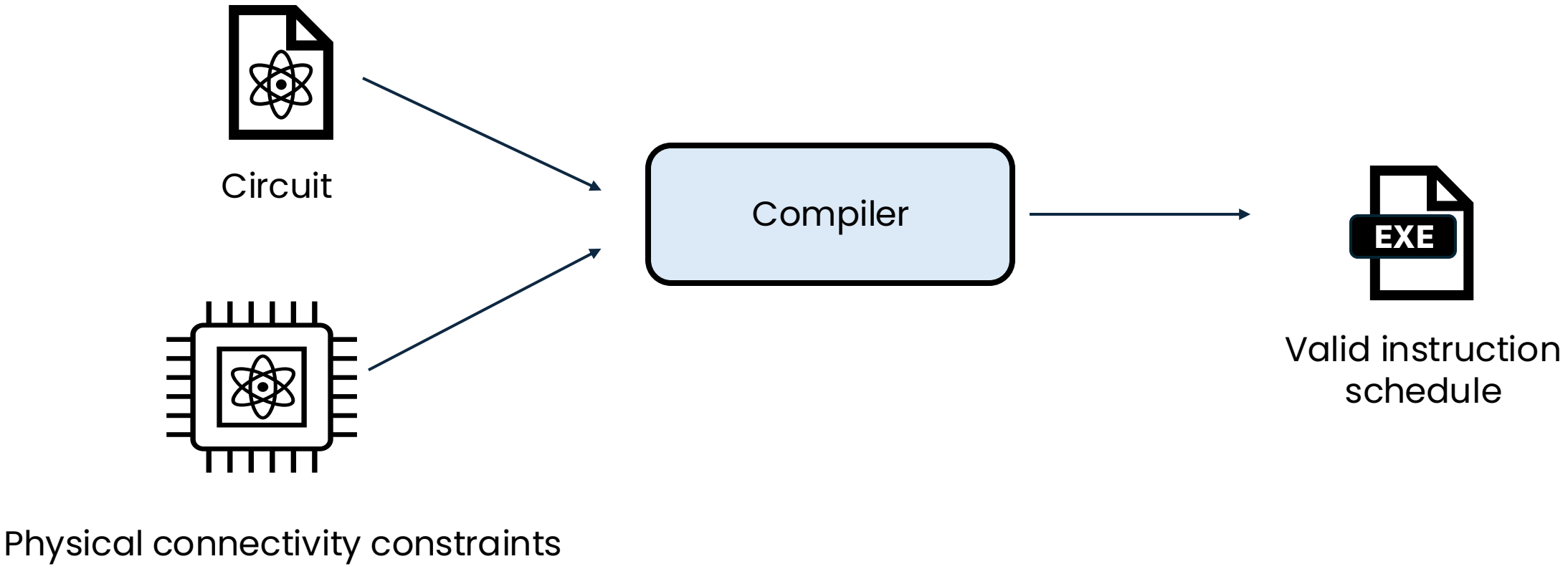
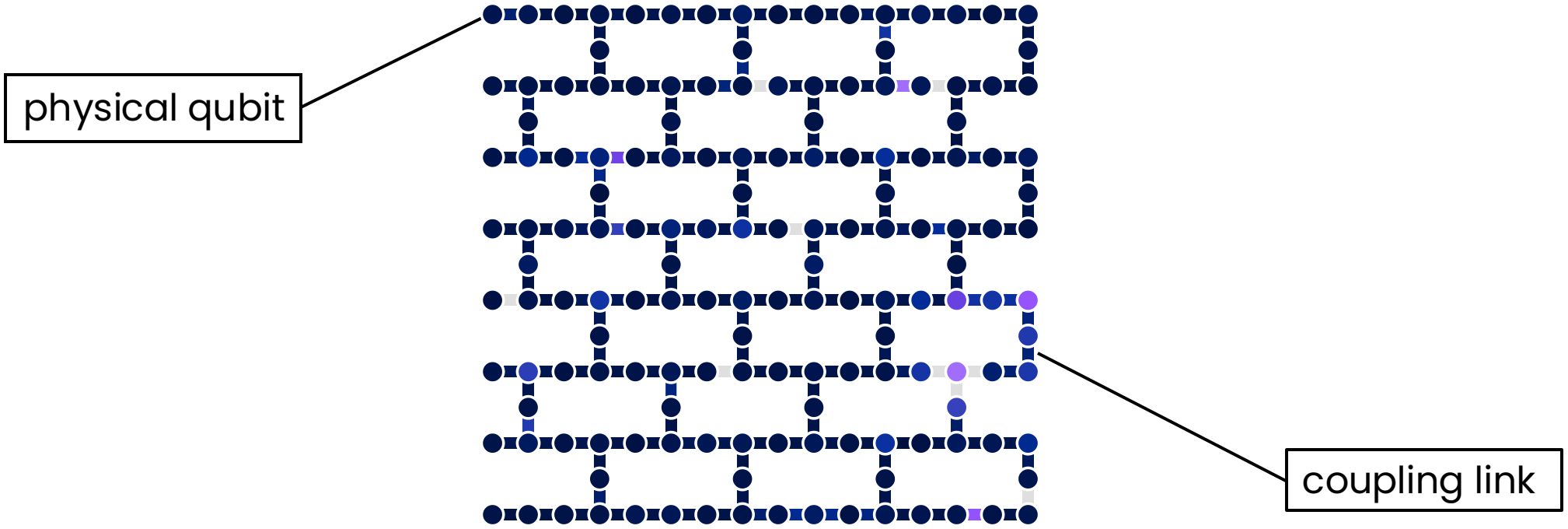
Qubit Mapping and Routing

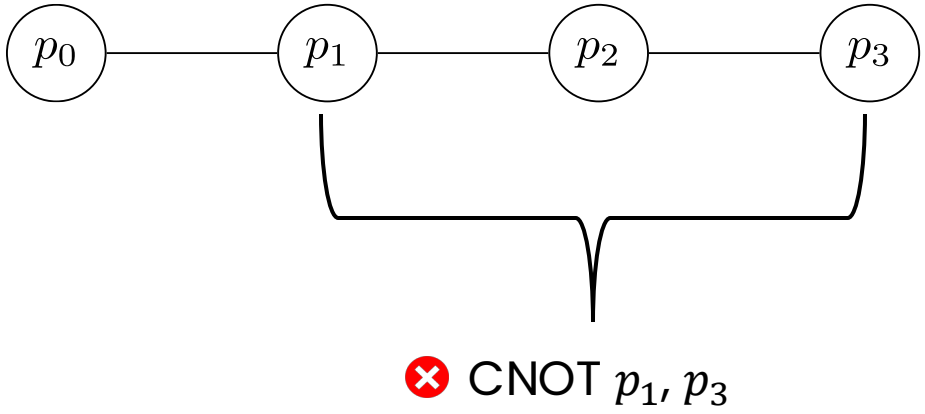

Circuit
Compiler
EXE
Valid instruction schedule
Physical connectivity constraints

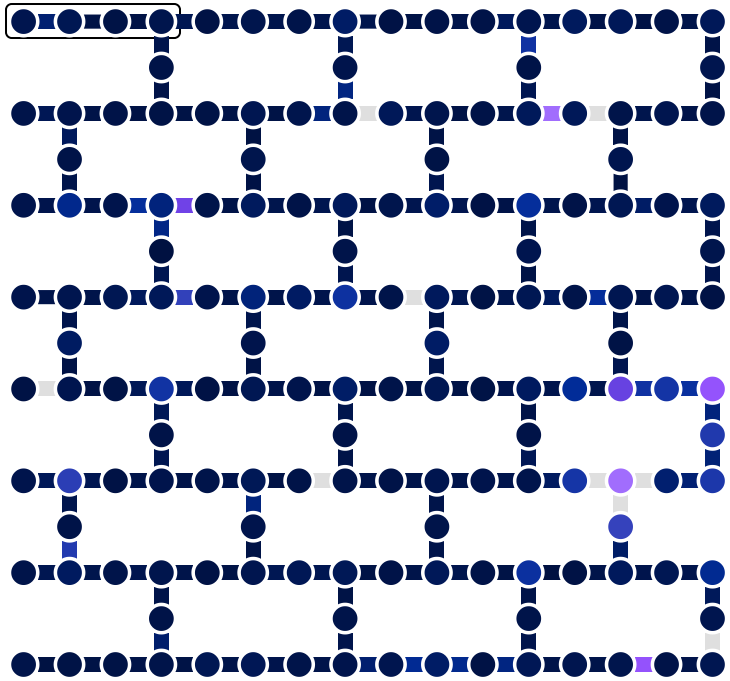
NISQ Qubit Mapping and Routing

Connectivity is limited

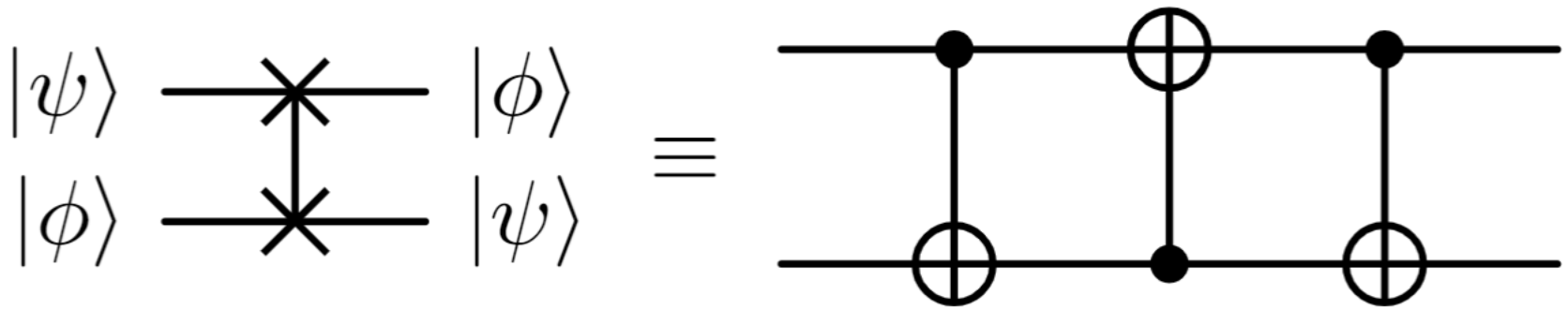


Connectivity is limited

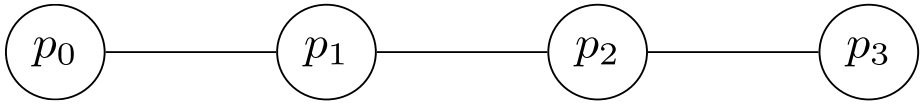




The SWAP gate



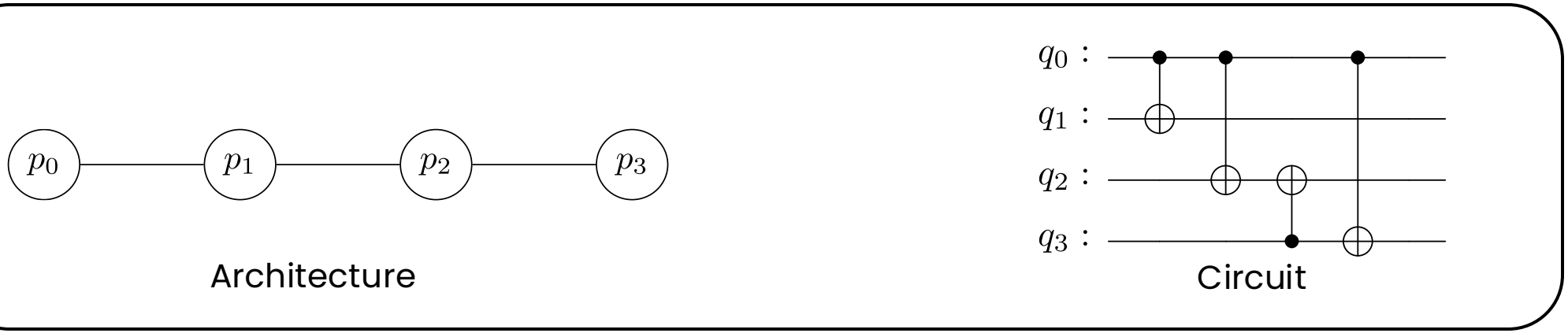
Routing with added SWAPs

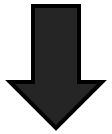


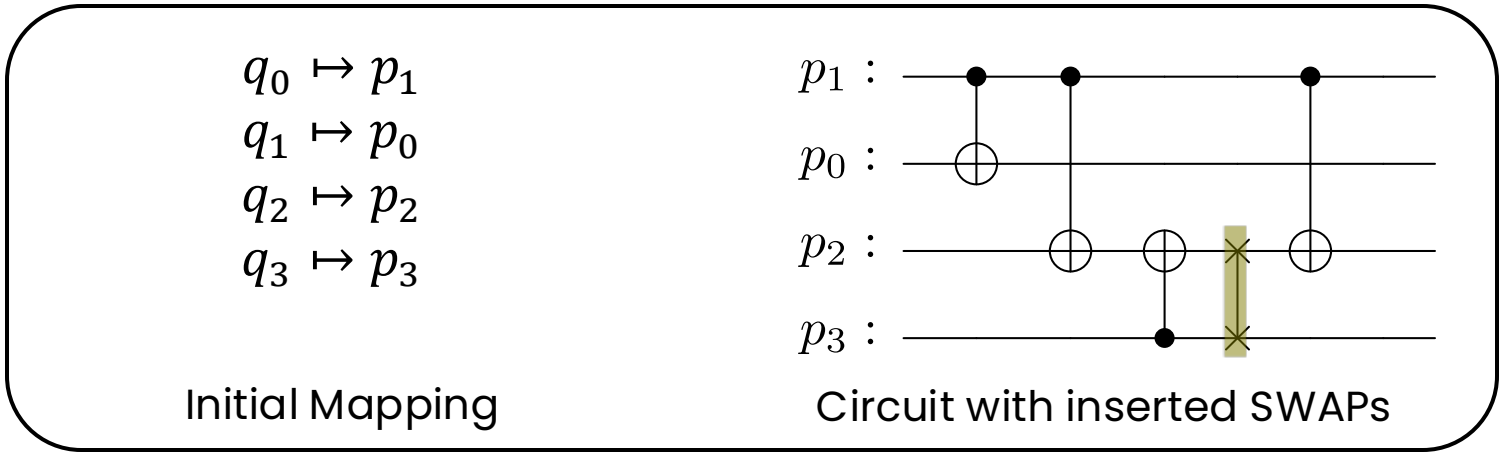
CNOT p_1, p_3

SWAP p_2, p_3
CNOT p_1, p_2

NISQ Qubit Mapping and Routing







Extra two-qubit gates are costly!

ibm_**marrakesh**

QPU status	Online
Processor type	Heron r2

Qubits	2Q error (best/layered)	CLOPS
156	1.23e-3/3.88e-3	195K

Willow System Metrics	
Number of qubits	105
Average connectivity	3.47 (4-way typical)
Quantum Error Correction (Chip 1)	
Single-qubit gate error[1] (mean, simultaneous)	0.035% ± 0.029%
Two-qubit gate error[1] (mean, simultaneous)	0.33% ± 0.18% (CZ)
Measurement error (mean, simultaneous)	0.77% ± 0.21% (repetitive, measure qubits)
Reset options	Multi-level reset (|1⟩ state and above) Leakage removal (|2⟩ state only)
T_1 time (mean)	68 µs ± 13 µs[2]
Error correction cycles per second	909,000 (surface code cycle = 1.1 µs)
Application performance	$\Lambda_{3,5,7} = 2.14 \pm 0.02$

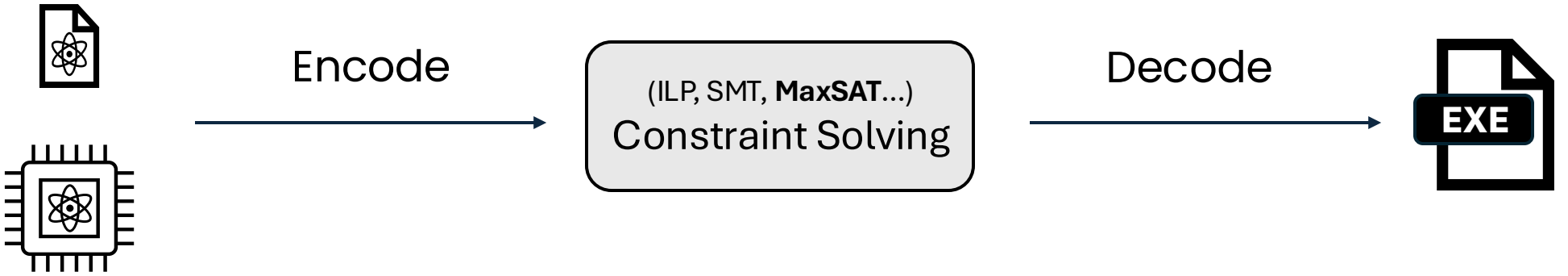
A spectrum of approaches

Scale to large instances
Solutions can be far from optimal

Optimal or near optimal solutions
Cannot handle large inputs

Constraint-based Approach

Encode QMR as an optimization problem; solve with existing tools

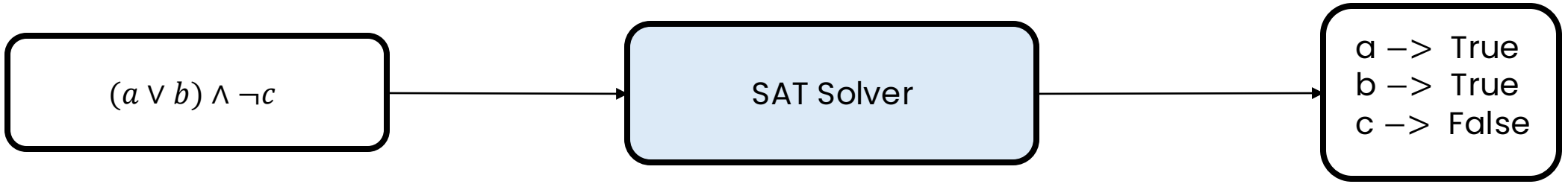


Qubit Mapping and Routing via MaxSAT

Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, Aws Albarghouthi
University of Wisconsin-Madison, Madison, WI, USA
{amolavi, axu44, mdiges, lpick2, stannu, albarghouthi}@wisc.edu

Satisfiability (SAT) solving

Goal: find an assignment of Boolean variables such that the full formula evaluates to *true*



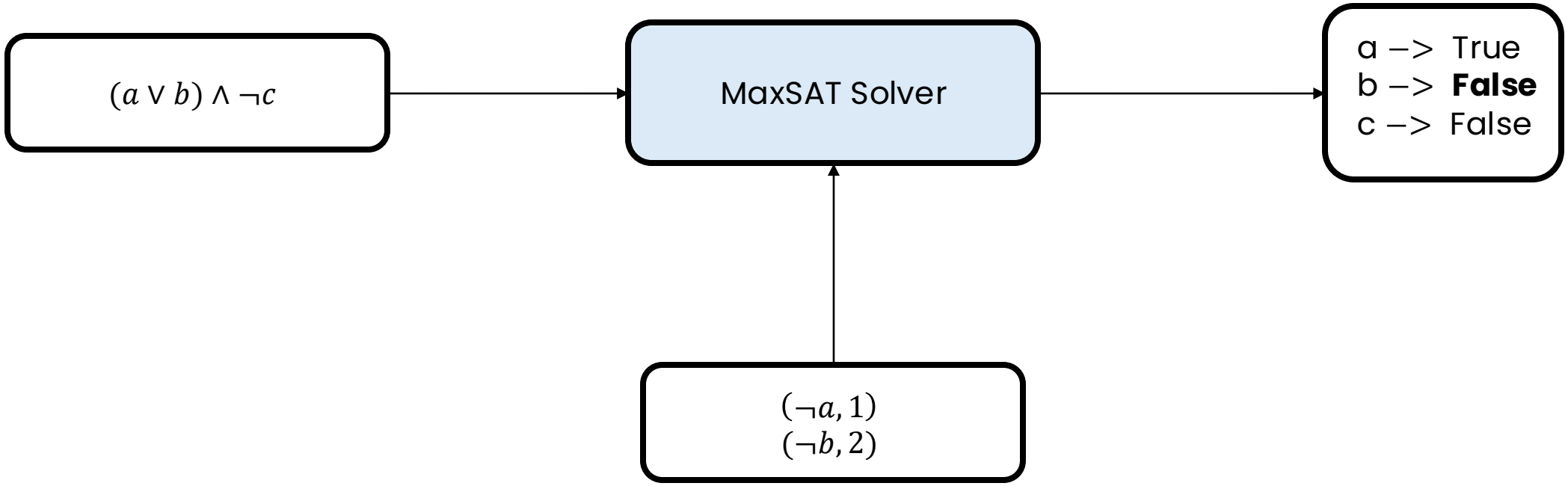
NP-complete, but often tractable in practice!

Many years of solver engineering: https://satcompetition.github.io/

MaxSAT solving

The optimization analogue of satisfiability solving

Express solution cost with soft constraints



Encoding overview

Introduce the following sets of Boolean variables:

- $map(q, p, t)$: Logical qubit q is mapped to physical qubit p at step t
- $swap(p, p', t)$: A SWAP operation is executed on edge (p, p') at step t

Constraints encode

- The maps are injective functions
- Two-qubit gates are executable
- SWAPs transform maps

Pay a cost of 1 for each swap variable set to *true*

Maps are injective functions

For each pair of distinct circuit qubits q and q'

$$map(q, p, t) \rightarrow \neg map(q', p, t)$$

For each pair of distinct physical qubits p and p'

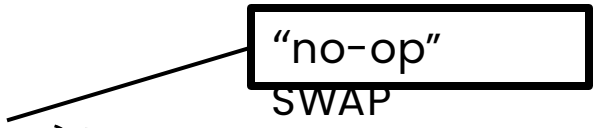
$$map(q, p, t) \rightarrow \neg map(q, p', t)$$

Two qubit gates are executable

For each two-qubit gate $g_k(q, q')$

$$\bigvee_{(p,p') \in Edges} map(q, p, k) \wedge map(q', p', k)$$

SWAPs transform maps



For each step t and swap (p_1, p_2) in $Edges \cup \{(0,0)\}$

$$swap(p_1, p_2, t) \rightarrow (map(q, p, t-1) \leftrightarrow map(q, \pi_e(p), t))$$

where

$$\pi_e(p_1) = p_2 \qquad \pi_e(p_2) = p_1$$

$$\pi_e(p) = p \text{ otherwise}$$

Minimize SWAPs

Soft constraint for each step t and edge (p, p')

$$cost\big(swap(p, p', t)\big) = 1$$

Note: “No-op” swap is free

Challenge: scaling with gate count

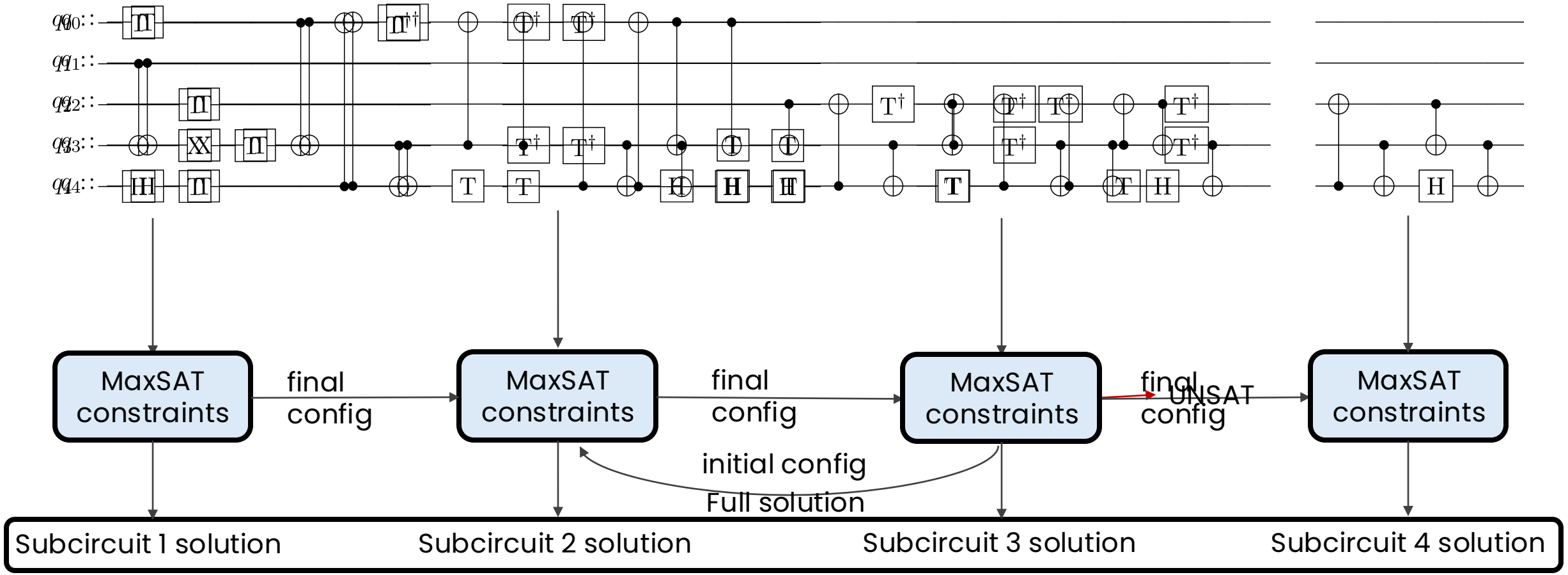
Search space grows exponentially with two-qubit gates count

Global constraint-solving is infeasible for large circuits

SATMAP approach: take a more “local” view

Circuit slicing

Idea: Solve one subproblem at a time and stitch together



Surface Code Mapping and Routing

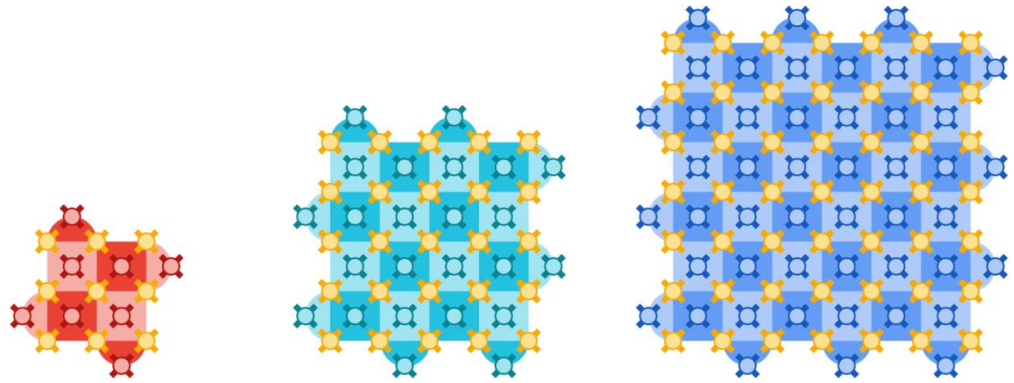
Quantum Error Correction

Encode a logical qubit into several physical qubits

Reduce error by scaling the logical qubit

Prerequisite for exciting applications

- Shor's algorithm,
- Quantum simulation

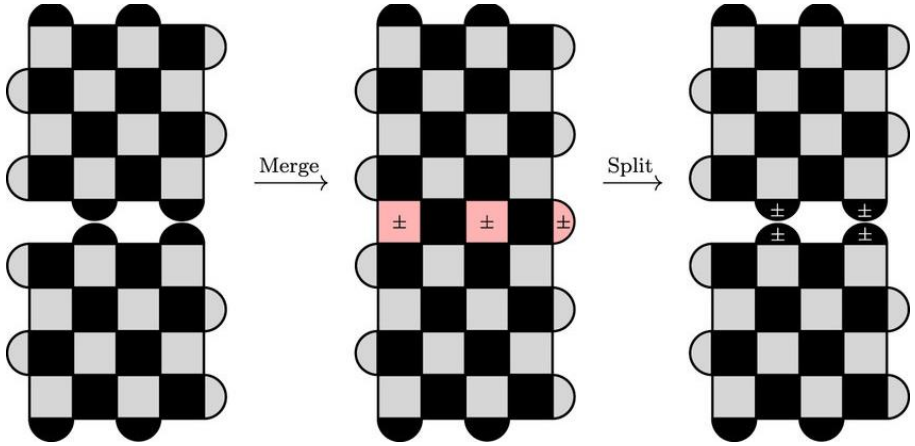


https://research.google/blog/making-quantum-error-correction-work/

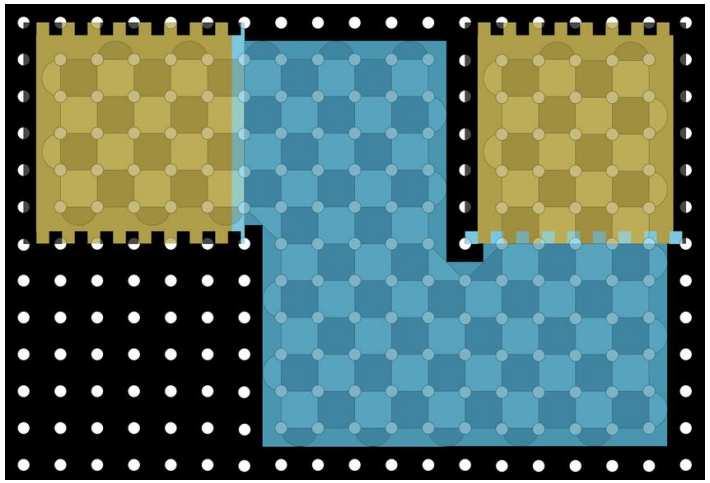
Two-qubit gates via lattice surgery

Lattice surgery: Logical qubits can be *merged* and *split*

Two qubit gates require lattice surgery with an intermediary

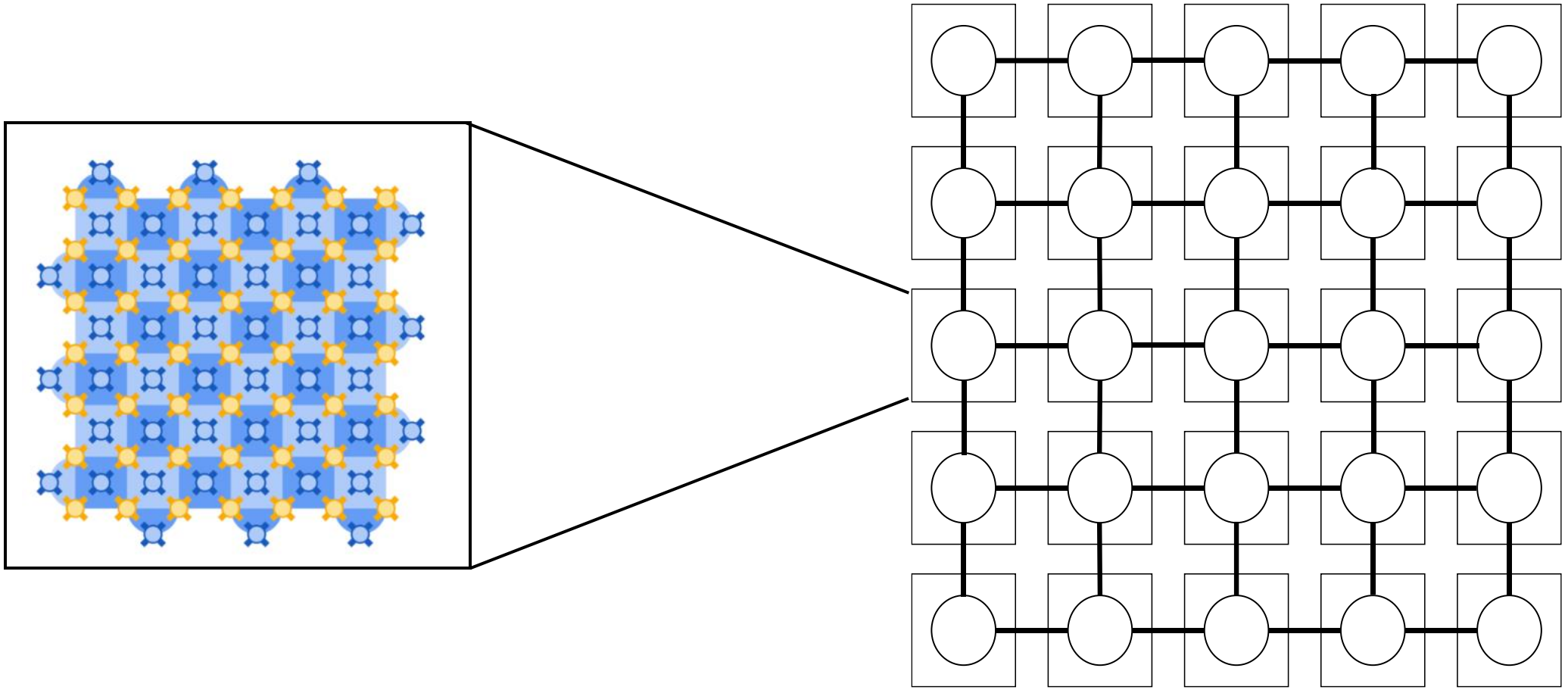


"Code Deformation and Lattice Surgery are Gauge Fixing" Vuillot et al. New J. Phys. 21 (2019)

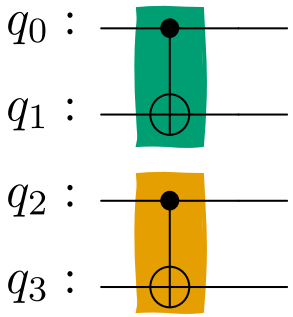


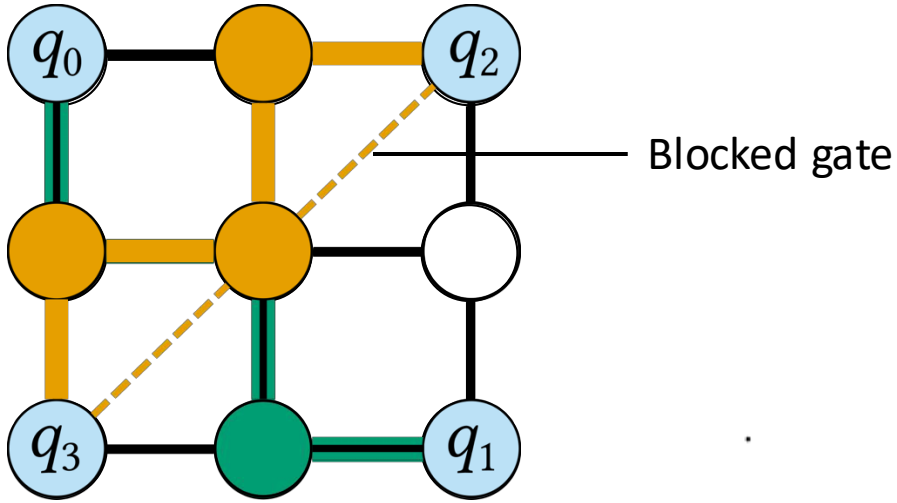
"A High Performance Compiler for Very Large Scale Surface Code Computations" Watkins et al. *Quantum* 8, 1354 (2024).

A graph model

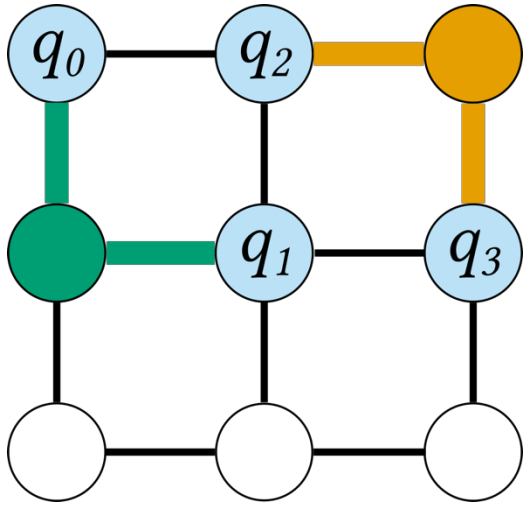


Preserving parallelism





Two step execution



Single step execution

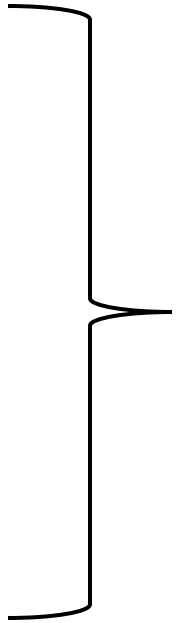
We need to choose our map and gate routes carefully to avoid serializing parallel gates

The DASCOT Approach

Both powered by
simulated annealing

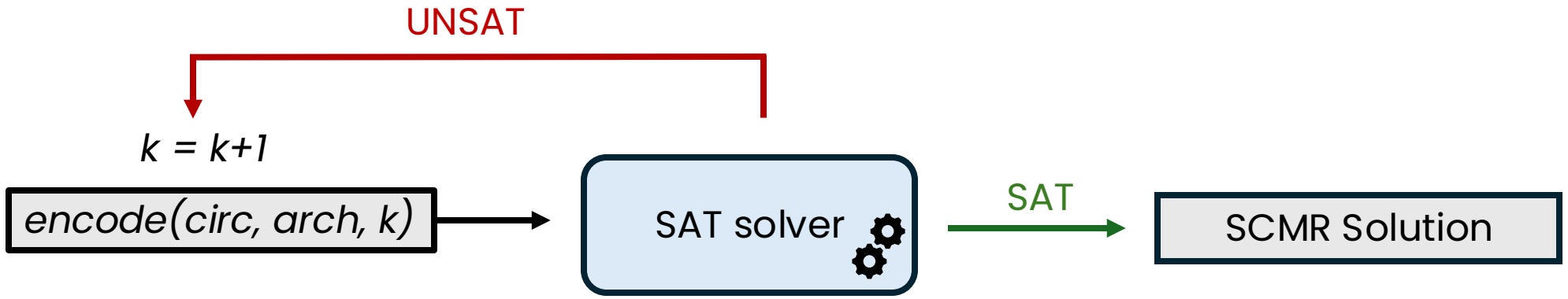
Mapping phase takes dependency-aware view
minimizes conflicts between *parallel* gates

Routing phase searches for the best gate routing order
better than a fixed prioritization heuristic



Optimal SCMR

We have also encoded the SCMR problem into SAT



Many of the same ideas as the NISQ case, with new formulas to represent the no-crossing constraint

Feasible for circuits with tens of gates and qubits

Check out the OOPSLA talk for more!

★ Dependency-Aware Compilation for Surface Code Quantum Architectures

Practical applications of quantum computing depend on fault-tolerant devices with error correction. We study the problem of compiling quantum circuits for quantum computers implementing surface codes. Optimal or near-optimal compilation is critical for both efficiency and correctness. The compilation problem requires (1) *mapping* circuit qubits to the device qubits and (2) *routing* execution paths between interacting qubits. We solve this problem efficiently and near-optimally with a novel algorithm that exploits the *dependency structure* of circuit operations to formulate discrete optimization problems that can be approximated via *simulated annealing*, a classic and simple algorithm. Our extensive evaluation shows that our approach is powerful and flexible for compiling realistic workloads.


Abtin Molavi
University of Wisconsin-Madison


Amanda Xu
University of Wisconsin-Madison


Swamit Tannu
University of Wisconsin-Madison


Aws Albarghouthi
University of Wisconsin-Madison
United States

11:15 on Saturday (18 Oct.) in Orchid Small!

What’s changed?

Mapping: distance doesn’t matter (directly), focus on conflicts

Routing: conflicts between gates mean that order matters

No added gates; execution time is primary objective

A Specification Language for Qubit Mapping and Routing

Based on work conditionally accepted to POPL 26

Generating Compilers for Qubit Mapping and Routing

ABTIN MOLAVI, University of Wisconsin-Madison, USA
AMANDA XU, University of Wisconsin-Madison, USA
ETHAN CECCHETTI, University of Wisconsin-Madison, USA
SWAMIT TANNU, University of Wisconsin-Madison, USA
AWS ALBARGHOUTHI, University of Wisconsin-Madison, USA

Quantum computers promise to solve important problems faster than classical computers, potentially unlocking breakthroughs in materials science, chemistry, and beyond. Optimizing compilers are key to realizing this potential, as they minimize expensive resource usage and limit error rates. A compiler must convert architecture-independent quantum circuits to a form executable on a target *quantum processing unit* (QPU). A critical step is *qubit mapping and routing* (QMR), which finds mappings from circuit qubits to QPU qubits and plans instruction execution while satisfying the QPU's constraints. The challenge is that the landscape of quantum architectures is incredibly diverse and fast-evolving. Given this diversity, hundreds of papers have addressed the QMR problem for different qubit hardware, connectivity constraints, and quantum error correction schemes. For each new set of constraints, researchers develop specialized compilation algorithms.

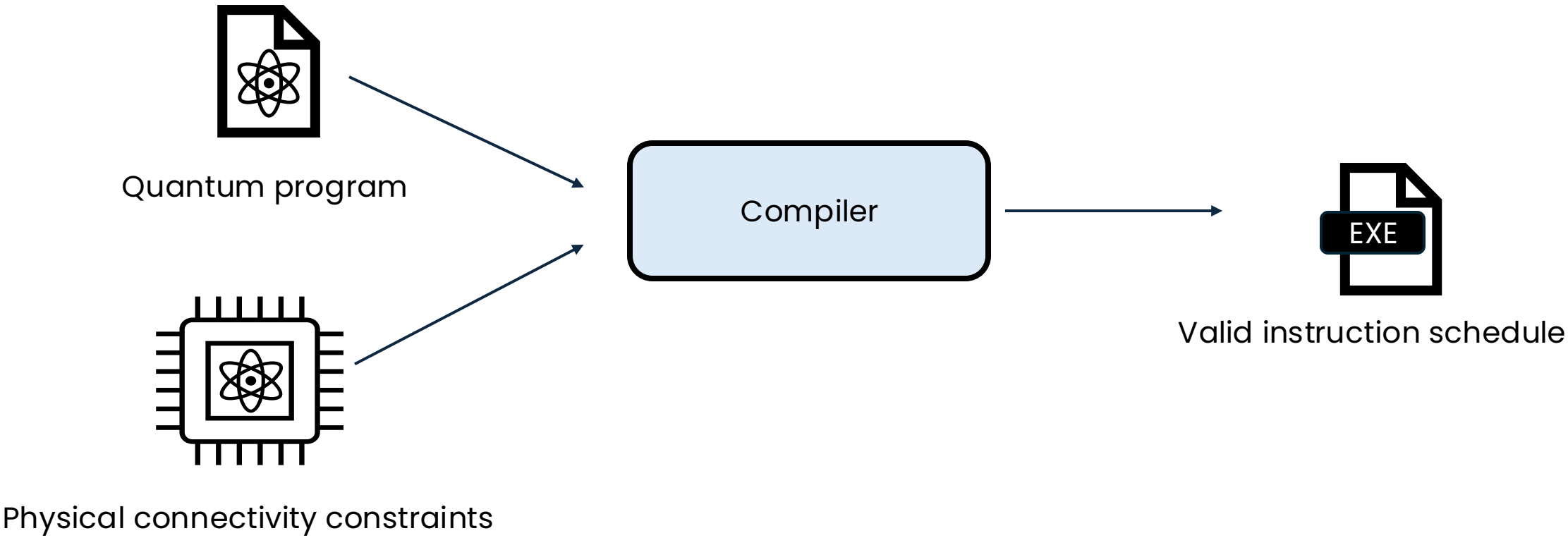
We present an approach for automatically generating qubit mapping and routing compilers for arbitrary quantum architectures. Though each QMR problem is different, we identify a common core structure—*device state machine*—that we use to formulate an *abstract* QMR *problem*. Our formulation naturally leads to a domain-specific language, Marol, for specifying QMR problems—for example, the well-studied NISQ mapping and routing problem requires only 12 lines of Marol. We demonstrate that QMR problems, defined in Marol, can be solved with a powerful parametric solver that can be instantiated for any Marol program. We evaluate our approach through case studies of important QMR problems from prior and recent work, covering noisy and fault-tolerant quantum architectures on all major hardware platforms. Our thorough evaluation shows that generated compilers are competitive with handwritten, specialized compilers in terms of runtime and solution quality. For instance, for the well-studied NISQ mapping and routing problem, we find solutions that match or improve upon the leading industrial toolkit QISKIT on half of the benchmarks. On the newly introduced interleaved logical qubit architecture, we outperform the proposed baseline compilation pipeline in solution quality for 93% of benchmarks. We envision that our approach will simplify development of future quantum compilers as new quantum architectures continue to emerge.

1 Introduction

Quantum computation promises to surpass classical methods in important domains, potentially unlocking breakthroughs in materials science, chemistry, machine learning, and beyond. Quantum computing is at an inflection point: scientists are scaling quantum hardware [11, 20], demonstrating practical quantum error correction protocols [15], and exploring promising application domains [29]. However, to fully realize the potential of quantum hardware available today and on the horizon, we need optimizing quantum circuit compilers. A compiler must convert architecture-independent, circuit-level descriptions of quantum programs to a form executable on a target *quantum processing unit* (QPU). Inefficient compilation that induces significant runtime overhead is unacceptable. For one, access to quantum compute is limited and costly. Further, quantum computation is error-prone, and longer computations are associated with a higher probability of a logical error, even when quantum error-correcting codes are applied.

arxiv draft available now!

Returning to our Abstract Picture



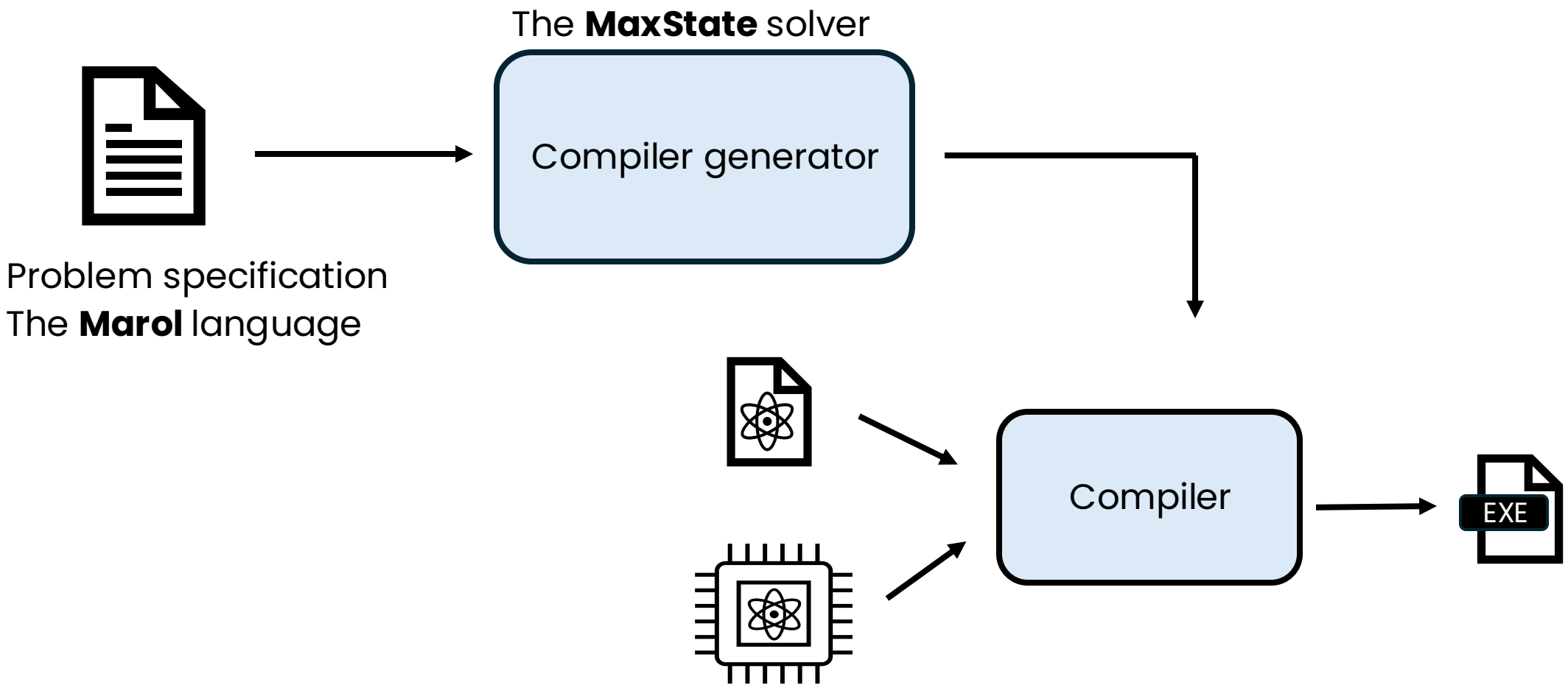
Variations on a theme

We have seen two examples, but there are many more

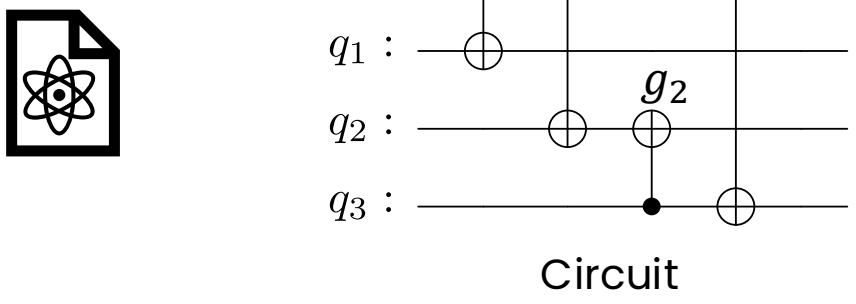
What if some qubits are more reliable that others? [Tannu19]
What if qubits can physically move during execution? [Wang24]
What if we can natively execute gates over >2 qubits? [Silva24]

What if…

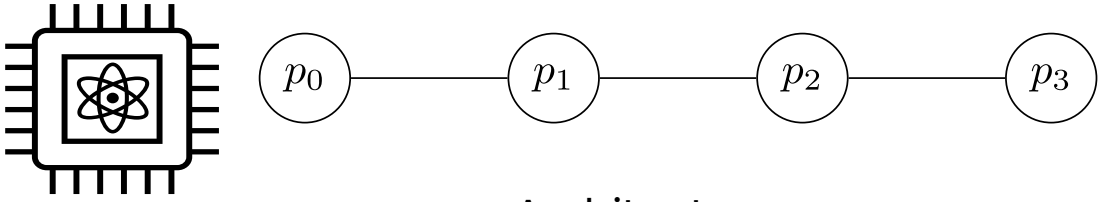
A Compiler Generator for QMR



Another look at a QMR solution



Circuit



Architecture

Step	Gates	Qubit Map
1	$g_0 \mapsto (p_0, p_1)$	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
2	$g_1 \mapsto (p_1, p_2)$	--
3	$g_2 \mapsto (p_3, p_2)$	--
4	$g_3 \mapsto (p_1, p_2)$	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $\boldsymbol{q_2 \mapsto p_3}$ $\boldsymbol{q_3 \mapsto p_2}$

QMR generically

Shared between problems
Data types:
Step, Gate, Arch, Map

Constraints:
Steps respect dependency
Maps are injective functions

Unique to each problem
How do I implement a gate?
How can I transition between steps?

Step	Gates	Qubit Map
1	$g_0 \mapsto (p_0, p_1)$	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
2	$g_1 \mapsto (p_1, p_2)$	--
3	$g_2 \mapsto (p_3, p_2)$	--
4	$g_3 \mapsto (p_1, p_2)$	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $\boldsymbol{q_2 \mapsto p_3}$ $\boldsymbol{q_3 \mapsto p_2}$

```
RouteInfo:
  GateRealization{edge : (Loc,Loc)}
  routed_gates = [CX]
  realize_gate = map(|x| → GateRealization{edge = x},
                     Arch.edges_between(State.map[Gate.qubits[0]], State.map[Gate.qubits[1]]))

TransitionInfo:
  Transition{edge : (Loc,Loc)}
  get_transitions = map(|x| → Transition{edge = x}, Arch.edges())
  apply = value_swap(QubitMap, Trans.edge.(0), Trans.edge.(1))
  cost = if Trans == IdTrans
         then 0.0
         else 1.0
```

To get a solver for a QMR problem, just write a Marol program defining it

What if I'm interested in variability between different gate error rates?

```
RouteInfo:
    ...
TransitionInfo:
  ...
  cost = if Trans = IdTrans
         then 0.0
         else Arch.edge_cost[Trans.edge.(0)][Trans.edge.(1)]
ArchInfo:
  Arch{edge_cost : List[List[Float]]}
StateInfo:
  cost=fold(0,
            |acc,x| → acc+x,
            map(|x| → Arch.edge_cost[State.map[x.qubits[0]]][State.map[x.qubits[1]]],
                State.route))
```

Refine your device model with a few lines, no compiler rewrite needed

Constructing a maximal step

fn maximal_step(Arch, Map, Gates, realize_gate)

    Initialize s with qubit map Map

    for g **in** Gates

        let impls = realize_gate(Arch, Map, Gates)

        if impls is not empty:

            s.add($g \mapsto impls.pop()$)

    return s

Solving our example

Gates	Qubit Map
$g_2 \mapsto (p_3, p_2)$	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$

get_transitions, apply

Transition	Qubit Map
Identity	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
SWAP p_0, p_1	$q_0 \mapsto p_0$ $q_1 \mapsto p_1$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
SWAP p_1, p_2	$q_0 \mapsto p_1$ $q_1 \mapsto p_2$ $q_2 \mapsto p_0$ $q_3 \mapsto p_3$
SWAP p_2, p_3	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_3$ $q_3 \mapsto p_2$

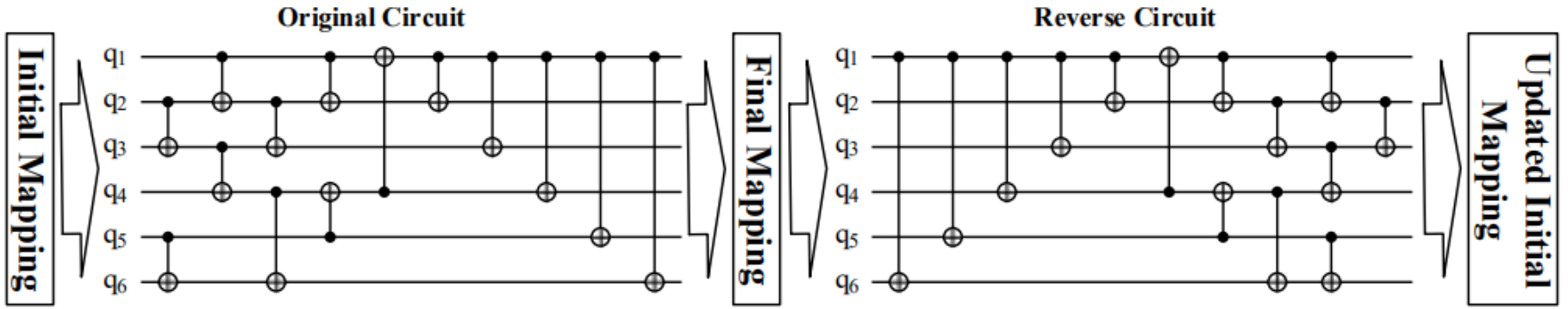
Transition	Qubit Map
Identity	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
SWAP p_0, p_1	$q_0 \mapsto p_0$ $q_1 \mapsto p_1$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
SWAP p_1, p_2	$q_0 \mapsto p_1$ $q_1 \mapsto p_2$ $q_2 \mapsto p_0$ $q_3 \mapsto p_3$
SWAP p_2, p_3	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_3$ $q_3 \mapsto p_2$

maximal_state

Gates	Qubit Map
{}	$q_0 \mapsto p_1$ $q_1 \mapsto p_0$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
{}	$q_0 \mapsto p_0$ $q_1 \mapsto p_1$ $q_2 \mapsto p_2$ $q_3 \mapsto p_3$
{}	$q_0 \mapsto p_1$ $q_1 \mapsto p_2$ $q_2 \mapsto p_0$ $q_3 \mapsto p_3$
$\boldsymbol{g_3 \mapsto (p_1, p_2)}$	$\boldsymbol{q_0 \mapsto p_1}$ $\boldsymbol{q_1 \mapsto p_0}$ $\boldsymbol{q_2 \mapsto p_3}$ $\boldsymbol{q_3 \mapsto p_2}$

Avoiding duplicated work

Implement good ideas like SABRE once and for all!



"Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices"

<u>Other examples</u>
Interaction graph embedding
Criticality-awareness

Isolating problem differences

Formalize the distinctions between problem

Gate blocking is reduced to an automated syntactic check