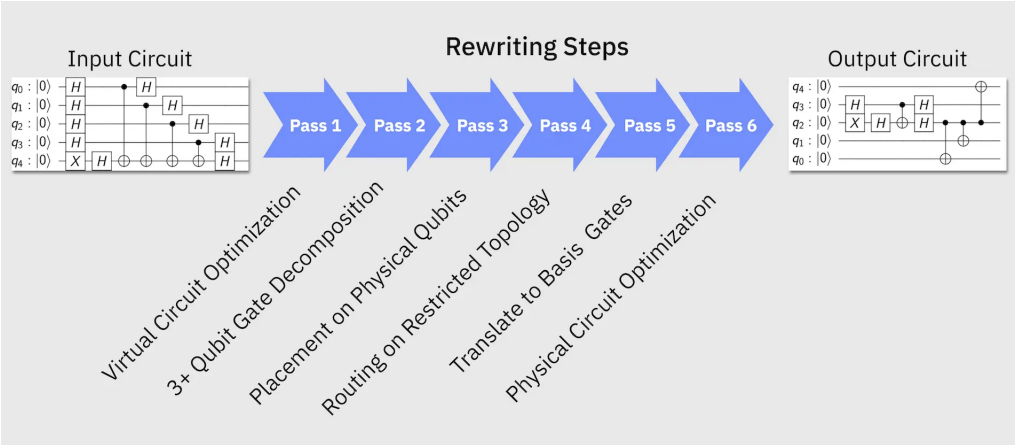
Transpiler

`qiskit.transpiler`

Overview

Transpilation is the process of rewriting a given input circuit to match the topology of a specific quantum device, and/or to optimize the circuit for execution on present day noisy quantum systems.

Most circuits must undergo a series of transformations that make them compatible with a given target device, and optimize them to reduce the effects of noise on the resulting outcomes. Rewriting quantum circuits to match hardware constraints and optimizing for performance can be far from trivial. The flow of logic in the rewriting tool chain need not be linear, and can often have iterative sub-loops, conditional branches, and other complex behaviors. That being said, the standard compilation flow follows the structure given below:



Qiskit has four pre-built transpilation pipelines available here: `qiskit.transpiler.preset_passmanagers`. Unless the reader is familiar with quantum circuit optimization methods and their usage, it is best to use one of these ready-made routines. By default the preset pass managers are composed of six stages:

1. `init` - This stage runs any initial passes that are required before we start embedding the circuit to the backend. This typically involves unrolling custom instructions and converting the circuit to all 1 and 2 qubit gates.
2. `layout` - This stage applies a layout, mapping the virtual qubits in the circuit to the physical qubits on a backend. See Layout Stage for more details.
3. `routing` - This stage runs after a layout has been applied and will inject gates (i.e. swaps) into the original circuit to make it compatible with the backend's connectivity. See Routing Stage for more details.
4. `translation` - This stage translates the gates in the circuit to the target backend's basis set. See Translation Stage for more details.
5. `optimization` - This stage runs the main optimization loop repeatedly until a condition (such as fixed depth) is reached. See Optimization Stage for more details.
6. `scheduling` - This stage is for any hardware-aware scheduling passes. See Scheduling Stage for more details.

When using `transpile()`, the implementation of each stage can be modified with the `*_method` arguments (e.g. `layout_method`). These can be set to one of the built-in methods and can also refer to available external plugins. See `qiskit.transpiler.preset_passmanagers.plugin` for details on this plugin interface.

Working with Preset Pass Managers

Qiskit includes functions to build preset `PassManager` objects. These preset passmanagers are used by the `transpile()` function for each optimization level. There are 4 optimization levels ranging from 0 to 3, where higher optimization levels take more time and computational effort but may yield a more optimal circuit. Optimization level 0 is intended for device characterization experiments and, as such, only maps the input circuit to the constraints of the target backend, without performing any optimizations. Optimization level 3 spends the most effort to optimize the circuit. However, as many of the optimization

techniques in the transpiler are heuristic based, spending more computational effort does not always result in an improvement in the quality of the output circuit.

If you'd like to work directly with a preset pass manager you can use the `generate_preset_pass_manager()` function to easily generate one. For example:

```
from qiskit.transpiler.preset_passmanagers impor
from qiskit.providers.fake_provider import Generic
Ba

backend = GenericBackendV2(num_qubits=5)
pass_manager = generate_preset_pass_manager(3, backe
```

which will generate a `StagedPassManager` object for optimization level 3 targeting the `GenericBackendV2` backend (equivalent to what is used internally by `transpile()` with `backend=GenericBackendV2(5)` and `optimization_level=3`). You can use this just like you would any other `PassManager`. However, because it is a `StagedPassManager` it also makes it easy to compose and/or replace stages of the pipeline. For example, if you wanted to run a custom scheduling stage using dynamical decoupling (via the `PadDynamicalDecoupling` pass) and also add initial logical optimization prior to routing, you would do something like (building off the previous example):

```
import numpy as np
from qiskit.circuit.library import HGate, PhaseGate
from qiskit.transpiler import PassManager
from qiskit.transpiler.passes import (
    ALAPScheduleAnalysis,
    CXCancellation,
    InverseCancellation,
    PadDynamicalDecoupling,
)

dd_sequence = [XGate(), XGate()]
scheduling_pm = PassManager(
    [
        ALAPScheduleAnalysis(target=backend.target)
        PadDynamicalDecoupling(target=backend.targe
    ]
)
inverse_gate_list = [
    HGate(),
    (RXGate(np.pi / 4), RXGate(-np.pi / 4)),
    (PhaseGate(np.pi / 4), PhaseGate(-np.pi / 4)),
```

```
    (TGate(), TdgGate()),
]
logical_opt = PassManager(
    [
        CXCancellation(),
        InverseCancellation(inverse_gate_list),
    ]
)


# Add pre-layout stage to run extra logical optimiz
pass_manager.pre_layout = logical_opt
# Set scheduling stage to custom pass manager
pass_manager.scheduling = scheduling_pm
```

Now, when the staged pass manager is run via the `run()` method, the `logical_opt` pass manager will be called before the `layout` stage, and the `scheduling_pm` pass manager will be used for the `scheduling` stage instead of the default.

Custom Pass Managers

In addition to modifying preset pass managers, it is also possible to construct a pass manager to build an entirely custom pipeline for transforming input circuits. You can use the `StagedPassManager` class directly to do this. You can define arbitrary stage names and populate them with a `PassManager` instance. For example, the following code creates a new `StagedPassManager` that has 2 stages, `init` and `translation`.:

```
from qiskit.transpiler.passes import (
    UnitarySynthesis,
    Collect2qBlocks,
    ConsolidateBlocks,
    UnitarySynthesis,
    Unroll3qOrMore,
)
from qiskit.transpiler import PassManager, StagedPa

basis_gates = ["rx", "ry", "rxx"]
init = PassManager([UnitarySynthesis(basis_gates, m
translate = PassManager(
    [
        Collect2qBlocks(),
        ConsolidateBlocks(basis_gates=basis_gates),
```

```
        UnitarySynthesis(basis_gates),
    ]
)

staged_pm = StagedPassManager(
    stages=["init", "translation"], init=init, tran
)
```

There is no limit on the number of stages you can put in a `StagedPassManager`.

The Stage Generator Functions may be useful for the construction of custom `generate_embed_passmanager` generates a `PassManager` to “embed” a selected initial `Layout` from a layout pass to the specified target device.

Representing Quantum Computers

To be able to compile a `QuantumCircuit` for a specific backend, the transpiler needs a specialized representation of that backend, including its constraints, instruction set, qubit properties, and more, to be able to compile and optimize effectively. While the `BackendV2` class defines an interface for querying and interacting with backends, its scope is larger than just the transpiler’s needs including managing job submission and potentially interfacing with remote services. The specific information needed by the transpiler is described by the `Target` class

For example, to construct a simple `Target` object, one can iteratively add descriptions of the instructions it supports:

```
from qiskit.circuit import Parameter, Measure
from qiskit.transpiler import Target, InstructionPr
from qiskit.circuit.library import UGate, RZGate, R

target = Target(num_qubits=3)
target.add_instruction(CXGate(), {(0, 1): Instructi
target.add_instruction(
    UGate(Parameter('theta'), Parameter('phi'), Par
    {
        (0,): InstructionProperties(error=.00001, d
        (1,): InstructionProperties(error=.00002, d
    }
)
```

```
target.add_instruction(
    RZGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RYGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RXGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    CZGate(),
    {
        (1, 2): InstructionProperties(error=.0001,
        (2, 0): InstructionProperties(error=.0001,
    }
)
target.add_instruction(
    Measure(),
    {
        (0,): InstructionProperties(error=.001, dur
        (1,): InstructionProperties(error=.002, dur
        (2,): InstructionProperties(error=.2, durat
    }
)
print(target)
```

```
Target
Number of qubits: 3
Instructions:
    cx
        (0, 1):
            Duration: 5e-07 sec.
            Error Rate: 0.0001
    u
        (0,):
            Duration: 5e-08 sec.
            Error Rate: 1e-05
        (1,):
            Duration: 6e-08 sec.
```

```
            Error Rate: 2e-05
    rz
        (1,):
            Duration: 5e-08 sec.
            Error Rate: 1e-05
        (2,):
            Duration: 6e-08 sec.
            Error Rate: 2e-05
    ry
        (1,):
            Duration: 5e-08 sec.
            Error Rate: 1e-05
        (2,):
            Duration: 6e-08 sec.
            Error Rate: 2e-05
    rx
        (1,):
            Duration: 5e-08 sec.
            Error Rate: 1e-05
        (2,):
            Duration: 6e-08 sec.
            Error Rate: 2e-05
    cz
        (1, 2):
            Duration: 5e-07 sec.
            Error Rate: 0.0001
        (2, 0):
            Duration: 5e-07 sec.
            Error Rate: 0.0001
    measure
        (0,):
            Duration: 5e-05 sec.
            Error Rate: 0.001
        (1,):
            Duration: 6e-05 sec.
            Error Rate: 0.002
        (2,):
            Duration: 5e-07 sec.
            Error Rate: 0.2
```

This `Target` represents a 3 qubit backend that supports `CXGate` between qubits 0 and 1, `UGate` on qubits 0 and 1, `RZGate`, `RXGate`, and `RYGate` on qubits 1 and 2, `CZGate` between qubits 1 and 2, and qubits 2 and 0, and `Measure` on all qubits.

There are also specific data structures to represent a specific subset of information from the `Target`. For example, the `CouplingMap` class is used to solely represent the connectivity constraints of a backend as a directed graph. A coupling map can be generated from a `Target` using the `Target.build_coupling_map()` method. These

`Target` using the `Target.build_coupling_map()` method. These data structures typically pre-date the `Target` class but are still used by some transpiler passes that do not work natively with a `Target` instance yet or when dealing with backends that aren't using the latest `BackendV2` interface.

For example, if we wanted to visualize the `CouplingMap` for the example 3 qubit `Target` above:

```
from qiskit.circuit import Parameter, Measure
from qiskit.transpiler import Target, InstructionPr
from qiskit.circuit.library import UGate, RZGate, R

target = Target(num_qubits=3)
target.add_instruction(CXGate(), {(0, 1): Instructi
target.add_instruction(
    UGate(Parameter('theta'), Parameter('phi'), Par
    {
        (0,): InstructionProperties(error=.00001, d
        (1,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RZGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RYGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RXGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    CZGate(),
    {
        (1, 2): InstructionProperties(error=.0001,
        (2, 0): InstructionProperties(error=.0001,
    }
)
target.add_instruction(
```

```
    Measure(),
    {
        (0,): InstructionProperties(error=.001, dur
        (1,): InstructionProperties(error=.002, dur
        (2,): InstructionProperties(error=.2, durat
    }
)

target.build_coupling_map().draw()
```

This shows the global connectivity of the `Target` which is the combination of the supported qubits for `CXGate` and `CZGate`. To see the individual connectivity, you can pass the operation name to `CouplingMap.build_coupling_map()`:

```
from qiskit.circuit import Parameter, Measure
from qiskit.transpiler import Target, InstructionPr
from qiskit.circuit.library import UGate, RZGate, R

target = Target(num_qubits=3)
target.add_instruction(CXGate(), {(0, 1): Instructi
target.add_instruction(
    UGate(Parameter('theta'), Parameter('phi'), Par
    {
        (0,): InstructionProperties(error=.00001, d
        (1,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RZGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RYGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RXGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
```

```
    CZGate(),
    {
        (1, 2): InstructionProperties(error=.0001, 
        (2, 0): InstructionProperties(error=.0001, 
    }
)
target.add_instruction(
    Measure(),
    {
        (0,): InstructionProperties(error=.001, dur
        (1,): InstructionProperties(error=.002, dur
        (2,): InstructionProperties(error=.2, durat
    }
)

target.build_coupling_map('cx').draw()
```

```
from qiskit.circuit import Parameter, Measure
from qiskit.transpiler import Target, InstructionPr
from qiskit.circuit.library import UGate, RZGate, R

target = Target(num_qubits=3)
target.add_instruction(CXGate(), {(0, 1): Instructi
target.add_instruction(
    UGate(Parameter('theta'), Parameter('phi'), Par
    {
        (0,): InstructionProperties(error=.00001, d
        (1,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RZGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RYGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
target.add_instruction(
    RXGate(Parameter('theta')),
    {
        (1,): InstructionProperties(error=.00001, d
        (2,): InstructionProperties(error=.00002, d
    }
)
```

```
)
target.add_instruction(
    CZGate(),
    {
        (1, 2): InstructionProperties(error=.0001, 
        (2, 0): InstructionProperties(error=.0001, 
    }
)
target.add_instruction(
    Measure(),
    {
        (0,): InstructionProperties(error=.001, dur
        (1,): InstructionProperties(error=.002, dur
        (2,): InstructionProperties(error=.2, durat
    }
)

target.build_coupling_map('cz').draw()
```

Transpiler Stage Details

Below are a description of the default transpiler stages and the problems they solve. The default passes used for each stage are described, but the specifics are configurable via the `*_method` keyword arguments for the `transpile()` and `generate_preset_pass_manager()` functions which can be used to override the methods described in this section.

Translation Stage

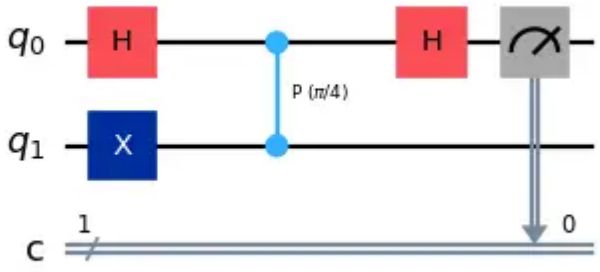
When writing a quantum circuit you are free to use any quantum gate (unitary operator) that you like, along with a collection of non-gate operations such as qubit measurements and reset operations. However, most quantum devices only natively support a handful of quantum gates and non-gate operations. The allowed instructions for a given backend can be found by querying the `Target` for the devices:

```
from qiskit.providers.fake_provider import Gener
backend = GenericBackendV2(5)

print(backend.target)
```

Every quantum circuit run on the target device must be expressed

Every quantum circuit run on the target device must be expressed using only these instructions. For example, to run a simple phase estimation circuit:

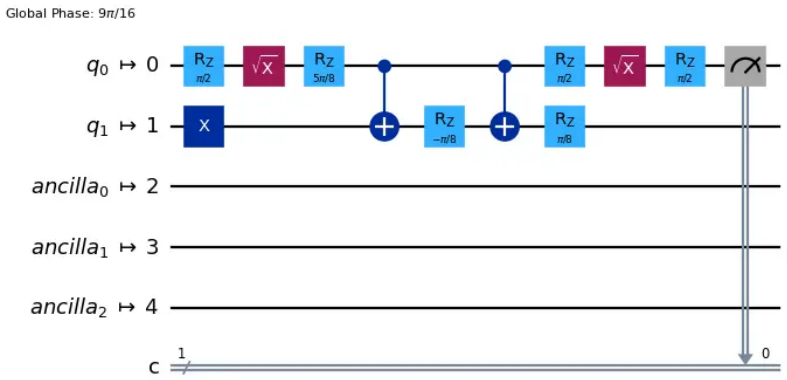
```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.providers.fake_provider import GenericB

backend = GenericBackendV2(5)

qc = QuantumCircuit(2, 1)

qc.h(0)
qc.x(1)
qc.cp(np.pi/4, 0, 1)
qc.h(0)
qc.measure([0], [0])
qc.draw(output='mpl')
```



We have H, X, and controlled-P gates, none of which are in our device's basis gate set, and thus must be translated. We can transpile the circuit to show what it will look like in the native gate set of the target IBM Quantum device (the `GenericBackendV2` class generates a fake backend with a specified number of qubits for test purposes):

```
from qiskit import transpile
from qiskit import QuantumCircuit
from qiskit.providers.fake_provider import GenericB

backend = GenericBackendV2(5)

qc = QuantumCircuit(2, 1)

qc.h(0)
qc.x(1)
qc.cp(np.pi/4, 0, 1)
qc.h(0)
qc.measure([0], [0])
```

```
qc.measure([0], [0])

qc_basis = transpile(qc, backend)
qc_basis.draw(output='mpl')
```



A few things to highlight. First, the circuit has gotten longer with respect to the original. This can be verified by checking the depth of both circuits:

```
print('Original depth:', qc.depth(), 'Decomposed Dep
```

```
Original depth: 4 Decomposed Depth: 10
```

Second, although we had a single controlled gate, the fact that it was not in the basis set means that, when expanded, it requires more than a single `CXGate` to implement. All said, unrolling to the basis set of gates leads to an increase in the depth of a quantum circuit and the number of gates.

It is important to highlight two special cases:

1. If A swap gate is not a native gate and must be decomposed this requires three CNOT gates:

   ```
   from qiskit.providers.fake_provider import G
   backend = GenericBackendV2(5)

   print(backend.operation_names)
   ```

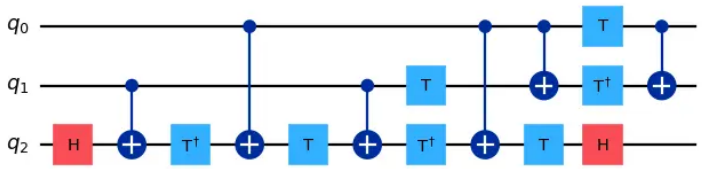
   ```
   ['id', 'rz', 'sx', 'x', 'cx', 'measure', 'delay']
   ```

   As a product of three CNOT gates, swap gates are expensive operations to perform on noisy quantum devices. However, such operations are usually necessary for embedding a circuit into the

limited gate connectivities of many devices. Thus, minimizing the number of swap gates in a circuit is a primary goal in the transpilation process.

2. A Toffoli, or controlled-controlled-not gate (`ccx`), is a three-qubit gate. Given that our basis gate set includes only single- and two-qubit gates, it is obvious that this gate must be decomposed. This decomposition is quite costly:

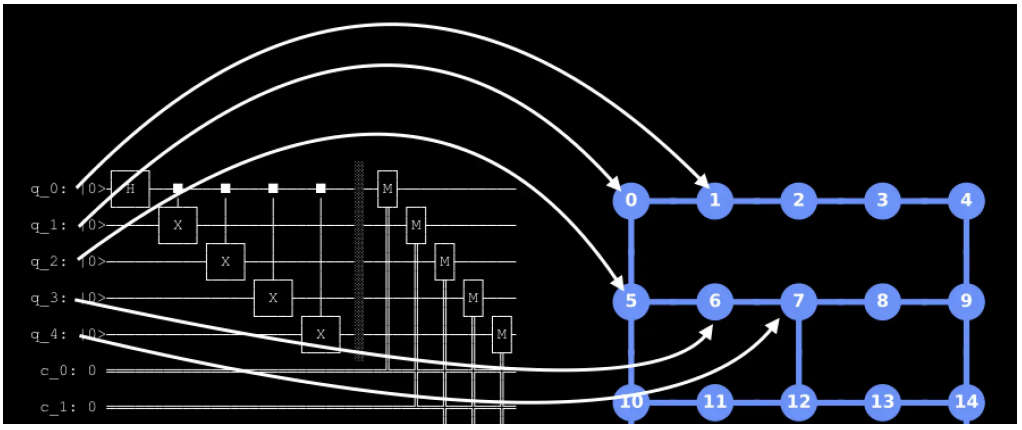
```
from qiskit.circuit import QuantumCircuit

ccx_circ = QuantumCircuit(3)
ccx_circ.ccx(0, 1, 2)
ccx_circ.decompose().draw(output='mpl')
```

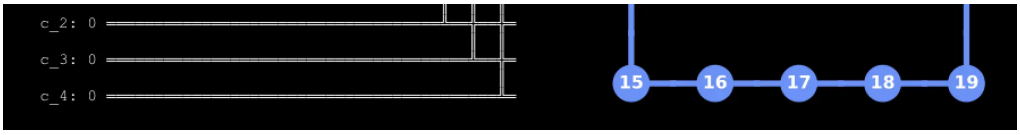


For every Toffoli gate in a quantum circuit, the hardware may execute up to six CNOT gates, and a handful of single-qubit gates. From this example, it should be clear that any algorithm that makes use of multiple Toffoli gates will end up as a circuit with large depth and will therefore be appreciably affected by noise and gate errors.

Layout Stage

Quantum circuits are abstract entities whose qubits are “virtual” representations of actual qubits used in computations. We need to be able to map these virtual qubits in a one-to-one manner to the “physical” qubits in an actual quantum device.





By default, qiskit will do this mapping for you. The choice of mapping depends on the properties of the circuit, the particular device you are targeting, and the optimization level that is chosen. The choice of initial layout is extremely important for minimizing the number of swap operations needed to map the input circuit onto the device topology and for minimizing the loss due to non-uniform noise properties across a device. Due to the importance of this stage, the preset pass managers try a few different methods to find the best layout. Typically this involves 2 steps: first, trying to find a “perfect” layout (a layout which does not require any swap operations), and then, a heuristic pass that tries to find the best layout to use if a perfect layout cannot be found. There are 2 passes typically used for the first stage:

- `VF2Layout`: Models layout selection as a subgraph isomorphism problem and tries to find a subgraph of the connectivity graph that is isomorphic to the graph of 2 qubit interactions in the circuit. If more than one isomorphic mapping is found a scoring heuristic is run to select the mapping which would result in the lowest average error when executing the circuit.
- `TrivialLayout`: Maps each virtual qubit to the same numbered physical qubit on the device, i.e. `[0,1,2,3,4]` -> `[0,1,2,3,4]`. This is historical behavior used only in `optimization_level=1` to try to find a perfect layout. If it fails to do so, `VF2Layout` is tried next.

Next, for the heuristic stage, 2 passes are used by default:

- `SabreLayout`: Selects a layout by starting from an initial random layout and then repeatedly running a routing algorithm (by default `SabreSwap`) both forward and backward over the circuit, using the permutation caused by swap insertions to adjust that initial random layout. For more details you can refer to the paper describing the algorithm: arXiv:1809.02573 `SabreLayout` is used to select a layout if a perfect layout isn’t found for optimization levels 1, 2, and 3.
- `TrivialLayout`: Always used for the layout at optimization level 0.

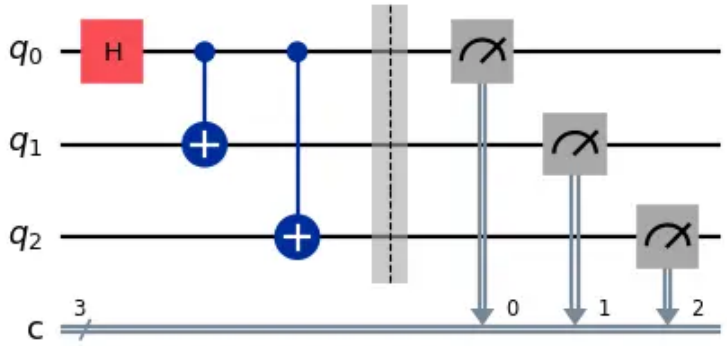
There are other passes than can be used for the heuristic stage, but are not included in the default pipeline, such as:

are not included in the default pipeline, such as:

- `DenseLayout`: Finds the sub-graph of the device with greatest connectivity that has the same number of qubits as the circuit.

Let's see what layouts are automatically picked at various optimization levels. The circuits returned by `qiskit.compiler.transpile()` are annotated with this initial layout information, and we can view this layout selection graphically using `qiskit.visualization.plot_circuit_layout()`:

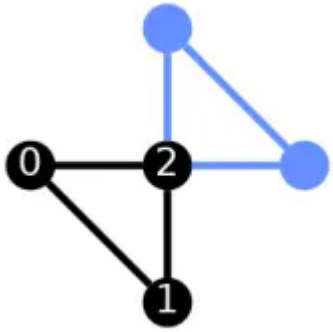
```
from qiskit import QuantumCircuit, transpile
from qiskit.visualization import plot_circuit_layou
from qiskit.providers.fake_provider import Fake5QV1
backend = Fake5QV1()

ghz = QuantumCircuit(3, 3)
ghz.h(0)
ghz.cx(0,range(1,3))
ghz.barrier()
ghz.measure(range(3), range(3))
ghz.draw(output='mpl')
```



- **Layout Using Optimization Level 0**

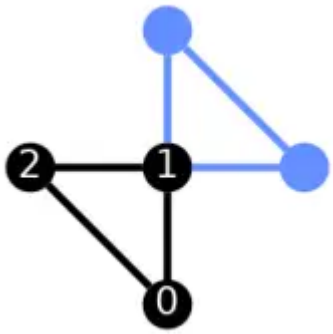
```
from qiskit import QuantumCircuit, tr
from qiskit.visualization import plot_circ
from qiskit.providers.fake_provider import
backend = Fake5QV1()

ghz = QuantumCircuit(3, 3)
ghz.h(0)
ghz.cx(0,range(1,3))
ghz.barrier()
ghz.measure(range(3), range(3))

new_circ_lv0 = transpile(ghz, backend=back
```

```
new_circ_lv0 = transpile(ghz, backend=back
plot_circuit_layout(new_circ_lv0, backend)
```



- **Layout Using Optimization Level 3**

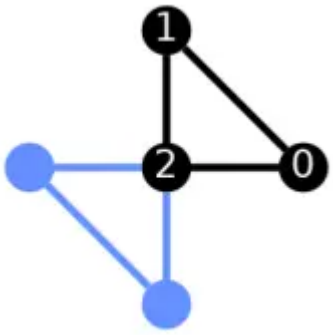
```
from qiskit import QuantumCircuit, tr
from qiskit.visualization import plot_circ
from qiskit.providers.fake_provider import
backend = Fake5QV1()

ghz = QuantumCircuit(3, 3)
ghz.h(0)
ghz.cx(0,range(1,3))
ghz.barrier()
ghz.measure(range(3), range(3))

new_circ_lv3 = transpile(ghz, backend=back
plot_circuit_layout(new_circ_lv3, backend)
```



It is possible to override automatic layout selection by specifying an initial layout. To do so we can pass a list of integers to `qiskit.compiler.transpile()` via the initial_layout keyword argument, where the index labels the virtual qubit in the circuit and the corresponding value is the label for the physical qubit to map onto:

```
from qiskit import QuantumCircuit, transpile
from qiskit.visualization import plot_circuit_layou
```

```
from qiskit.providers.fake_provider import Fake5QV1
backend = Fake5QV1()

ghz = QuantumCircuit(3, 3)
ghz.h(0)
ghz.cx(0,range(1,3))
ghz.barrier()
ghz.measure(range(3), range(3))

# Virtual -> physical
#    0    ->    3
#    1    ->    4
#    2    ->    2

my_ghz = transpile(ghz, backend, initial_layout=[3,
plot_circuit_layout(my_ghz, backend)
```



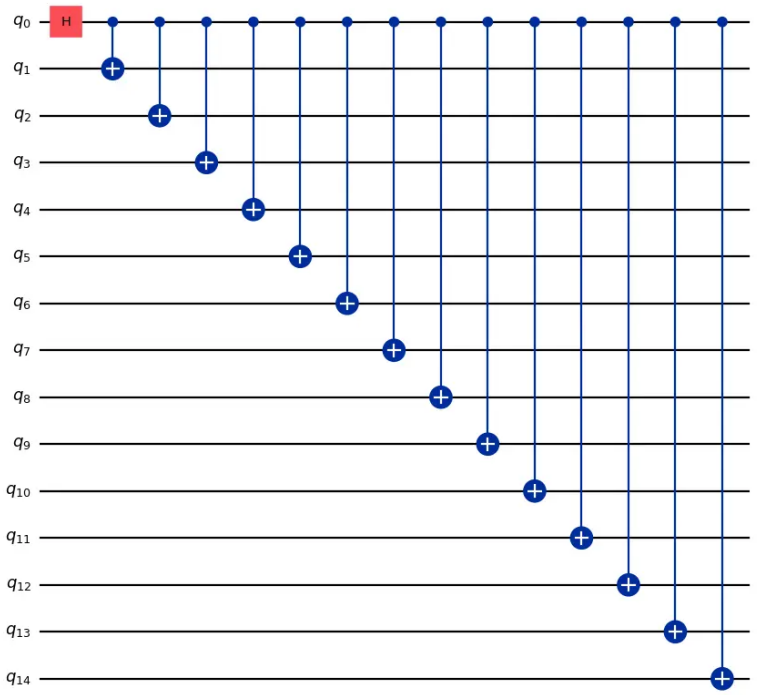
Routing Stage

In order to implement a 2-qubit gate between qubits in a quantum circuit that are not directly connected on a quantum device, one or more swap gates must be inserted into the circuit to move the qubit states around until they are adjacent on the device gate map. Each swap gate typically represents an expensive and noisy operation to perform. Thus, finding the minimum number of swap gates needed to map a circuit onto a given device, is an important step (if not the most important) in the whole execution process.

However, as with many important things in life, finding the optimal swap mapping is hard. In fact it is in a class of problems called NP-hard, and is thus prohibitively expensive to compute for all but the smallest quantum devices and input circuits. To get around this, by default Qiskit uses a stochastic heuristic algorithm called `SabreSwap` to compute a good, but not necessarily optimal swap mapping. The use of a stochastic method means the circuits generated by

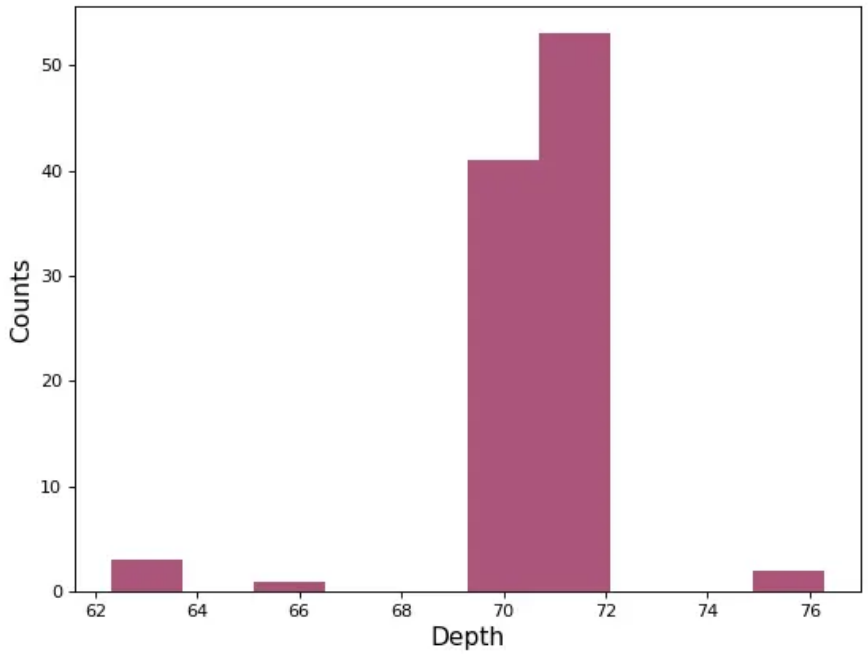
`transpile()` are not guaranteed to be the same over repeated runs. Indeed, running the same circuit repeatedly will in general result in a distribution of circuit depths and gate counts at the output.

In order to highlight this, we run a GHZ circuit 100 times, using a "bad" (disconnected) `initial_layout` in a heavy hex coupling map:




```
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, transpile
from qiskit.providers.fake_provider import GenericB
from qiskit.transpiler import CouplingMap

coupling_map = CouplingMap.from_heavy_hex(3)
backend = GenericBackendV2(coupling_map.size(), cou

ghz = QuantumCircuit(15)
ghz.h(0)
ghz.cx(0, range(1, 15))

depths = []
for i in range(100):
    depths.append(
        transpile(
            ghz,
            backend,
            seed_transpiler=i,
```

```
            layout_method='trivial'  # Fixed layout
        ).depth()
    )

plt.figure(figsize=(8, 6))
plt.hist(depths, align='left', color='#AC557C')
plt.xlabel('Depth', fontsize=14)
plt.ylabel('Counts', fontsize=14);
```



This distribution is quite wide, signaling the difficulty the swap mapper is having in computing the best mapping. Most circuits will have a distribution of depths, perhaps not as wide as this one, due to the stochastic nature of the default swap mapper. Of course, we want the best circuit we can get, especially in cases where the depth is critical to success or failure. The `SabreSwap` pass will by default by run its algorithm in parallel with multiple seed values and select the output which uses the fewest swaps. If you would like to increase the number of trials `SabreSwap` runs you can refer to Working with Preset Pass Managers and modify the `routing` stage with a custom instance of `SabreSwap` with a larger value for the `trials` argument.

Typically, following the swap mapper, the routing stage in the preset pass managers also includes running the `VF2PostLayout` pass. As its name implies, `VF2PostLayout` uses the same basic algorithm as `VF2Layout`, but instead of using it to find a perfect initial layout, it is designed to run after mapping and try to find a layout on qubits with lower error rates which will result in better output fidelity when running the circuit. The details of this algorithm are described in arXiv:2209.15512

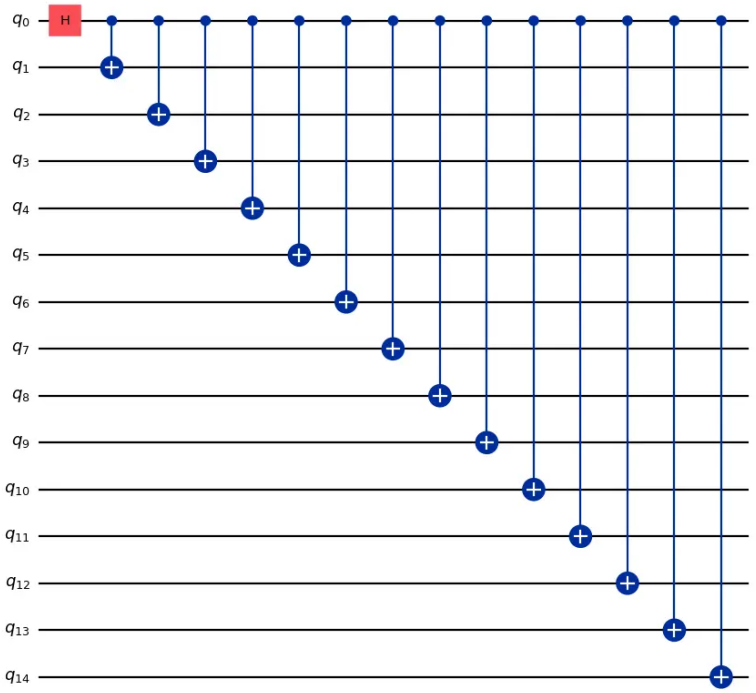
arXiv:2209.15512 .

Optimization Stage

Decomposing quantum circuits into the basis gate set of the target device, and the addition of swap gates needed to match hardware topology, conspire to increase the depth and gate count of quantum circuits. Fortunately many routines for optimizing circuits by combining or eliminating gates exist. In some cases these methods are so effective the output circuits have lower depth than the inputs. In other cases, not much can be done, and the computation may be difficult to perform on noisy devices. Different gate optimizations are turned on with different `optimization_level` values. Below we show the benefits gained from setting the optimization level higher:

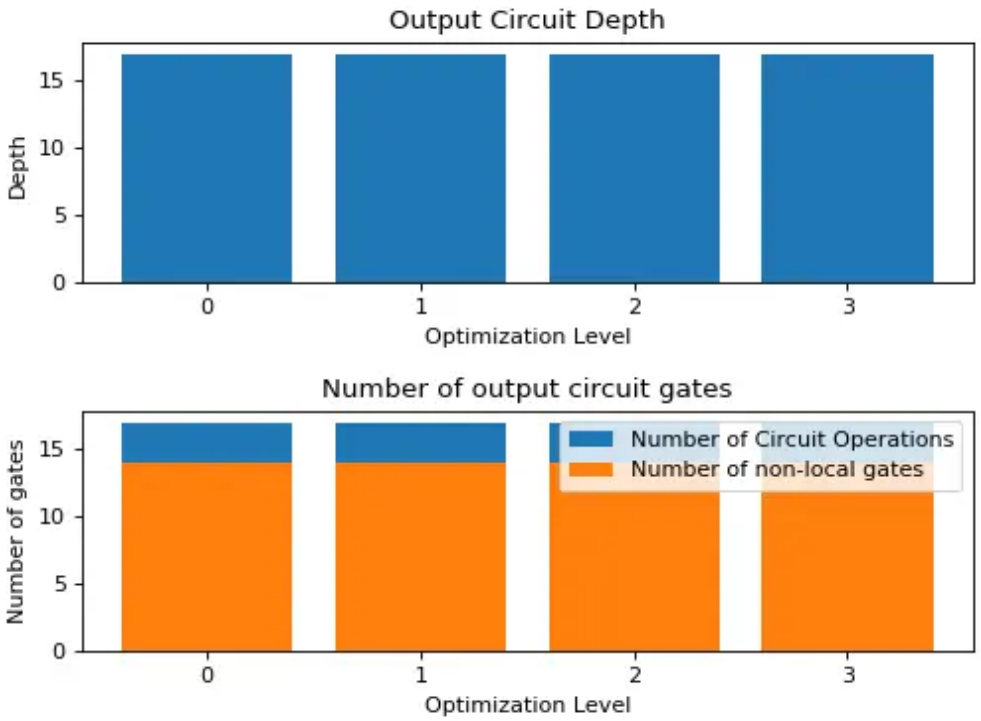
Important

The output from `transpile()` varies due to the stochastic swap mapper. So the numbers below will likely change each time you run the code.



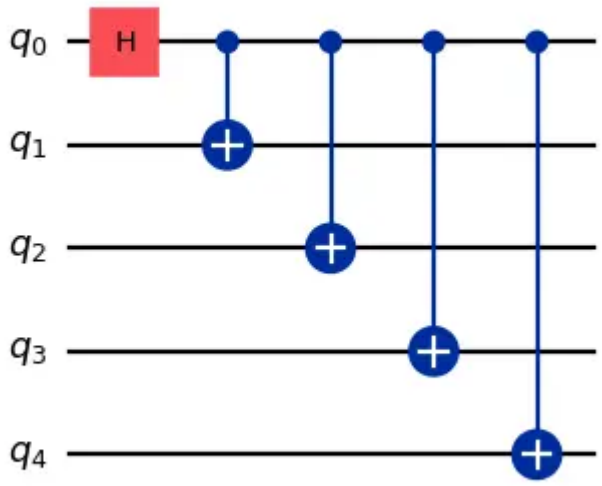

```
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, transpile
```

```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.fake_provider import GenericB
backend = GenericBackendV2(16)

ghz = QuantumCircuit(15)
ghz.h(0)
ghz.cx(0, range(1, 15))

depths = []
gate_counts = []
non_local_gate_counts = []
levels = [str(x) for x in range(4)]
for level in range(4):
    circ = transpile(ghz, backend, optimization_le
    depths.append(circ.depth())
    gate_counts.append(sum(circ.count_ops().values
    non_local_gate_counts.append(circ.num_nonlocal
fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.bar(levels, depths, label='Depth')
ax1.set_xlabel("Optimization Level")
ax1.set_ylabel("Depth")
ax1.set_title("Output Circuit Depth")
ax2.bar(levels, gate_counts, label='Number of Circu
ax2.bar(levels, non_local_gate_counts, label='Numbe
ax2.set_xlabel("Optimization Level")
ax2.set_ylabel("Number of gates")
ax2.legend()
ax2.set_title("Number of output circuit gates")
fig.tight_layout()
plt.show()
```



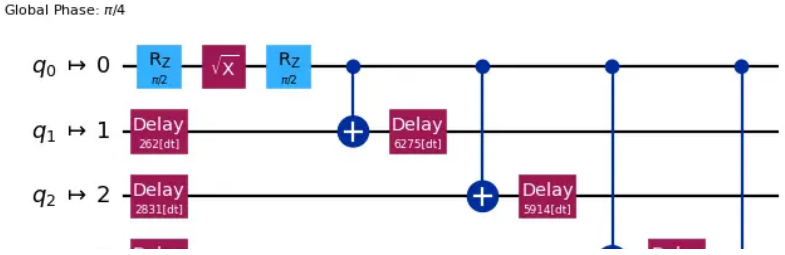
Scheduling Stage

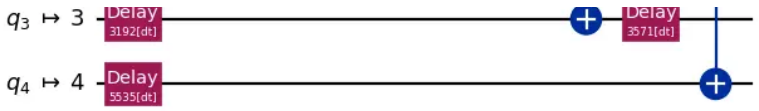
After the circuit has been translated to the target basis, mapped to the device, and optimized, a scheduling phase can be applied to optionally account for all the idle time in the circuit. At a high level, the scheduling can be thought of as inserting delays into the circuit to account for idle time on the qubits between the execution of instructions. For example, if we start with a circuit such as:



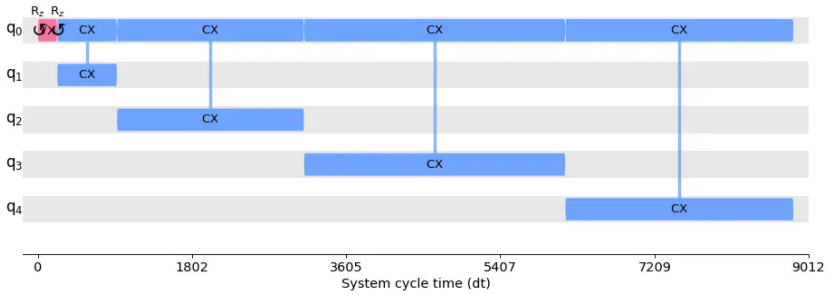
we can then call `transpile()` on it with `scheduling_method` set:

```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.fake_provider import GenericB
backend = GenericBackendV2(5)
ghz = QuantumCircuit(5)
ghz.h(0)
ghz.cx(0,range(1,5))
circ = transpile(ghz, backend, scheduling_method="a
circ.draw(output='mpl')
```





You can see here that the transpiler inserted `Delay` instructions to account for idle time on each qubit. To get a better idea of the timing of the circuit we can also look at it with the `timeline.draw()` function:



The scheduling of a circuit involves two parts: analysis and constraint mapping, followed by a padding pass. The first part requires running a scheduling analysis pass such as `ALAPSchedulingAnalysis` or `ASAPSchedulingAnalysis` which analyzes the circuit and records the start time of each instruction in the circuit using a scheduling algorithm ("as late as possible" for `ALAPSchedulingAnalysis` and "as soon as possible" for `ASAPSchedulingAnalysis`) in the property set. Once the circuit has an initial scheduling, additional passes can be run to account for any timing constraints on the target backend, such as alignment constraints. This is typically done with the `ConstrainedReschedule` pass which will adjust the scheduling set in the property set to the constraints of the target backend. Once all the scheduling and adjustments/rescheduling are finished, a padding pass, such as `PadDelay` or `PadDynamicalDecoupling` is run to insert the instructions into the circuit, which completes the scheduling.

Scheduling Analysis with control flow instructions

When running scheduling analysis passes on a circuit, you must keep in mind that there are additional constraints on classical conditions and control flow instructions. This section covers the details of these additional constraints that any scheduling pass will need to account for.

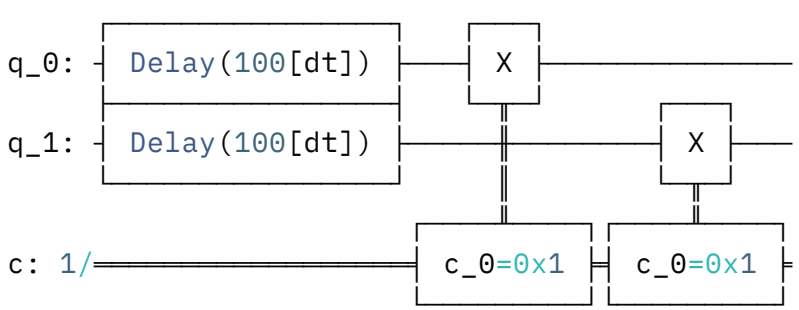
Topological node ordering in scheduling

The DAG representation of `QuantumCircuit` respects the node

ordering in the classical register wires, though theoretically two conditional instructions conditioned on the same register could commute, i.e. read-access to the classical register doesn't change its state.

```
qc = QuantumCircuit(2, 1)
qc.delay(100, 0)
qc.x(0).c_if(0, True)
qc.x(1).c_if(0, True)
```

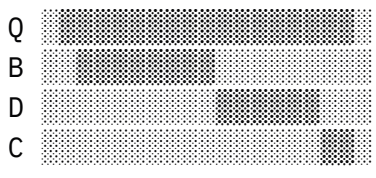
The scheduler SHOULD comply with the above topological ordering policy of the DAG circuit. Accordingly, the asap-scheduled circuit will become



Note that this scheduling might be inefficient in some cases, because the second conditional operation could start without waiting for the 100 dt delay. However, any additional optimization should be done in a different pass, not to break the topological ordering of the original circuit.

Realistic control flow scheduling (respecting microarchitecture)

In the dispersive QND readout scheme, the qubit (Q) is measured by sending a microwave stimulus, followed by a resonator ring-down (depopulation). This microwave signal is recorded in the buffer memory (B) with the hardware kernel, then a discriminated (D) binary value is moved to the classical register (C). A sequence from t0 to t1 of the measure instruction interval could be modeled as follows:



However, the `QuantumCircuit` representation is not accurate enough to represent this model. In the circuit representation, the corresponding `circuit.Qubit` is occupied by the stimulus

corresponding `ClassicalRegister` is occupied by the stimulus microwave signal during the first half of the interval, and the `Clbit` is only occupied at the very end of the interval.

The lack of precision representing the physical model may induce edge cases in the scheduling:

```
        ┌───┐
q_0: ───┤ X ├──────
        └─╥─┘   ┌─┐
q_1: ─────╫─────┤M├
       ┌──╨──┐  └╥┘
c: 1/══╡ c_0=0x1 ╞══╩═
       └─────────┘  0
```

In this example, a user may intend to measure the state of `q_1` after the `XGate` is applied to `q_0`. This is the correct interpretation from the viewpoint of topological node ordering, i.e. The `XGate` node comes in front of the `Measure` node. However, according to the measurement model above, the data in the register is unchanged during the application of the stimulus, so two nodes are simultaneously operated. If one tries to alap-schedule this circuit, it may return following circuit:

```
     ┌────────────────┐   ┌───┐
q_0: ┤ Delay(500[dt]) ├───┤ X ├──────
     └────────────────┘   └─╥─┘   ┌─┐
q_1: ───────────────────────╫─────┤M├
                         ┌──╨──┐  └╥┘
c: 1/════════════════════╡ c_0=0x1 ╞══╩═
                         └─────────┘  0
```

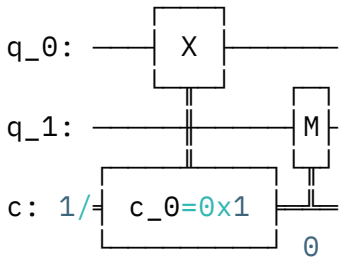
Note that there is no delay on the `q_1` wire, and the measure instruction immediately starts after t=0, while the conditional gate starts after the delay. It looks like the topological ordering between the nodes is flipped in the scheduled view. This behavior can be understood by considering the control flow model described above,

```
: Quantum Circuit, first-measure
0 ░░░░░░░░░░░░▒▒▒▒▒▒░
1 ░▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒░

: In wire q0
Q ░░░░░░░░░░░░░░░▒▒▒░
C ░░░░░░░░░░░░▒▒░░░░░

: In wire q1
Q ░▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒░
```

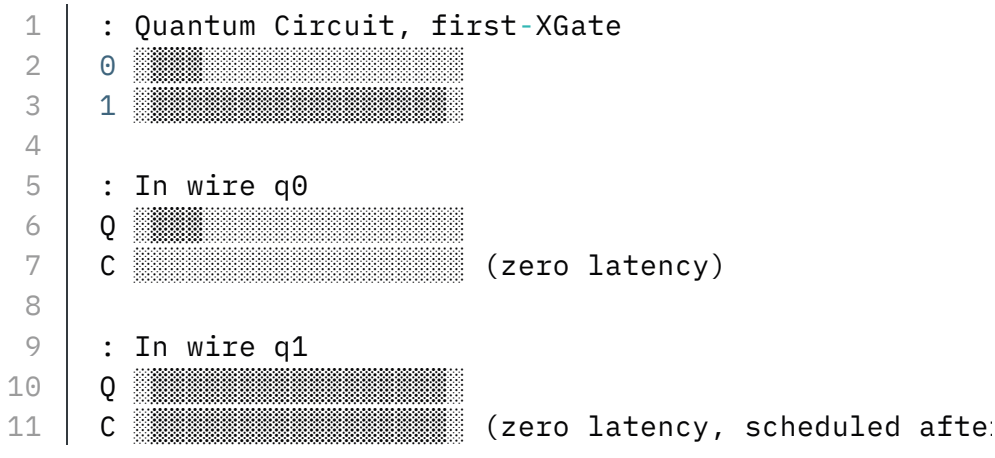
Since there is no qubit register overlap between Q0 and Q1, the node ordering is determined by the shared classical register C. As you can see, the execution order is still preserved on C, i.e. read C then apply `XGate`, finally store the measured outcome in C. But because `DAGOpNode` cannot define different durations for the associated registers, the time ordering of the two nodes is inverted.

This behavior can be controlled by `clbit_write_latency` and `conditional_latency`. `clbit_write_latency` determines the delay of the register write-access from the beginning of the measure instruction (t0), while `conditional_latency` determines the delay of conditional gate operations with respect to t0, which is determined by the register read-access. This information is accessible in the backend configuration and should be copied to the pass manager property set before the pass is called.

Due to default latencies, the alap-scheduled circuit of above example may become



If the backend microarchitecture supports smart scheduling of the control flow instructions, such as separately scheduling qubits and classical registers, the insertion of the delay yields an unnecessarily longer total execution time.



However, this result is much more intuitive in the topological ordering view. If a finite conditional latency value is provided, for example, 30 dt, the circuit is scheduled as follows:

```
                ┌──────────────────┐ ┌─┐
q_0: ─┤ Delay(30[dt]) ├─┤ X ├──────
      ├──────────────────┤ └─┘ ┌─┐
q_1: ─┤ Delay(30[dt]) ├──────┤M├─
      └──────────────────┘      └╥┘
                   ┌──────────┐  ╥
c: 1/═══════════════╡ c_0=0x1 ╞══╩═
                    └──────────┘  0
```

with the timing model:

```
: Quantum Circuit, first-xgate
0 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
1 ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

: In wire q0
Q ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
C ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ (30dt latency)

: In wire q1
Q ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
C ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
```

See https://arxiv.org/abs/2102.01682 for more details.

Transpiler API

Transpiler Target

`Target`([description, num_qubits, dt, ...])	The intent of the `Target` object is to inform Qiskit's compiler about the constraints of a particular backend so the compiler can

	compile an input circuit to something that works and is optimized for a device.
`InstructionProperties`([duration, error, ...])	A representation of the properties of a gate implementation.

Pass Manager Construction

`StagedPassManager`([stages])	A pass manager pipeline built from individual stages.
`PassManager`([passes, max_iteration])	Manager for a set of Passes and their scheduling during transpilation.
`PassManagerConfig`([initial_layout, ...])	Pass Manager Configuration.

Layout and Topology

`Layout`([input_dict])	Two-ways dict to represent a Layout.
`CouplingMap`([couplinglist, description])	Directed graph specifying fixed coupling.
`TranspileLayout`(initial_layout, ...[, ...])	Layout attributes for the output circuit from transpiler.

Scheduling

`InstructionDurations`([instruction_durations, dt])	Helper class to provide durations of instructions for scheduling.

Abstract Passes

`TransformationPass`(*args, **kwargs)	A transformation pass: change DAG, not property set.
`AnalysisPass`(*args, **kwargs)	An analysis pass: change property set, not DAG.

Exceptions

TranspilerError

exception `qiskit.transpiler.TranspilerError(*message)`

GitHub

Exceptions raised during transpilation.

Set the error message.

TranspilerAccessError

exception `qiskit.transpiler.TranspilerAccessError(*message)`

GitHub

GitHub

DEPRECATED: Exception of access error in the transpiler passes.

Set the error message.

CouplingError

exception `qiskit.transpiler.CouplingError(*msg)`

GitHub

Base class for errors raised by the coupling graph object.

Set the error message.

LayoutError

exception `qiskit.transpiler.LayoutError(*msg)`

GitHub

Errors raised by the layout object.

Set the error message.

CircuitTooWideForTarget

exception `qiskit.transpiler.CircuitTooWideForTarget(*message)`

GitHub

Error raised if the circuit is too wide for the target.

Set the error message.

InvalidLayoutError

exception `qiskit.transpiler.InvalidLayoutError(*message)`

GitHub

Error raised when a user provided layout is invalid.

Set the error message.

Yes No

Report a bug or request content on GitHub .