Quantum Circuit Routing Optimises the Wrong Metric

Closing the Proxy Gap Between SWAP Count and Schedule Length

Dan Cernatinschi[1]

Supervisor(s): S. Feld[1]**, A. Kundu**[1]**, M. Spaan**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Dan Cernatinschi
Final project course: CSE3000 Research Project
Thesis committee: S. Feld, A. Kundu, M. Spaan, A. Lukina

Abstract

To run a quantum program on real hardware, a compiler must rewrite it so that every interacting pair of qubits is physically adjacent on the chip, which it does by inserting extra SWAP operations. How these SWAPs are chosen decides how reliably the program runs: each one adds gates and lengthens the schedule, and on today's noisy devices both effects make a wrong answer more likely. Reliability thus depends on two costs at once, the program's size and its running time, yet the routing pass that quantum compilers deploy, SABRE, optimises only the first: it minimises the number of SWAPs. The SWAP count stands in for size but not for running time, even though running time is the cost that limits reliability most on current hardware. Optimising it alone is a case of a pass tuning a convenient proxy rather than the quantity that ultimately matters.

We show this proxy gap is real and then close it. SABRE-MS keeps SABRE's objective and adds the missing one, the running time, so the router balances the two costs instead of ignoring one. A single tunable weight sets how much each of the two metrics, the program's size and its running time, counts in the balance. It cuts a compiled program's number of cycles by about 20% on average against the SABRE that Qiskit ships, a standard reliability model shows the trade pays off despite the added operations, the gain holds as circuits grow to tens of qubits, and on a real 156-qubit IBM processor it raises the measured circuit fidelity by about 2.5×. The same idea, applied to a very different reinforcement-learning router, helps it in the same way, evidence that the benefit belongs to the objective rather than to SABRE itself.

1 Introduction

A quantum algorithm is written as if any two qubits can interact directly. Real hardware does not allow that: on a physical chip, a two-qubit gate can act only on certain pairs of qubits, the ones wired together on the device, which form its *coupling graph* (Figure 1 shows several; Section 2 defines it). A *compiler* turns the abstract algorithm into an equivalent circuit the hardware can actually run. This requires several transformations; the one this paper studies handles two-qubit gates whose qubits the chip does not connect, and so cannot run as written. The compiler repairs this by inserting *SWAP* gates, each of which exchanges the contents of two connected qubits, moving a qubit one position along the chip, until the two qubits a gate needs sit on a connected pair (Figure 2 shows a small example; Section 2 makes it precise). Choosing those SWAPs is called *routing*, and it is the focus of this paper. A stage just before it, *initial mapping*, chooses where each qubit starts on the chip and so shapes how much routing is needed; we hold it and the other compiler stages fixed and change only routing. Routing matters because every SWAP it inserts adds gates and lengthens the circuit, and on noisy hardware both lower the chance of a correct result.

There are usually many valid ways to insert these SWAPs, and they are not equally good. A compiled circuit is judged on its *success probability*, the chance it returns the right answer, which has two parts: it drops with the number of gates, which each carry an error, and it drops with how long the circuit runs, because a qubit holds its state only for a limited time and decays once that time is exceeded (it *decoheres*). The runtime, the circuit's *makespan*, is the term that matters most on current hardware [9]. The routing pass that Qiskit [5] deploys, SABRE [7], optimises only the first term: it simply inserts as few SWAPs as possible. Fewer SWAPs means fewer gates, so this is a reasonable stand-in for the gate-error part. The SWAP count is correlated with the makespan as well, since fewer gates usually run in less time, but the two are not the same quantity and can diverge: where the SWAPs are placed, not just how many, decides the makespan, so a routing with more SWAPs can finish in a shorter schedule (Section 2 explains why). SABRE optimises the SWAP count alone, so among routings it cannot tell which will run shortest, and leaves the makespan, the dominant cost, to chance.

The makespan is not even a far-off cost the router could be forgiven for missing: it is the explicit objective of a later pass, the scheduling pass that follows routing in the same pipeline. So by minimising SWAP count, routing optimises a stand-in for a cost a downstream pass goes on to minimise directly, and the SWAP-optimal routing can force a longer schedule than a routing SABRE ranked below it would have. We call this mismatch, between the metric an early pass optimises and the cost a later pass is left to pay, the *proxy gap*. It is a general hazard in a multi-stage compiler, and this paper studies it for the routing stage: how large the routing-to-scheduling gap is, and whether scoring routing by the makespan it leads to can close it.

Gate count and makespan are the two costs the router itself controls, so the objective that fits the routing stage is one that weighs both. **SABRE-MS** gives SABRE that objective: it keeps the SWAP-count term and adds a makespan term, so the two costs are balanced rather than one replaced by the other, with a single weight λ setting how the balance is struck. The weight matters because the right balance is not universal: $\lambda = 0$ is exactly SABRE, and the value that shortens the schedule most shifts with the circuit. One weight therefore spans a whole family of routers, from production SABRE at one end to a strongly

makespan-driven router at the other, and the router selects within that family per run by the makespan it reaches. We develop the cost function in Section 4.

Research question.

> **RQ.** Existing routers, including SABRE, the pass Qiskit deploys, minimise inserted SWAP count, a proxy that correlates with the makespan but does not target it. *How large is the resulting loss, can a routing pass that also scores its choices by the downstream makespan recover it without giving up the SWAP-count objective, and does the recovered makespan translate into a higher success probability for the circuit?*

Four sub-questions break this down, each answered in a later section:

SQ1 Is the proxy gap real, and can we close it? Does adding a makespan term to SABRE's score, while keeping its SWAP-count term, cut end-to-end makespan against production SABRE? (Sections 4, 5.1, 5.2)

SQ2 The shorter schedule is bought with extra SWAPs: does that trade still pay off on the end metric, the success probability of the circuit? (Sections 5.3, 5.4)

SQ3 Does the makespan reduction hold at scale, out to tens of qubits? (Section 5.5)

SQ4 Is the makespan benefit specific to SABRE, or to the makespan objective itself? We add the same schedule signal to a structurally unrelated learned (reinforcement-learning) router; if its makespan improves too, the benefit comes from the objective, not from SABRE itself. (Section 5.6)

Contributions.

1. **SABRE-MS**, a modification of SABRE that balances a circuit's two costs, its SWAP count and its makespan, instead of minimising the first alone. It adds a makespan term to SABRE's score and keeps, across trials, the shortest-makespan routing rather than the one with fewest SWAPs, with a single weight λ setting the balance ($\lambda = 0$ recovers SABRE). The router picks λ per circuit by the makespan it reaches, for a fixed constant-factor overhead and the same asymptotic complexity as LightSABRE. It cuts mean makespan by 20.0% on MQT Bench, and a pool decomposition and a 200-trial reachability test show the gain comes from routings SABRE never generates, not from re-picking the ones it does (Sections 4, 5.2).
2. A success-probability analysis charging the extra SWAPs against the shorter schedule: SABRE-MS improves a standard reliability model on the large majority of routing-heavy circuits, with $\lambda = 0$ included as a fallback candidate (Section 5.3).
3. Confirmation on real hardware: on a 156-qubit IBM superconducting device, run on its better-quality qubits, the makespan reduction raises the measured circuit fidelity (the chance the device returns the right answer) by about 2.5× over production SABRE; on poorer qubits, where other error sources dominate, it matches production SABRE, as the reliability model predicts (Section 5.4).
4. Generalisation evidence: the gain holds in a qubit-count sweep from 9 to 49 qubits with paired statistical tests, and the same schedule signal helps a structurally unrelated reinforcement-learning router, evidence that the benefit belongs to the makespan objective rather than to SABRE (Sections 5.5, 5.6).

2 Background

2.1 Qubits, gates, and the connectivity constraint

We start with the little quantum computing this paper needs. A qubit is the quantum unit of information, and a quantum circuit is a sequence of gates on qubits. We work with just two kinds of gate: single-qubit gates and the two-qubit controlled-NOT (CNOT). Together these are universal: any quantum computation can be carried out, to arbitrary accuracy, by single-qubit gates and CNOTs [15]. This is also how real machines run: each device exposes a small native gate set of single-qubit gates plus one two-qubit entangling gate, and the compiler rewrites every circuit into it before execution, after which the rest of the pipeline, including everything in this paper, operates on it [5]. The IBM superconducting processor we run on in Section 5.4 is one such machine; its native two-qubit gate is not CNOT itself but an equivalent one a

few single-qubit gates away, which the compiler handles, so we describe routing in terms of the CNOT throughout.

A quantum circuit is written over *logical* qubits, as if a CNOT could act on any pair of them. On real hardware a CNOT can act only on certain pairs of physical qubits. These pairs form the device's *coupling graph* $G = (V, E)$, where the vertices V are the physical qubits and an edge $(u, v) \in E$ marks a pair the hardware can apply a CNOT to; Figure 1 draws the six we use, which range from sparse chains and rings to denser grids and *heavy-hex*, a sparse hexagonal connectivity pattern used by IBM's current processors [5], on which each qubit connects to at most three others. The compiler must therefore assign each logical qubit to a physical one through a *mapping* $\pi : \{1, \ldots, n\} \to V$, an injection that places the n logical qubits on distinct physical qubits ($n \leq |V|$), and a CNOT on logical qubits (a, b) is then executable only when $(\pi(a), \pi(b)) \in E$. For most circuits no single π places every interacting pair on an edge of G at once, so some CNOTs are left on disconnected qubits and cannot run.

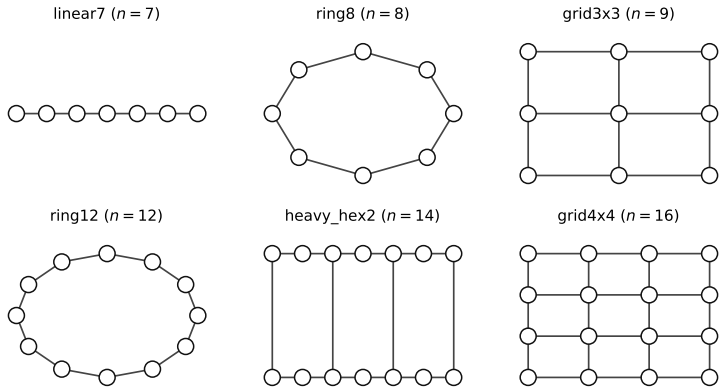

Figure 1: The six coupling graphs the experiments route on (Section 4.3); n is the qubit count, and the heavy-hex patch is the geometry IBM's processors use. Sparser graphs, with their longer distances between qubits, force more routing.

Routing repairs this by inserting SWAP gates. A SWAP on two connected qubits exchanges their states, which swaps the two logical qubits sitting on them and leaves the rest of the computation untouched. By inserting a sequence of SWAPs the router moves the two qubits of a blocked CNOT together until they share an edge, and the CNOT runs. Figure 2 shows why this is sometimes unavoidable: on a star, the only edges are between the hub and each outer qubit, so two outer qubits can interact only after a SWAP moves one of them onto the hub. The repair is not free, though: one SWAP is three CNOTs [15], so every SWAP the router adds is three gates the original circuit never contained.

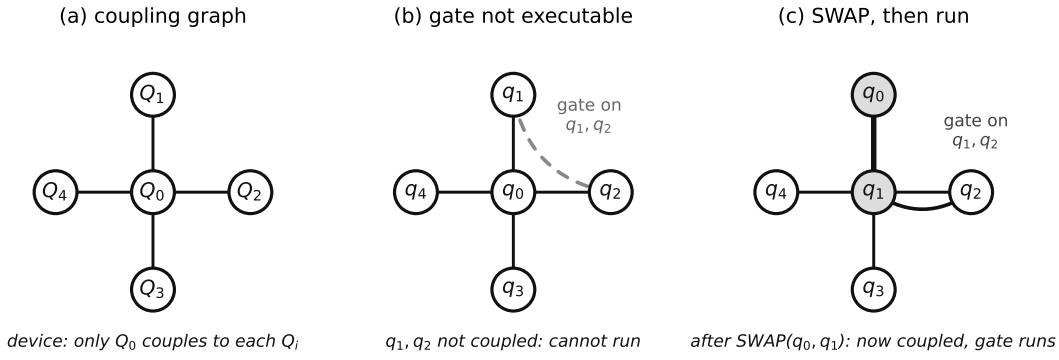

Figure 2: Routing on a five-qubit star. (a) The device connects the hub Q_0 to each outer qubit, and no two outer qubits to each other. (b) With the algorithm's qubits placed $q_i \mapsto Q_i$, a gate is needed on q_1 and q_2, which sit on two outer qubits and are not connected, so it cannot run. (c) Swapping the states of the connected pair (q_0, q_1) moves q_1 onto the hub, next to q_2, after which the gate runs. The SWAP changes which physical qubit carries each logical qubit, leaving the computation the circuit performs unchanged.

2.2 The compilation pipeline and its cost

A compiled circuit is judged on its *success probability*, the chance that a run returns the correct answer. Two hardware limitations pull this down [7]. The first is gate error: every gate can introduce an error, and these errors accumulate, so a circuit with more gates is more likely to fail, which is why a SWAP, three CNOTs, is expensive. The second is decoherence: a qubit holds its state only for a short coherence time, after which it decays, so a circuit that runs longer is more likely to fail. The first cost grows with the number of gates, the second with how long the circuit runs, so a good compiled circuit must use *few gates* and *run quickly*. A standard model combines the two into one success-probability estimate, with a gate-error factor and a decoherence factor [11, 10]; Section 5.3 uses it to weigh our results, and until then we keep the two terms apart, because the routing stage we study acts on each through a different quantity, the SWAP count and the makespan defined next.

The two terms are not equally important on present hardware. End-to-end fidelity studies on current superconducting devices find that circuit *duration* is the single strongest predictor of how reliably a circuit runs, ahead of gate count, because decoherence over the run is the dominant error source [9]. The right measure of that duration is the makespan rather than the circuit depth, the plain count of layers of gates, since gates on a real device take different amounts of time and a layer count weights them all equally [19]. The makespan is therefore not a secondary concern: on the devices in use today it is the term of the success probability that matters most, and it is precisely the term the routing pass leaves out.

Qiskit [5] compiles a circuit through a long sequence of passes. Four of them matter here, and they run in the order of Figure 3:

- *Initial mapping* chooses the physical qubit each logical qubit starts on, that is the mapping π of Section 2.1. Qiskit uses `SabreLayout`.
- *Routing* inserts SWAPs until every CNOT lies on a connected pair. Qiskit uses `SabreSwap`, an implementation of SABRE [7] described in Section 2.3.
- *Gate cancellation* reorders gates through their commutation rules and removes adjacent gates that undo each other [15, 21]. On a routed circuit it can merge part of an inserted SWAP into a neighbouring program gate, so not every SWAP the router adds survives to the final circuit.
- *Scheduling* assigns each gate a start time. Gates on disjoint qubits may share a cycle, while a single qubit runs one gate at a time.

We change only the routing pass and leave the other three as Qiskit ships them.

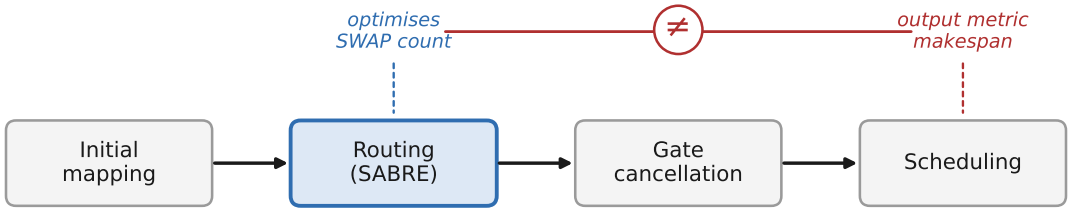

Figure 3: The four compilation passes we consider. The finished circuit is judged on its success probability, which has a gate-error term (it grows with the SWAP count) and a decoherence term (it grows with the makespan, fixed by the last two passes). Routing (SABRE) minimises the SWAP count alone, so it optimises a proxy for the first term and is blind to the second, even though on current hardware the makespan term is the dominant one. The two quantities are correlated but not equal, because cancellation and scheduling weight individual SWAPs unequally (Section 2.4).

The last pass fixes how long the circuit takes to run, so we define it precisely. A *schedule* assigns every gate g a start cycle $\mathrm{start}(g)$; the *scheduler* is the pass that produces one. Each gate has an integer duration $\mathrm{dur}(g)$ in cycles: one for a single-qubit gate, two for a CNOT, and six for a SWAP (Section 4.3 explains these values). A schedule needs a policy for when each gate starts. The standard choice, and the one Qiskit ships as a transpiler pass [5], is as-soon-as-possible (ASAP) scheduling, which starts each gate at the first cycle at which all its qubits are free. We use ASAP throughout and fix it across every experiment. Processing the gates in circuit order, a gate g on the qubit set $Q(g)$ starts once all its qubits are free,

$$\mathrm{start}(g) \;=\; \max_{q \in Q(g)} r_q, \qquad r_q \;\leftarrow\; \mathrm{start}(g) + \mathrm{dur}(g) \quad \text{for } q \in Q(g), \tag{1}$$

where r_q is the next free cycle of qubit q and starts at 0. The *makespan* M is the cycle on which the last gate finishes,

$$M \;=\; \max_{g} \big[\mathrm{start}(g) + \mathrm{dur}(g)\big]. \tag{2}$$

The makespan equals the length of the circuit's *critical path*, the longest chain of gates that must run one after another because each holds a qubit the next one needs. It is the time the circuit occupies the device, and so the running time that decoherence depends on.

These two quantities, the makespan M and the SWAP count, are what a good routing keeps small, and they usually move together: a SWAP is three CNOTs, so inserting fewer of them normally leaves a shorter schedule as well. SABRE optimises the SWAP count alone and relies on this alignment. But the two are not the same quantity, and they can come apart. Two routings with the same SWAP count can have different makespans, and a routing with *more* SWAPs can even produce a significantly shorter circuit, because the makespan is fixed downstream by cancellation and scheduling, which do not weight the inserted SWAPs equally. Section 2.4 explains why.

Initial mapping and routing are both NP-hard [16, 7], so Qiskit solves them with randomised greedy heuristics that break ties at random rather than search for an optimum. The cancellation and scheduling passes that follow are fixed rewrites that depend only on their input, so varying the routing pass under a single fixed scheduler isolates its effect on the makespan; how the choice of scheduling policy itself interacts with routing is a separate question we leave outside our scope.

2.3 The SABRE routing heuristic

Our method modifies SABRE, so we recall the one piece of it we change: its score. SABRE walks the circuit front to back and, whenever the next gates are blocked, inserts the SWAP that most reduces the distance between the qubits those gates need together. Each candidate SWAP s is rated by

$$H(s) \;=\; \tfrac{1}{|F|} \sum_{(a,b)\in F} d(\pi'(a),\pi'(b)) \;+\; w_e \cdot \tfrac{1}{|\mathcal{E}|} \sum_{(a,b)\in \mathcal{E}} d(\pi'(a),\pi'(b)), \tag{3}$$

where d is shortest-path distance in the coupling graph, π' is the mapping s would produce, F is the *front layer*, the set of gates currently blocked and waiting to run, and the second term sums over $\mathcal{E}$, a fixed-size window of upcoming gates, at weight $w_e = 0.5$. SABRE takes the lowest-scoring SWAP and breaks ties at random, so different random seeds can produce different routings. The version Qiskit deploys, LightSABRE [22], runs K seeds and keeps the routing with the fewest SWAPs. So SABRE optimises the SWAP count at two levels: the per-step score in Eq. 3 steers each SWAP by coupling-graph distance, a proxy for how many more SWAPs the routing will need, and the across-seed rule then keeps whichever finished routing has the fewest.

SABRE ships an optional `decay` term, the closest existing mechanism to ours. It raises a qubit's score for a few steps after that qubit takes part in a SWAP, so the router avoids placing consecutive SWAPs on the same qubits; the term resets every few steps and after each two-qubit gate. Its only input is which qubits the last few SWAPs used: it does not see the program's other gates, when a qubit is actually busy, or how long any gate takes. It therefore recovers a little parallelism, lowering circuit *depth* (the count of gate layers), but cannot place a SWAP into a qubit's real idle time as our finish-time term does, and depth in any case ignores gate durations and so tracks the runtime only loosely [19]. It is accordingly far weaker than our term, gaining little makespan over plain SABRE in our experiments and leaving its routings and SWAP counts much as they were (Section 5.2). Decay is best read as a small afterthought, then, rather than a real attempt to balance the schedule against the SWAP count: SABRE's objective stays the SWAP count, and the schedule the routed circuit will run under is the omission our method corrects.

2.4 Why fewer SWAPs is not the same as a shorter circuit

SABRE's target, the SWAP count, treats every inserted SWAP as equally costly. They are not: the two passes after routing, cancellation and scheduling, make a SWAP's cost depend on where it is placed.

Cancellation determines how many of a SWAP's three CNOTs remain. A SWAP placed next to a program CNOT on the same pair of qubits cancels one of its three CNOTs against that neighbour and keeps two, and a program CNOT on each side reduces it to one; a SWAP on a pair the circuit does not use again keeps all three. Scheduling determines how much time the remaining CNOTs add. A SWAP on qubits that no pending gate needs runs next to the rest of the circuit and does not change the critical path. A SWAP on the qubits the next gate needs must finish before that gate starts, and it lengthens the makespan.

Both effects depend on where a SWAP is placed, not on how many SWAPs there are, and this is the source of the disagreement promised in Section 2.2. The count gives every inserted SWAP the same weight,

whereas the makespan, the length of the critical path, ignores the SWAPs that run off it and weights the rest by the delay they add. So the two routings with equal count come apart when their SWAPs fall on different parts of the critical path, and the routing with more SWAPs wins when its extra SWAPs cancel or run in parallel. SABRE sees only the count and cannot tell these cases apart. The rest of the paper measures the gap this opens and then closes it (Section 5.1).

3 Related work

The blind spot we exploit is not specific to SABRE; it runs through the routing literature. Routers are built and judged on the SWAP count, sometimes the post-optimisation gate count or the circuit depth, but not the schedule the circuit finally runs under. SABRE [7] and the LightSABRE [22] that Qiskit ships minimise the SWAP count (Section 2.3). Later work keeps the same target and only searches it harder: ForeSight evaluates many SWAP candidates per step to drive the count down [2], and the learned router AlphaRouter [17], where it names its goal, calls the SWAP count "the most important metric... rather than the gate depth". The makespan is in none of these objectives.

Why it is left out so consistently is rarely stated outright; a few explanations are plausible:

- the SWAP count is taken to be a good enough stand-in, since the two correlate, so optimising it is assumed to give the scheduler a good circuit;
- the makespan is seen as the scheduler's concern, downstream of routing, rather than something the router should target;
- the running time is treated as secondary to gate count for reliability.

Our results argue against all three. The correlation is real but not exact, and a sizeable gap remains: under a fixed, standard scheduler, the choice of routing alone moves the makespan substantially, and the SWAP-optimal routing is routinely not the shortest (Section 5.1); and the makespan, not the SWAP count, is the term that dominates the success probability on current hardware [9].

The closest prior work to ours is NASSC [8], which makes the same starting observation that not all SWAPs cost the same, but acts on the *post-optimisation gate count*: it routes so that more inserted CNOTs cancel against neighbouring gates. That captures the cancellation effect but not the scheduling one, and it still targets a gate count rather than the makespan the scheduler produces.

A second, orthogonal line makes routing and mapping noise-aware, steering gates onto the least error-prone qubits and links using measured per-edge error rates [18, 10, 11]. This attacks the gate-error term of the success probability, whereas the makespan we target attacks the decoherence term, so the two are complementary rather than competing and could be combined.

Learned routers replace the hand-designed heuristic with a policy trained by reinforcement learning, often in a Gymnasium-style environment such as qgym [20], whose routing environment is built to be modular, with the reward left for the user to define. Even so, the rewards chosen are SWAP count or a distance proxy, not the schedule: the omission is a choice, not a limit of the tooling. That same modularity is what lets us, in Section 5.6, drop a schedule-aware reward into such an agent and show the signal helps a router with no SABRE structure, which separates the benefit of the objective from the algorithm.

4 Methodology

SWAP count and makespan come apart because the passes after routing weight SWAPs unequally (Section 2.4). A first sign of this is easy to test on SABRE itself, which already routes a circuit several times and keeps the fewest-SWAPs trial: rank that same pool by makespan instead, and it often picks a different routing (Section 5.1). But this only reorders routings SABRE already produces, so it is a lower bound, not the fix: the shorter-makespan routings mostly lie outside the pool and must be generated, which takes a router whose score weighs the schedule as it routes. We design **SABRE-MS** to be that router. Section 4.1 defines it, Section 4.2 explains why a schedule-aware score generates shorter-makespan routings, and Section 4.3 fixes the benchmark set and protocol shared by every experiment and summarises the experiments that answer each sub-question.

4.1 SABRE-MS

SABRE-MS adds a makespan term to SABRE's score, weighted by λ, so the router balances the makespan against the SWAP count. The term needs the running makespan of the routing so far, which SABRE-MS gets by running the ASAP scheduler of Section 2.2 as it routes, rather than only after.

Concretely, SABRE-MS keeps a vector finish $\in \mathbb{Z}^n_{\geq 0}$, where finish$[q]$ is the cycle at which physical qubit q next becomes free under the ASAP schedule of the gates emitted so far. It starts at zero, and each emitted gate updates it by the ASAP rule of Eq. 1: a gate on (q_1, q_2) starts at $\max(\text{finish}[q_1], \text{finish}[q_2])$, and both entries are set to that start plus the gate's duration (Section 4.3: 2 cycles for a CNOT, 6 for a SWAP). Each update is $O(1)$, and the running makespan is $\max_q \text{finish}[q]$.

SABRE-MS makes two changes to production LightSABRE. First, it adds one term to the per-step score of Eq. 3: for a candidate SWAP s on physical qubits (q_1, q_2),

$$H_{\text{MS}}(s) \;=\; H(s) \;+\; \lambda \cdot \max(\text{finish}[q_1], \text{finish}[q_2]), \tag{4}$$

with a scalar weight $\lambda \geq 0$. The added term is the cycle at which the candidate SWAP would start: it cannot begin until both its qubits are free, which is when the later of the two becomes free, so the term is the max of their two finish times and not a sum. It is large for a SWAP on qubits the program has kept busy and small for a SWAP whose qubits are free now, so SABRE-MS favours SWAPs that slot into idle time, where they are less likely to lengthen the schedule. The original term $H(s)$ stays in the score untouched, so λ sets the balance between the two costs of Section 2.2: it does not replace the SWAP-count objective but adds the makespan objective alongside it. Two consequences follow. At $\lambda = 0$ the score is exactly SABRE's, so the makespan term is off and the routing is the original SABRE's; as λ grows the router buys more makespan at the price of extra SWAPs. The router resolves this trade by the rule below, choosing λ by the makespan it reaches; because $\lambda = 0$ is in its grid and recovers SABRE, a circuit for which no positive λ shortens the makespan simply falls back to original SABRE. The second change is to the selection rule: SABRE-MS keeps LightSABRE's K-trial structure but returns the trial with the shortest post-cancellation makespan rather than the fewest SWAPs. So the first change alters the routing each trial produces, and the second alters which trial is kept; Section 5.2 measures the two separately.

Cost. A single routing trial costs the same asymptotically as LightSABRE: it scores $O(|F|)$ candidate SWAPs per step at $O(1)$ each under LightSABRE's incremental implementation [22], and SABRE-MS's additions are all $O(1)$ per SWAP or per gate (the max-lookup, the finish-time vector with $O(n)$ memory, and a selection rule that reads a makespan already computed downstream), with no data-structure change. The λ-selection sweep adds a further fixed number of trials ($K_0 \,|\Lambda|$, below), independent of circuit size, so the whole router stays in LightSABRE's complexity class and differs only by a constant factor.

Choosing λ. We use a fixed, small grid $\Lambda = \{0, 0.005, 0.01, 0.02, 0.05, 0.10, 0.25\}$, which includes $\lambda = 0$, and for each circuit choose the value whose routing has the shortest makespan. The selection is a short sweep done before the final run: for each $\lambda \in \Lambda$ we route at a small budget of $K_0 = 5$ trials, keep the λ with the shortest makespan, and route once more at the full budget K, which adds the fixed $K_0 \,|\Lambda|$ trials of the cost above. A single λ per topology, fixed from a training set, gives essentially the same results; we select per circuit for simplicity.

4.2 Why it should work: scheduling and cancellation

SABRE-MS does not insert fewer SWAPs; on most configurations it inserts more. Its makespan reduction comes instead from *where* it places them, through the two placement effects of Section 2.4. The first is direct: the finish-time term penalises a SWAP whose qubits are still busy and so favours SWAPs on qubits that are free now, which run in the qubits' current idle time instead of extending the critical path. This is the *scheduling* channel; it is independent of the optimization pass. The second, *absorption*, is inherited from SABRE. SABRE only ever considers SWAPs on a qubit that a waiting gate needs, so the SWAP it inserts to bring a CNOT's qubits together sits on that CNOT's pair, right before the CNOT runs. That is exactly the SWAP-next-to-CNOT pattern cancellation can fuse (Section 2.4). SABRE-MS draws its SWAP candidates from the same set, so it keeps this tendency; whether the finish-time term strengthens or weakens it is a question for measurement, which we leave to Section 5.2, where we split the two channels and correlate the makespan gain with how much of SABRE-MS's SWAPs cancellation absorbs.

How much the term helps likely varies with the circuit and family, which we investigate in Section 5.2.

The same mismatch is not unique to routing. The initial-mapping stage also scores its choices by SWAP count, so it has the same blind spot, and fixing it there would in principle add to the gain (Section 5.1). We scope this paper to the routing stage and leave a makespan-aware mapping to future work.

4.3 Experimental setup

This section fixes the benchmark set, the shared pipeline and statistics, and then the protocol of each experiment, so that Section 5 can report outcomes without redefining how they were obtained.

The benchmark set. We evaluate on two datasets, both routed through the same pipeline.

Both run on the same six coupling graphs (Section 2.1, Figure 1), spanning a range of connectivity, some sparser and some denser: `linear7`, `ring8`, `grid3x3`, `ring12`, `heavy_hex2`, and `grid4x4`, at 7 to 16 qubits. `heavy_hex2` is a 14-qubit patch of the heavy-hex lattice (Section 2.1).

The first is MQT Bench [13], a suite widely used to benchmark quantum-routing methods, which is why we adopt it. We take all 11 of its algorithm families that generate at these qubit counts, not a hand-picked subset: quantum Fourier transforms (`qft`, `qftentangled`), phase estimation (`qpeexact`, `qpeinexact`), amplitude estimation (`ae`), the variational families (`vqe_su2`, `qaoa`), a quantum neural network (`qnn`), and the state-preparation circuits (`ghz`, `graphstate`, `wstate`). Pairing each family with each graph gives 66 configurations. On 46 of them the circuit needs SWAPs; we call these routing-meaningful. The other 20 need almost no routing at these sizes, mostly the shallow families (`ghz`, `qnn`, `wstate`), which place with few or no SWAPs. These near-zero cells act as controls, where the method should do nothing, and it does.

The second is a synthetic dataset of our own. It draws four families from Qiskit's circuit library [5], all standard in earlier routing studies (the quantum Fourier transform; Quantum Volume (QV) [1], a random benchmarking circuit; the hardware-efficient variational eigensolver VQE (EfficientSU2, full entanglement) [6]; and the quantum approximate optimisation algorithm QAOA on random Erdős–Rényi graphs [3]), and adds two families we generate ourselves, `random` and `parallel`, whose interactions are drawn fresh per seed, over those graphs. We add it for two reasons: its generators yield many distinct circuits per configuration, giving the paired tests more samples and stronger statistical power than the single fixed circuit MQT Bench provides; and their flexibility let us study the proxy gap and settle design choices during development.

Three experiments need circuits or graphs the benchmark set does not contain, for reasons specific to each: the scaling study reaches larger sizes, the hardware run is bounded by device time, and the RL router is trained on its own distribution. Each states its circuits in its own protocol below.

Pipeline and cost model. We run the full Qiskit pipeline: basis decomposition into $\{\text{CNOT}, R_X, R_Z, X\}$, initial mapping by SabreLayout, routing by SABRE (or SABRE-MS), gate cancellation, then ASAP scheduling on a duration table of single-qubit = 1 cycle, CNOT = 2, SWAP = 6. These follow a real device: `ibm_marrakesh`, the 156-qubit IBM superconducting processor we also run on in Section 5.4. We set one cycle to half its median two-qubit gate time, so a CNOT spans two cycles, a SWAP is three CNOTs (6 cycles), and a single-qubit gate, the faster operation, takes the single-cycle unit. Makespan is reported in every table unless stated otherwise.

Comparison protocol and statistics. Every SABRE–SABRE-MS comparison runs both methods on the *same* circuits: SABRE at $K = 20$ routing trials, keeping the fewest-SWAPs trial (the production rule), and SABRE-MS at the final budget $K = 20$ after its λ choice (Section 4.1), keeping the shortest-makespan trial. We pair the two methods at the circuit level, and every significance test in the paper is the same one: a one-sided Wilcoxon signed-rank test on the paired per-circuit makespans, reported as a p-value, with 95% confidence intervals from a paired bootstrap where we give them. The Results section therefore states only the p-value or "significant at $p < 0.05$" without restating the test. Headline tables report means over the circuits in each configuration, with the per-configuration sample size stated in each caption.

Reproducibility. The full pipeline builds on Qiskit [5] and qgym [20], with the RL-router ablation trained by PPO [14, 4]. All random seeds are fixed and every experiment, including the exact package versions and the `ibm_marrakesh` hardware job identifiers, is reproducible from the released code.

We now give the protocol of each experiment in turn; the results follow in the matching subsection of Section 5.

Proxy gap (SQ1). As motivated at the start of Section 4, the cheapest first sign that the two costs disagree can be read off SABRE itself. We route each circuit under $K = 60$ SABRE seeds, building a pool of 60 candidate routings, and rank that one pool two ways: by fewest SWAPs (the production rule) and by shortest post-cancellation makespan. A circuit is *misaligned* when the two rules pick different routings out of the pool, and the makespan it recovers is the difference between the two picks. Because both rules choose only among the routings SABRE itself produced, this measures a lower bound on the gap a schedule-aware router could close: it can reorder SABRE's own pool but not generate the shorter routings that lie outside it.

Main comparison and mechanism (SQ1). The main comparison routes every circuit with both SABRE and SABRE-MS under the shared protocol above (both at $K = 20$, paired on the same circuits) and reports the makespan reduction by family. To trace where that reduction comes from, we add two measurements on

the same routed circuits. The *channel split* measures each makespan twice, once on the raw routed circuit and once after gate cancellation: any reduction already present before cancellation is the scheduling channel, and the extra reduction that cancellation adds is the absorption channel. The *absorption rate* then counts, per inserted SWAP, how many of its three CNOTs cancellation removes; we correlate this rate against the makespan reduction across configurations to test whether more absorption goes with a larger gain.

Reachability (SQ1). The main comparison shows SABRE-MS cuts the makespan, but not where that gain comes from: a better ranking of routings SABRE already produces, or routings it never produces at all. We separate the two with a shared pool per circuit: (A) the SABRE pool by fewest SWAPs, (B) the SABRE pool by shortest makespan, (C) the SABRE-MS pool by shortest makespan, so $A \to B$ isolates the selection rule and $B \to C$ the new score. This still leaves an obvious objection: perhaps SABRE-MS wins only because it gets more tries, and plain SABRE would catch up if simply re-run more often. To rule that out, we run SABRE for $K = 200$ seeds, ten times SABRE-MS's budget, take its shortest-makespan routing as the best plain re-running can reach, and compare SABRE-MS at $K = 20$ against it.

Success probability (SQ2). We score reliability with the Expected Success Probability (ESP) [11, 10], a standard reliability proxy in the noise-aware compilation literature, which combines exactly the two terms of Section 2.2. For a routed circuit with N post-cancellation two-qubit gates (each SWAP counting as three CNOTs) and N_1 single-qubit gates, per-CNOT error ϵ, per-single-qubit error ϵ_1, and coherence time T_2 in cycles,

$$\mathrm{ESP} \;=\; (1-\epsilon)^{N}\,(1-\epsilon_1)^{N_1} \cdot \prod_{q} \exp\Big(-\tfrac{t_{\mathrm{idle}}(q)}{T_2}\Big), \tag{5}$$

where $t_{\mathrm{idle}}(q)$ is qubit q's idle time in the schedule. The first two factors charge every surviving gate, so they count SABRE-MS's extra SWAPs in full; the last charges idle decoherence. We evaluate it on the same circuits and selection rule as the main comparison, with parameters measured from `ibm_marrakesh` ($\epsilon = 0.286\%$ per CNOT, 0.03% per single-qubit gate, $T_2 = 81\,\mu$s, that is 2382 cycles at 34 ns each).

Hardware (SQ2). ESP is only a model, so to confirm that the makespan reduction actually buys higher success probability we measure it directly on real hardware. On `ibm_marrakesh` we score each routed circuit by its fidelity under a mirror test [12] (Appendix A gives the full protocol). Two choices keep the test meaningful. We run on an 8-qubit subgraph of low-error qubits, because the makespan we improve enters fidelity through the decoherence term; on noisier qubits gate and readout error dominate and mask the effect we are trying to measure. And we truncate each circuit to its first 15 two-qubit gates, because the mirror test doubles a circuit's depth and a full circuit would run past the device's noise floor. The appendix also reports a noisier-subgraph control.

Scaling (SQ3). We sweep qubit count along two topology families, square grids from 3×3 (9 qubits) to 7×7 (49 qubits) and heavy-hex patches from 14 to 42 qubits, on the three dense families that scale their gate count with qubits (QFT, QAOA, and phase estimation, QPE) at $K = 20$.

Learned router (SQ4). To test whether the makespan-aware objective generalises beyond SABRE, we make a reinforcement-learning router schedule-aware in the same spirit and check whether it improves too. We train a qgym [20] routing agent with MaskablePPO [14, 4] under two conditions that share the policy architecture, training budget, and circuit distribution. The *baseline* agent uses qgym's SwapQuality reward, which rewards executing gates and penalises SWAPs. The *schedule-aware* agent differs only by being made aware of the schedule, exactly as SABRE-MS is: its observation adds the per-qubit finish times, and its reward adds a terminal term proportional to the final makespan. We work at small scale, since MaskablePPO trains poorly as the device and circuits grow: two 5-qubit topologies and `ring8`, with circuits capped at 35 gates.

Summary of experiments. Table 1 lists the experiments that answer each sub-question, the data each runs on, and what each tests, in the order Section 5 reports them.

5 Results

We answer the four sub-questions in turn. Sections 5.1 and 5.2 show the proxy gap is real and that SABRE-MS closes most of it (SQ1); Sections 5.3 and 5.4 charge the extra SWAPs against the shorter schedule in a

Table 1: The experiments, in reporting order. "Set" is the benchmark each runs on: MQT, the synthetic dataset, the scaling sweep, the real device, or the qgym agent (Section 4.3).

SQ	Experiment	Set	What it tests
SQ1	Proxy-gap reselection	MQT, synth.	is the SWAP-count/makespan gap real
SQ1	Main comparison + mechanism	MQT, synth.	does SABRE-MS cut makespan, and how
SQ1	Reachability vs 200-trial SABRE	MQT, synth.	is the gain reachability, not reranking
SQ2	ESP trade	MQT	do the extra SWAPs still pay off
SQ2	Mirror-test fidelity	`ibm_marrakesh`	does the trade show on real hardware
SQ3	Qubit-count sweep	scaling	does the gain hold to 49 qubits
SQ4	Schedule-signal ablation	qgym	is it the objective, not SABRE

standard reliability model and check the trade on a real device (SQ2); Section 5.5 tests scale to 49 qubits (SQ3); and Section 5.6 shows the same schedule signal helps a structurally unrelated RL router, so the benefit is the objective, not SABRE (SQ4).

5.1 The proxy gap is real: a motivating measurement (SQ1)

Before building a method we check that fewer SWAPs and a shorter makespan really are different goals, with the proxy-gap reselection of Section 4.3. It only reorders routings SABRE already produced, so the makespan it recovers is a lower bound on the gap; if even this conservative within-pool test shows one, the SWAP count is an imperfect proxy for the makespan and a makespan-aware objective is worth building.

It does. On the MQT set [13], the two rules disagree on 48% of configurations, and the makespan rule is 3.4% shorter on average ($p = 6.5 \times 10^{-5}$); the randomised synthetic families show the same gap, with the rules disagreeing on 57% of configurations and the makespan rule 7.3% shorter. The gap is structural (Figure 4): it is large on deep, entangling families and near zero on shallow state-preparation circuits, tracking circuit structure rather than qubit count. The same disagreement appears at the initial-mapping stage too, which motivates a makespan-aware mapping in the same spirit; but fixing it there on top of routing adds at most 2.3 percentage points, so we leave it outside this paper's scope and target routing alone. This is the gap the rest of the paper sets out to close.

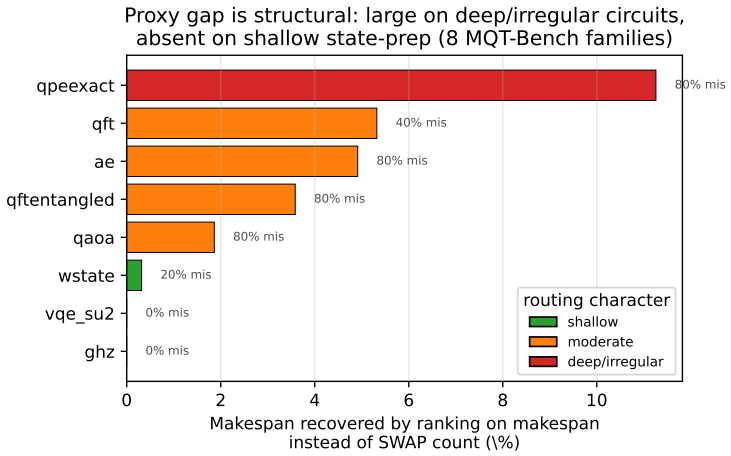

Figure 4: The gap is structural, not a function of size. For each MQT-Bench family, the makespan recovered by ranking SABRE's own trial pool on makespan rather than SWAP count, i.e. the mean of $(M_{\text{swap}} - M_{\text{ms}})/M_{\text{swap}}$ where M_{swap} is the fewest-SWAP pick and M_{ms} the shortest-makespan pick from the same pool; the label on each bar ("% mis") is the share of that family's configurations where the two rules pick different routings. Deep, irregularly entangling circuits (phase estimation) recover the most; shallow state-preparation circuits recover nothing. The `graphstate` family is omitted as not seed-stable (Section 7).

5.2 SABRE-MS and where its gain comes from (SQ1)

Table 2 gives the main comparison's mean makespan reduction for the six routing-heavy families over the six core topologies, with the mean over all 46 routing-meaningful cells in the final row.

Table 2: Mean makespan reduction of SABRE-MS over production SABRE at $K = 20$, on the MQT set, by family over the six core topologies. Values are mean ± standard deviation across the configurations in each family. The six routing-heavy families shown span 36 cells; the mean over all 46 routing-meaningful cells is +20.0%.

MQT family	Configs	Δ makespan (%)
qftentangled	6	$+31.8 \pm 17.0$
qpeinexact	6	$+26.3 \pm 7.5$
qpeexact	6	$+25.3 \pm 6.4$
ae	6	$+26.7 \pm 7.9$
qft	6	$+19.9 \pm 8.3$
qaoa	6	$+13.6 \pm 5.9$
All 46 routing-meaningful		$\mathbf{+20.0 \pm 13.2}$

SABRE-MS reduces mean makespan by 20.0% on the MQT set. The gain is largest on the deep, repeatedly-entangling families and smallest on QAOA, the same ordering the proxy gap followed in Section 5.1. The synthetic dataset shows the same effect: 13.0% mean reduction across 24 configurations, all 24 significant at $p < 0.05$, with QFT at 18.2%, VQE at 17.9%, and QAOA again the least at 9.5%.

The gain splits into the two predicted channels. Section 4.2 predicted two sources for the reduction, a scheduling channel visible before cancellation and an absorption channel added by it, and the channel split of Section 4.3 separates them. SABRE-MS cuts raw makespan by 14.6%, so it is already shorter than production SABRE before cancellation runs ($p = 3 \times 10^{-5}$), and cancellation then takes the total to 20.0%. The two channels do not sum cleanly, because cancellation reshapes the circuit and so changes its schedule, so we read each figure as a floor on its channel rather than an exact split. Measured directly, SABRE-MS's absorption rate is 12.2 percentage points above SABRE's (median), averaged over the eight routing-meaningful core cells, and the per-configuration absorption uplift tracks the makespan gain at Pearson $r = 0.93$ (Figure 5).

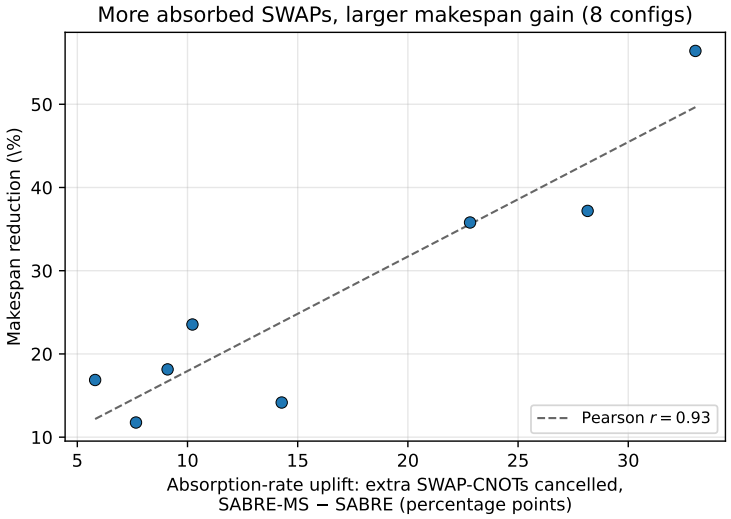

Figure 5: The absorption channel, per configuration. Each point is one configuration at $K = 20$: the horizontal axis is the percentage-point increase in the fraction of SWAP-CNOTs the cancellation pass eliminates (SABRE-MS minus SABRE), the vertical axis is the percentage makespan reduction. The configurations where more of SABRE-MS's SWAPs are absorbed are largely the ones where it shortens the makespan most (Pearson $r = 0.93$).

The comparison is against the strongest SABRE, not just the default. SABRE ships three scoring modes: `basic` (front-layer distance only), `lookahead` (the default, adding the windowed term of Eq. 3), and `decay` (the depth-aware mode, which discourages reusing the same qubits and so is the closest existing trade of SWAP count against depth). Across ten routing-heavy cells, at a single trial each so the comparison is at equal compute, SABRE-MS is shorter than all three: by 35.2% against `basic`, 24.0% against `lookahead`, and 23.0% against `decay` on average, winning on 672 of 800 paired circuits even against `decay`. The depth-aware

mode helps, but it tunes a proxy for the schedule rather than the schedule itself, and the schedule-aware term beats it; the remaining results compare against `lookahead`, the production default.

This gain has two possible sources: the new *score*, which steers SABRE-MS toward routings SABRE would not produce, and the new *selection rule*, which keeps the shortest-makespan trial rather than the fewest-SWAPs one. The decomposition of Section 4.3 separates them, with step $A \to B$ the selection rule alone and $B \to C$ what the new score adds. Table 3 reports the split.

Table 3: Decomposition of SABRE-MS's gain on the MQT set, means over circuits. The six routing-heavy families are shown; the final row averages all 46 routing-meaningful cells. "Rescore" ($A \to B$) is the gain from ranking SABRE's own pool by makespan; "Algorithm" ($B \to C$) is the additional gain from SABRE-MS's score; "Total" is their sum and "Frac." is the rescore share of it.

Family	Rescore	Algorithm	Total	Frac.
qftentangled	+3.0	+28.8	+31.8	10%
qpeinexact	+0.7	+25.6	+26.3	3%
qpeexact	+1.7	+23.6	+25.3	7%
ae	+0.9	+25.9	+26.7	3%
qft	+1.0	+18.9	+19.9	5%
qaoa	+0.3	+13.3	+13.6	2%
All 46 routing configs, mean	+1.0	+19.0	+20.0	**5%**

Almost all of the gain is the new score. Reranking SABRE's own $K = 20$ pool by makespan recovers only 1.0%, a 5% share, while the score adds the other 19.0% (the within-pool gap is larger at the $K = 60$ pool of Section 5.1, but still only a small fraction of what the score reaches). The shorter-makespan routings are not sitting in SABRE's pool ranked below the SWAP-optimal one; they are not in the pool at all, and have to be generated. The synthetic set splits the same way: reranking recovers only a small share of the gain, and the score supplies the rest.

The reachability test of Section 4.3 confirms this. The 200-trial SABRE best barely beats the production best of 20, because the extra trials turn up no substantially shorter routing. SABRE-MS, at only $K = 20$, beats the 200-trial SABRE best on 40 of the 46 MQT configurations, by 17.5% on average, and on all 24 synthetic configurations by 16.2%. The gap is therefore one of reachability: the shorter-makespan routings lie outside SABRE's reach at any seed count, and only a schedule-aware score brings them into it. This is the central result of the paper.

The reason is in SABRE's search rule, not in how many times it is run. At each blocked step SABRE inserts the SWAP that most reduces front-layer distance, so its trajectory is a greedy descent on distance. The seeds only break ties between equally-distance-reducing SWAPs; they do not change the direction of descent. A shorter-makespan routing often needs a SWAP that is *not* distance-optimal, one that leaves a busy qubit free and accepts a slightly longer path, so it sits off the greedy trajectory and no number of seeds reaches it. This is why reranking SABRE's own pool recovers almost nothing while the schedule-aware score, which changes the descent direction itself, recovers the rest: the makespan term makes SABRE-MS willing to take the distance-suboptimal SWAP when it shortens the schedule.

5.3 Does it improve success probability? (SQ2)

The makespan is the term of the success probability that SABRE was ignoring, but it is not the only term, so a shorter schedule is worth having only if it is not paid for with too many extra gates. SABRE-MS buys its shorter makespan by inserting more SWAPs: on 26 of the 46 routing configurations it leaves more two-qubit gates than production SABRE. Every extra two-qubit gate adds error, so a shorter schedule built from more gates is not automatically more reliable. We test the trade with the ESP metric of Section 4.3 (Eq. 5), which charges SABRE-MS's extra SWAPs in full through its gate-error factor and credits the shorter schedule through its decoherence factor. We evaluate it on the same 46 routing-meaningful cells as the main comparison.

The trade pays off (Table 4). Over the 46 routing-meaningful configurations SABRE-MS improves ESP on 36, with an overall median ratio of 1.22×. Restricting to the routing-heavy cells, those with the most two-qubit gates, where ESP best separates the two methods, the median rises to 1.33×, with a worst case of 0.99×. Where SABRE-MS cuts gate count it wins on all 13 such cells; where it adds gates it still wins on 21 of 26, because the shorter schedule outweighs the added gate error; and the remaining 7 cells, where the gate count is unchanged, contribute the other 2 wins. The losses are cells where the makespan gain was too small to offset the gates the shorter schedule cost. Because the idle factor compounds over qubits, the ESP

ratio grows faster than the makespan ratio, and the largest ratios (2 to 6×) fall on the cells with the largest makespan reduction.

Table 4: Expected success probability of SABRE-MS relative to production SABRE on the MQT set, by family over the six core topologies. The ratio is $\mathrm{ESP}_{\mathrm{MS}}/\mathrm{ESP}_{\mathrm{SABRE}}$ (above 1 favours SABRE-MS), under the `ibm_marrakesh` parameters of Section 4.3; "Wins" counts cells with ratio > 1. The six routing-heavy families are shown; the final row aggregates all 46 routing-meaningful cells.

MQT family	Cells	Median ESP ratio	Wins
qftentangled	6	1.36×	6/6
qpeinexact	6	1.35×	6/6
qpeexact	6	1.31×	6/6
ae	6	1.78×	6/6
qft	6	1.25×	6/6
qaoa	6	1.24×	5/6
All 46 routing-meaningful		**1.22×**	**36/46**

ESP is an optimistic proxy: it omits readout error, crosstalk, and leakage, and charges idle decoherence over the whole makespan. It also depends on the device: SABRE-MS acts only through the decoherence factor of Eq. 5, so its benefit is large on qubits where decoherence limits the run and small where gate and readout error do. The hardware measurement of Section 5.4 bears this out on a real device.

5.4 Checking the trade on a 156-qubit device (SQ2)

ESP is a model. The question it leaves open is whether the makespan reduction actually shows up as higher fidelity on real hardware, so we run SABRE and SABRE-MS on `ibm_marrakesh` (Section 4.3) and measure the fidelity of each routed circuit with a mirror test [12], the standard scalable way to score circuit fidelity on real hardware (Appendix A gives the protocol).

We run on an 8-qubit subgraph of the device's lowest-error qubits, and the reason is exactly the point of the experiment, not a way to flatter it. SABRE-MS improves only the decoherence part of fidelity; where gate and readout error are large they dominate the outcome and bury that part, so the makespan reduction can only be seen once those error sources are small enough to leave decoherence as the limiting factor. A low-error subgraph is where that holds. On it, SABRE-MS beats production SABRE on both families we could resolve, reaching 1.7× its fidelity on `qftentangled` and 3.6× on `qft`, a geometric mean of 2.5×.

To show this is not a single lucky subgraph, we also run the same circuits on a noisier one. There SABRE-MS roughly matches SABRE (geometric mean 0.9×), because gate and readout error now dominate and swamp the decoherence term the makespan reduction improves. This conditionality is itself the result, and it is exactly what the ESP model implies (Section 5.3).

5.5 Generalising to 49 qubits (SQ3)

The results so far are at modest sizes. SQ3 asks whether the makespan reduction survives as circuits grow, so we sweep qubit count along grids up to 49 qubits and heavy-hex up to 42 as set out in Section 4.3. Figure 6 plots the mean makespan reduction against qubit count, with bootstrap 95% confidence bands.

The reduction does not decay with scale. On grids it stays near 16 to 21% across the range, 15.9% at 9 qubits and 20.1% at 49. On heavy-hex, the geometry IBM hardware uses, it grows with size, from 12.6% at 14 qubits to 26.4% at 42. All three families gain at the two largest sizes: on the 49-qubit grid QFT gains 18.0%, QAOA 31.7%, and QPE 10.7%; on the 42-qubit heavy-hex QFT gains 23.2%, QAOA 26.9%, and QPE 29.0%. Of the 24 (size, family) configurations, 23 are significant at $p < 0.05$; the one exception is the 49-qubit grid QPE cell ($p = 0.13$, only three test circuits). The gain therefore holds out to 49 qubits, and on the heavy-hex geometry it grows.

5.6 The objective, not SABRE: an RL router (SQ4)

SABRE-MS modifies a specific hand-designed heuristic. To test whether the benefit is tied to that heuristic or to the makespan objective itself, we run the RL-router ablation of Section 4.3: a qgym agent trained under qgym's own SwapQuality reward, and under a schedule-aware reward that adds the same finish-time signal SABRE-MS uses, with everything else held equal.

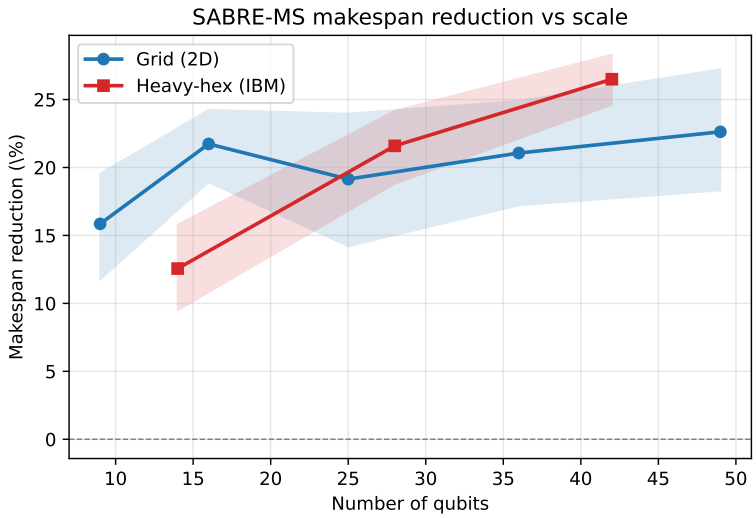

Figure 6: SABRE-MS makespan reduction against production SABRE as a function of qubit count, on grid and heavy-hex topologies (mean over QFT, QAOA, QPE; shaded band is a bootstrap 95% CI). The reduction is stable-to-growing with scale, around 20% on the 49-qubit grid and $\approx 26\%$ on the 42-qubit heavy-hex. We plot makespan only: beyond 16 qubits ESP underflows to near zero for both methods, so it no longer separates them and we keep makespan as the headline metric at scale (ESP is reported up to 16 qubits in Section 5.3).

Table 5: Learned-router ablation at $K = 20$, mean post-cancellation makespan over 6 circuits per cell (lower is better), circuits capped at 35 gates. Both agents share architecture, budget, and training distribution; only the schedule signal differs. Bold marks the schedule-aware agent where it beats the baseline (8 of 9 cells). On `ring8` only QAOA stays within the 35-gate cap; the other families produce larger circuits at eight qubits.

Topology	Family	Baseline RL	Schedule-aware RL	SA gain
linear5	QFT	78.2	**73.0**	+6.6%
	QV	61.3	**61.0**	+0.5%
	VQE	92.3	**86.8**	+6.0%
	QAOA	42.3	**40.8**	+3.5%
ring5	QFT	74.3	**65.0**	+12.6%
	QV	49.7	51.3	−3.4%
	VQE	82.0	**75.7**	+7.7%
	QAOA	37.2	**36.2**	+2.7%
ring8	QAOA	114.7	**76.7**	+33.1%

Table 5 reports the result. The schedule-aware agent gives a shorter makespan than the baseline on eight of the nine (topology, family) cells, by 7.7% on average; the one exception, ring5/QV, is a family with little schedule slack to recover. The gain is largest on `ring8`/QAOA (33.1%), the only cell beyond five qubits: at eight qubits the other families exceed the 35-gate cap we impose to keep training stable, so QAOA is the one family that fits. Adding the schedule signal therefore helps a router with no SABRE structure, which answers SQ4: the benefit comes from the makespan objective, not from the SABRE heuristic.

We read this narrowly. The RL router is a controlled ablation, not a competitive routing method, and the result it supports is just the ablation's: holding architecture, budget, and training distribution fixed, the schedule signal alone shortens the makespan a learned router produces, across both topologies and families. That a router with no SABRE structure benefits from the same signal is the evidence that the gain belongs to the makespan objective rather than to any one algorithm.

6 Discussion

The broader message is about what the field optimises. Routing today is built and judged almost entirely on the SWAP count, because new routers tend to be incremental changes to SABRE and inherit its metric (Section 3). We have shown that this metric, although it correlates with what matters, is not what matters:

the same SWAP count admits routings whose makespans differ substantially, the makespan is the success-probability term that dominates on current hardware, and scoring routing for it directly recovers a large gap that SABRE leaves on the table. The proxy the community has settled on is leaving circuit quality unclaimed.

The shape of our fix is as much the point as its size. SABRE-MS does not discard the SWAP-count objective; it adds the makespan objective beside it and exposes the balance as a single weight, with $\lambda = 0$ recovering production SABRE exactly. One weight therefore spans a whole family of routers, and the right point in that family is not fixed: it depends on the device, the circuit, and how much decoherence rather than gate error limits the run. That makes a tunable balance the natural interface for a moving target. Qubit quality already varies widely across today's devices, and as the hardware and its error handling evolve, so will the worth of the makespan relative to the gate count; a router that can be dialled along that axis, at no cost when the answer is plain SABRE, is better suited to that landscape than one locked to a single proxy. The benefit also carries beyond the specific heuristic: the same schedule signal helps a structurally unrelated learned router, so what we are proposing is an objective for the routing stage, not a single algorithm.

The makespan reduction itself is large, consistent, and holds to the qubit counts we could test; how much of it converts into success probability depends on the device, which is a property of the noise rather than of the method. A few boundaries are worth stating plainly: we isolate the routing pass and hold the rest of the pipeline fixed, so interactions with a schedule-aware mapping or scheduler are out of scope here, and the on-device measurement is a confirmation on one processor rather than a device-scale study. These mark where the work has been tested, not weaknesses in the result, and each is a natural next step rather than a caveat.

7 Responsible research

This work is methodological and computational. It introduces no human-subject data and no personal or sensitive data: the circuits come from the public MQT Bench suite and from synthetic generators, and the hardware runs execute these same circuits on a shared IBM device, producing only measurement statistics.

We have aimed for full reproducibility. Every result in the paper is read from a released result file, and every experiment is deterministic given the fixed seeds we record: the synthetic circuits regenerate from a fixed seed, and the MQT-Bench circuits are a fixed external library call. The one exception is the `graphstate` family, whose within-pool reselection gap is sensitive to Qiskit's internal tie-breaking and so is not fully seed-stable; we therefore omit it from the structural-gap figure (Section 5.1), which measures exactly that gap. It remains in the aggregate makespan and ESP results, where it is one of 46 cells and slightly lowers the mean rather than inflating it. The benchmark set, the cost model (Section 4.3), and the per-run λ-selection rule (Section 4.1) are described in the text; the exact package versions, the random seeds, and the hardware job identifiers are released with the code, so the device runs can be inspected and the full pipeline reproduced.

Reporting integrity required us to discard several results that looked favourable but did not hold up. On the deepest hardware circuits the mirror test pushes both methods to the noise floor, where a fidelity *ratio* between two near-zero distributions can look like a large win ($5\times$ in one family); we judged these artifacts of the floor, not reliability gains, and rest the hardware claim only on the truncated circuits the device can actually resolve (Section 5.4; the floor artifacts are reported in Appendix A). We are likewise explicit that the on-device win is conditional: it is measured on the low-error qubits that day, and we report the result on a higher-error set of qubits alongside it rather than the favourable number alone, since the conditionality is itself the finding the model predicts. Our quantitative claims are also kept to references that are independent of the method being measured; where a comparison risked circularity, we state the limit rather than the flattering number. We see this honest reporting of where results hold and where they do not as part of the contribution, not a footnote to it.

8 Conclusions and future work

A circuit's success probability turns on two costs, its gate count and its makespan, and on current hardware the makespan is the heavier of the two [9]. SABRE minimises a proxy for the gate count, the SWAP count, and does not act on the makespan. Adding one schedule-aware term to the routing score, alongside the SWAP-count term rather than in place of it, closes most of that gap: SABRE-MS cuts mean makespan by 20.0% on MQT Bench and 13.0% on a synthetic set against production SABRE, by reaching routings ordinary SABRE never generates rather than by reranking the ones it does. The single weight λ balances the two costs, and because $\lambda = 0$ is always a candidate in the sweep, SABRE-MS falls back to plain SABRE

on circuits where no positive λ helps, for a fixed constant-factor selection overhead. The reduction holds to 49 qubits, a measured success-probability model shows the extra SWAPs are repaid in the large majority of routing-heavy cases, and on a 156-qubit IBM device the shorter schedule raises the measured fidelity by about 2.5$\times$ on good qubits. That the same signal helps an RL router in the same direction is the evidence that the benefit belongs to the makespan objective, not to SABRE.

Future work. The most direct extension is the initial-mapping stage, which scores its own choices by SWAP count and so has the same blind spot; we already measure a residual gap there, and a makespan-aware mapping on top of makespan-aware routing should compound the gain. More broadly, the proxy gap is not specific to routing: any compiler stage that optimises a local stand-in for a cost a later stage pays is a candidate for the same treatment, so auditing the rest of the quantum pipeline, and perhaps the analogous early-metric versus final-cost mismatches in classical compilers, is a direction in itself. On the objective, a natural next step is to score routing against the success-probability model directly rather than against the makespan as its proxy, folding in per-edge device error rates [10, 11, 18], which are orthogonal to the schedule signal and one additive term away; the case for replacing a single local proxy only grows as hardware constraints become more realistic. Finally, the device evidence can be strengthened with larger and more varied hardware runs, turning the model confirmation into a device-scale result.

References

[1] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, "Validating quantum computers using randomized model circuits," *Phys. Rev. A*, vol. 100, no. 3, p. 032328, 2019, doi: 10.1103/PhysRevA.100.032328.

[2] P. Das, S. K. Vittal, and M. Qureshi, "ForeSight: Reducing SWAPs in NISQ programs via adaptive multi-candidate evaluations," 2022, *arXiv:2204.13142*.

[3] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014, *arXiv:1411.4028*.

[4] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," in *Proc. Int. FLAIRS Conf.*, vol. 35, 2022, doi: 10.32473/flairs.v35i.130584.

[5] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, "Quantum computing with Qiskit," 2024, *arXiv:2405.08810*.

[6] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets," *Nature*, vol. 549, no. 7671, pp. 242–246, 2017, doi: 10.1038/nature23879.

[7] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for NISQ-era quantum devices," in *Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 1001–1014, doi: 10.1145/3297858.3304023.

[8] J. Liu, P. Li, and H. Zhou, "Not all SWAPs have the same cost: A case for optimization-aware qubit routing," in *Proc. IEEE Int. Symp. High-Performance Computer Architecture (HPCA)*, 2022, pp. 709–725, doi: 10.1109/HPCA53966.2022.00058.

[9] R. Malarchick, "End-to-end fidelity analysis of quantum circuit optimization: From gate-level transformations to pulse-level control," 2026, *arXiv:2601.20871*.

[10] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," in *Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 1015–1029, doi: 10.1145/3297858.3304075.

[11] S. Nishio, Y. Pan, T. Satoh, H. Amano, and R. Van Meter, "Extracting success from IBM's 20-qubit machines using error-aware compilation," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 3, pp. 1–25, 2020, doi: 10.1145/3386162.

[12] T. Proctor, S. Seritan, K. Rudinger, E. Nielsen, R. Blume-Kohout, and K. Young, "Scalable randomized benchmarking of quantum computers using mirror circuits," *Phys. Rev. Lett.*, vol. 129, no. 15, p. 150502, 2022, doi: 10.1103/PhysRevLett.129.150502.

[13] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, vol. 7, p. 1062, 2023, doi: 10.22331/q-2023-07-20-1062.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[15] V. V. Shende, I. L. Markov, and S. S. Bullock, "Minimal universal two-qubit controlled-NOT-based circuits," *Phys. Rev. A*, vol. 69, no. 6, p. 062321, 2004, doi: 10.1103/PhysRevA.69.062321.

[16] M. Y. Siraichi, V. F. dos Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Proc. Int. Symp. Code Generation and Optimization (CGO)*, 2018, pp. 113–125, doi: 10.1145/3168822.

[17] W. Tang, Y. Duan, Y. Kharkov, R. Fakoor, E. Kessler, and Y. Shi, "AlphaRouter: Quantum circuit routing with reinforcement learning and tree search," in *Proc. IEEE Int. Conf. Quantum Computing and Engineering (QCE)*, 2024, doi: 10.1109/QCE60285.2024.00112.

[18] S. S. Tannu and M. K. Qureshi, "Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers," in *Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 987–999, doi: 10.1145/3297858.3304007.

[19] M. Tremba, P. Hovland, and J. Liu, "Is circuit depth accurate for comparing quantum circuit runtimes?" 2025, *arXiv:2505.16908*.

[20] S. van der Linde, W. de Kok, T. Bontekoe, and S. Feld, "qgym: A gym for training and benchmarking RL-based quantum compilation," in *Proc. IEEE Int. Conf. Quantum Computing and Engineering (QCE)*, vol. 2, 2023, pp. 26–30, doi: 10.1109/QCE57702.2023.10179.

[21] G. Vidal and C. M. Dawson, "Universal quantum circuit for two-qubit transformations with three controlled-NOT gates," *Phys. Rev. A*, vol. 69, no. 1, p. 010301, 2004, doi: 10.1103/PhysRevA.69.010301.

[22] H. Zou, M. Treinish, K. Hartman, A. Ivrii, and J. Lishman, "LightSABRE: A lightweight and enhanced SABRE algorithm," 2024, *arXiv:2409.08368*.

A Hardware measurement protocol

This appendix gives the full protocol behind the hardware result of Section 5.4.

Mirror test. A mirror test needs no reference output. We route a circuit with each method, append the routed circuit's inverse, and measure; a noiseless run then returns the all-zero bitstring, so any other outcome is device noise. We score each run by the Hellinger fidelity to the ideal all-zero distribution and report, per circuit, the ratio of the two methods' fidelities. Mirror tests are the standard scalable substitute for full state or process tomography on real hardware [12].

Keeping the circuits resolvable. The mirror test doubles a circuit's depth, which sends a full MQT circuit at 8 qubits past the device's noise floor, where a fidelity ratio between two near-zero distributions is meaningless. We therefore truncate each MQT circuit to its first 15 two-qubit gates: this keeps the qubit count and the routing pressure intact while holding the doubled depth below the noise floor (post-routing depth ≤ 700).

Qubit selection. Where a circuit runs governs its fidelity as much as how it is routed, so we run on an 8-qubit subgraph grown greedily from the qubit with the lowest measured readout error on the day we ran. Placing a program on low-error qubits is the standard noise-aware choice [10, 18]; we also report the same circuits on a noisier subgraph as a control. To show the gain is not a single lucky setting, we run every λ in the grid on the device, at $K = 5$ routing trials and 4096 shots, and report the best-fidelity one per cell; every λ beats SABRE on the good subgraph (Section 5.4). This device sweep is a robustness check, separate from the router's own makespan-based choice of λ.

Results in full. On the good subgraph, SABRE-MS beats production SABRE at every λ: the best-λ fidelity ratio is 1.7× on `qftentangled` and 3.6× on `qft` (geometric mean 2.5×). On the noisier subgraph the same circuits sit at a geometric-mean ratio of 0.9×, roughly matching SABRE, because there gate and readout error dominate the decoherence term the makespan reduction improves.

A note on the deep circuits. We also ran three full, untruncated circuits from the synthetic library generators (QFT, VQE, and QV) at 8 qubits, where the doubled-depth mirror test pushes both methods to the noise floor (SABRE fidelities around 0.01–0.05). The per-family ratios there are large (QFT 5.34×, VQE 1.61×, QV 1.19×) and their ordering tracks the simulated makespan gain, but for the reason given in Section 7 we treat them as floor artifacts, not reliability gains, and report them only because the ordering is consistent.