. The experimental results show that CTQr outperforms the state-of-art approach with 11.2%, 8.8%, and 54.6% average reduction in the circuit latency, number of additional gates, and execution time, respectively.

References

[1] Quantum supremacy using a programmable superconducting processor. nature 574, 505–510, Arute, F., Arya, K., Babbush, R. et al. 2019.

[2] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986.

[3] S. Asaad, C. Dickel, N. K. Langford, S. Poletto, A. Bruno, M. A. Rol, D. Deurloo, and L. DiCarlo. Independent, extensible control of same-frequency superconducting qubits by selective broadcasting. *npj Quantum Information*, 2(1), aug 2016.

[4] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. SPAA '12, page 308–317, 2012.

[5] D. C. McKay, T. Alexander, et al. Qiskit backend specifications for openqasm and openpulse experiments, 2018 [Online].

[6] C.-Y. Huang, C.-H. Lien, and W.-K. Mak. Reinforcement learning and dear framework for solving the qubit mapping problem. In *ICCAD 2022*, pages 1–9, 2022.

[7] IBM. Ibm quantum experience, https://research.ibm.com/quantum-computing, [Online].

[8] IBM. Ibm quantum services https://quantum-computing.ibm.com/services?services=systems, [Online].

[9] Johannes Kepler University Linz Institute for Integrated Circuits. Iic jku - ibmqx qasm circuits., 2019 [Online].

[10] L. Lao, H. van Someren, I. Ashraf, and C. G. Almudever. Timing and resource-aware mapping of quantum circuits to superconducting processors. *IEEE TCAD*, 41(2):359–371, feb 2022.

[11] G. Li, Y. Ding, and Y. Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *ASPLOS 2019*, page 1001–1014, 2019.

[12] J. Liu, P. Li, and H. Zhou. Not all swaps have the same cost: A case for optimization-aware qubit routing, 2022.

[13] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC 1985*, page 1–10, 1985.

[14] D. Maslov, G. Dueck, D. Miller, and C. Negrevergne. Quantum circuit simplification and level compaction. *IEEE TCAD*, 27(3):436–444, mar 2008.

[15] S. Niu, A. Suau, G. Staffelbach, and A. Todri-Sanial. A hardware-aware heuristic for the qubit mapping problem in the NISQ era. *IEEE TQE*, 1:1–14, 2020.

[16] S. Park and D. Kim. Mcqa: Muti-constraint qubit allocation for near-term ftqc. https://github.com/CSDL-postech/MCQA, 2022.

[17] S. Park, D. Kim, J.-Y. Sim, and S. Kang. Mcqa: Multi-constraint qubit allocation for near-ftqc device. In *ICCAD 2022*, pages 1–9, 2022.

[18] B. Tan and J. Cong. Optimal layout synthesis for quantum computing. In *ICCAD 2020*, 2020.

[19] R. Versluis, S. Poletto, N. Khammassi, B. Tarasinski, N. Haider, D. Michalak, A. Bruno, K. Bertels, and L. DiCarlo. Scalable quantum circuit and control for a superconducting surface code. *Physical Review Applied*, 8(3), sep 2017.

[20] C. Zhang, A. B. Hayes, L. Qiu, Y. Jin, Y. Chen, and E. Z. Zhang. Time-optimal qubit mapping. In *ASPLOS 2021*, page 360–374, 2021.