Fast Quantum-Circuit Superoptimization

Aws Albarghouthi
University of Wisconsin–Madison

Abstract

Optimizing quantum circuits is critical: circuits must fit within the resource limits of a quantum computer, and every unnecessary operation increases their cost and probability of failure. We present a simple optimization algorithm for quantum circuits that (1) is very fast, (2) scales to millions of operations, and (3) matches or outperforms the optimization quality of the best existing optimizers *and* superoptimizers.

Our key insight is that we can compactly represent circuit equivalence classes as a map which associates a *projective unitary* with a smallest representative circuit implementing it. We give a sound algorithm for synthesizing such maps, which we call *minimal unitary representative maps* (MURM). Armed with a MURM, our optimizer inspects every bounded subcircuit in a linear pass and replaces it with a minimal variant. Incremental computation and symbolic arithmetic make every replacement fast and sound. Our evaluation shows that our approach achieves significantly greater circuit reductions than leading optimizers, runs orders of magnitude faster, and scales to circuits with millions of operations.

https://github.com/qqq-wisc/tzap

1 Introduction

A critical layer in the quantum computing stack is the quantum circuit optimizer. The quantum circuit optimizer attempts to reduce the size of a quantum circuit so that it fits within the limits of a target quantum device, runs within a reasonable time, and has a low error probability.

Most existing optimization techniques target the comparatively small circuits that run on today's machines. The next frontier, *fault-tolerant quantum computing*, brings a different scale. Important workloads, including simulations of quantum chemistry [8, 26] and Shor's algorithm [37], are expected to require millions or even billions of quantum operations [8]. An optimizer for these *megaquop-scale* circuits must process enormous inputs quickly. Existing techniques fall short in scalability, optimization quality, or both.

Broadly speaking, existing approaches fall into two classes. *Traditional optimizers* follow the classical compiler playbook: they schedule passes for gate cancellation, rotation merging, and similar transformations [4, 5, 18, 21, 29]. These passes apply quickly. Their fixed transformations and orderings restrict the search, favoring speed over optimization quality.

Superoptimizers, on the other hand, search a vast space of equivalent circuits for the smallest circuit reachable within a time budget [40–44]. They can achieve impressive reductions in gate count, often surpassing traditional optimizers. The search is expensive and scales poorly: optimization can take hours even on circuits that are tiny compared with projected fault-tolerant workloads [41, 43, 44].

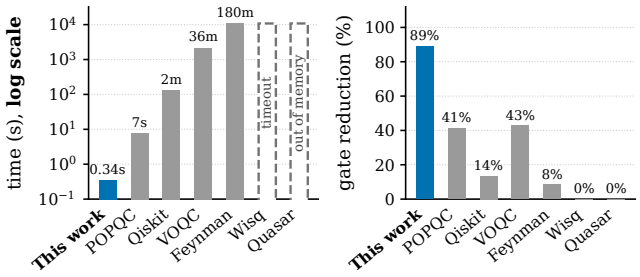


Figure 1. A 143,000-gate Hamiltonian simulation circuit; three hours per tool. This work cuts gates by 89% in 0.34 s—2× the reduction of the best competitor; orders of magnitude faster.

We ask whether we can solve the following *trilemma*:

Can we build a quantum-circuit optimizer that is very fast, scales linearly to megaquop circuits, and matches the optimization quality of superoptimizers?

We answer this question affirmatively. We present a radically simple optimization algorithm that is orders of magnitude faster than existing optimizers, scales to very large circuits in seconds, and matches or improves the gate-count reductions of the best existing optimizers!

Semantic rewriting. We build upon the rewrite-rule view of optimization. A traditional optimizer carries a few dozen rules written by hand, each matching one syntactic pattern: $HH \to \varepsilon$, $TT \to S$, and so on. Such rules are cheap to apply, but they can be myopic. Two subcircuits can do exactly the same thing and still need separate rules, because they differ in gate order, gate choice, or qubit layout, and the number of such variants explodes. Superoptimizers for quantum circuits tend to operate with a large rule set and perform an expensive search through the space of rule orderings at compile time [24, 40, 42, 43].

Our observation is that rewrite rules are a myopic and inefficient mechanism for representing optimizations. A rule is a statement about syntax, while the optimization it encodes is a statement about semantics: one circuit does the same thing as another, and the other one is smaller. Instead, inspired by work in the classical superoptimization literature [9] and circuit synthesis from the early aughts [31, 36], we represent the optimizations *semantically*. Specifically, we compactly

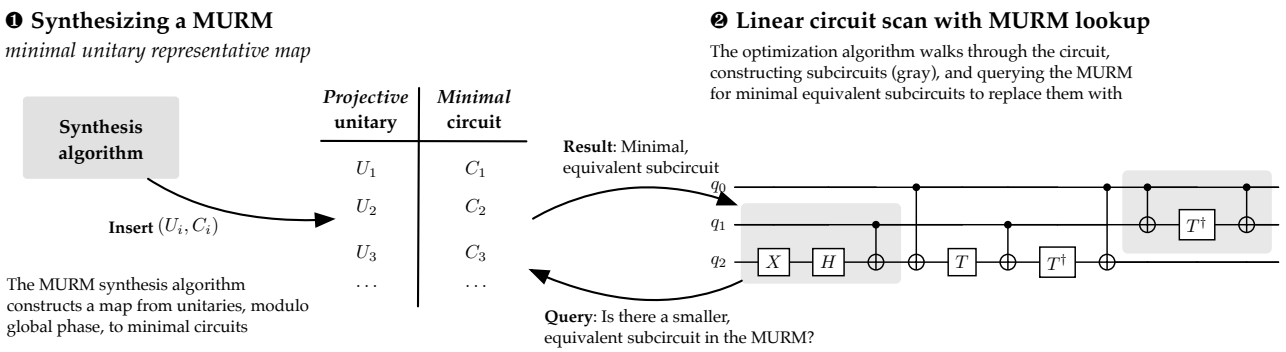


Figure 2. Overview of our quantum-circuit optimization approach

construct equivalence classes of circuits and associate a minimal representative circuit with each class.

We represent equivalence classes using a map that sends a unitary to the smallest circuit implementing it. We call it a *minimal unitary representative map* (MURM), and we give an algorithm that efficiently synthesizes one. A MURM is a remarkably compact way of expressing an enormous number of rewrites. The map our optimizer uses by default holds hundreds of thousands of entries. Each entry stands for a semantic transformation that a conventional optimizer would need *many* handwritten rules for. This practically enables very large one-shot rewrites—e.g., Fig. 3 shows a 25-gate subcircuit, found in a linear-algebra circuit, that a single MURM lookup collapses to five gates in one shot.

We synthesize a MURM by enumerating circuits breadth-first, one gate-count layer at a time. The empty circuit is considered first, followed by all one-gate circuits, all two-gate circuits, and so on. The first circuit to reach a unitary is inserted into the map; because enumeration is ordered by length, that first circuit is a smallest representative of its equivalence class.

Exact unitary matching is critical because the optimizer may perform an enormous number of replacements. With floating-point matrices, equivalent circuits may compare unequal, and tolerance-based comparisons may admit small semantic errors that accumulate across a large circuit. We work with Clifford+T circuits, which allows us to represent every matrix in exact symbolic arithmetic and canonicalize it to identify matrices that differ only by a global phase.

A linear optimization pass. Armed with a MURM, optimizing a circuit amounts to inspecting its subcircuits up to fixed gate and qubit bounds. Our algorithm does so greedily, in one forward pass through the circuit. For every subcircuit encountered, the algorithm computes its exact unitary, queries the MURM, and replaces the subcircuit whenever the MURM contains a smaller representative.

Enumerating subcircuits and querying the MURM can be slow, since a naive implementation builds every subcircuit, and every subcircuit's unitary, from scratch. We avoid this by working incrementally. Consecutive subcircuits usually differ by a single gate, so we extend a subcircuit by appending that gate and updating its unitary in place, rather than rebuilding either. With fixed gate and qubit bounds, the number of subcircuits is linearly bounded by the size of the input circuit, giving us a fast and scalable algorithm.

Evaluation. We implemented the algorithm as part of the TZAP optimizer and evaluated it against leading traditional and search-based optimizers (*superoptimizers*) on standard benchmarks and megaquop-scale workloads. Our approach can significantly exceed their circuit reductions (total gate count, two-qubit-gate count, and T-gate count) and runs orders of magnitude faster.

Fig. 1 shows results for a 143,000-gate Hamiltonian simulation circuit. Every optimizer is given three hours. Our approach removes 89% of its gates in *a third of a second*. The best competing reduction is 43%, and it costs 36 minutes—nearly four orders of magnitude more time for half the reduction.

We additionally formalize our approach in Lean [14] and prove its correctness, establishing that the optimized circuit is equivalent to the input circuit.

Contributions. We make the following contributions:

- The *minimal unitary representative map*, which associates a unitary with a minimal circuit realization.
- A sound and efficient MURM synthesis algorithm based on exact symbolic arithmetic.
- A fast linear-scan optimization algorithm that enumerates subcircuits and replaces them via MURM lookup.
- Machine-checked soundness proofs in Lean covering MURM minimality and the semantics preservation of the linear-scan optimization algorithm.
- A thorough evaluation of our implementation against state-of-the-art traditional optimizers and superoptimizers, including on megaquop-scale circuits.

2 Overview and Motivation

Imagine that we had an oracle that, given a unitary, returned its smallest circuit representative. To optimize a circuit, we could look at its subcircuits, compute the unitary of each one, and query the oracle. Whenever it returns a smaller circuit, we make the replacement. Circuit optimization would become a sequence of oracle queries. Can we build such an oracle and make it usable in practice?

2.1 Minimal Unitary Representative Maps

A *minimal unitary representative map* (MURM) maps a unitary to a smallest circuit that implements it. In other words, it stores a representative minimal circuit for each equivalence class of circuits. A MURM has the informal signature

$$\text{MURM} : U \longmapsto \text{a smallest circuit in Circ that implements } U,$$

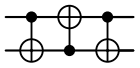
where Circ is a domain of circuits. There can be several smallest circuits for a given unitary; the MURM picks one.

The simplest examples are gates themselves. The Hadamard and CNOT examples are

$$U_H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \qquad \text{MURM}(U_H) = \text{—}\boxed{H}\text{—}\,.$$

$$U_{\text{CNOT}} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}, \qquad \text{MURM}(U_{\text{CNOT}}) =$$ 

Without a primitive SWAP gate, the two-qubit SWAP unitary maps to three CNOTs as its smallest representative:

$$U_{\text{SWAP}} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}, \qquad \text{MURM}(U_{\text{SWAP}}) =$$ 

Thus, even a small MURM quickly moves beyond one-gate identities and begins to capture useful multi-gate circuits.

Synthesizing a MURM. The way we synthesize the map is simple: enumerate circuits in order of size. We start with the empty circuit, then enumerate every one-gate circuit, every two-gate circuit, and so on until we reach the size bound of Circ. For each circuit C, we compute its unitary U_C. If the map does not yet contain U_C, we insert the pair $U_C \mapsto C$; otherwise, we discard C. Because we encounter shorter circuits first, the first circuit inserted for a unitary is necessarily a smallest representative.

This synthesis happens once, offline. The resulting MURM can then be reused across every input circuit we want to optimize. In this sense, the map packages a large collection of rewrite rules without requiring us to discover or state the rules individually: every circuit with unitary U can be replaced by $\text{MURM}(U)$.

Using a MURM efficiently and soundly. Storing an entire unitary matrix as a table key makes (1) the MURM large and (2) every lookup unnecessarily expensive. In practice, we index the MURM by a short *fingerprint* of the unitary, a hash of its entries:

$$\widehat{\text{MURM}} : \text{fp}(U) \longmapsto \text{a candidate smallest circuit for } U.$$

A fingerprint match is only a candidate match. After a lookup, we compare the query unitary with the unitary of the stored circuit and return the circuit only if they are equivalent.

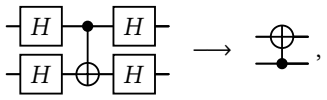
Floating-point representations of unitaries encounter arithmetic instability, potentially causing soundness bugs. Since we work with the standard discrete gate set, Clifford+T, we observe instead that we can represent matrix entries symbolically, in their *cyclotomic representation*, and compare that exact representation. This representation allows us to ensure soundness of our rewriting and not worry about numerical accuracy in our implementation.

Global phase and canonicalization. Above, we defined a MURM as a map from unitaries to circuits. More precisely, its keys are *projective unitaries*: equivalence classes of unitaries up to global phase. The matrices U and $e^{i\phi}U$ describe the same observable quantum evolution, so they must lead to the same MURM entry. We enforce this by putting every unitary into a canonical global-phase form before hashing or comparing it.

2.2 Optimizing with a MURM

Now that we have a MURM, we can view it as a large set of rewrite rules. Each entry says that any subcircuit with unitary U may be replaced by $\text{MURM}(U)$, the smallest circuit in the domain that implements U.

Existing superoptimizers typically construct explicit rules between pairs of equivalent circuits and then search for ways to apply them to an input [24, 40–43]. A rule such as



which flips the direction of a CNOT conjugated by Hadamards, is stored as this pair of circuits and fired wherever the left-hand side is found. This organization is problematic in three ways, each of which a MURM lets us reconsider.

- **Syntactic rules.** An explicit circuit-to-circuit rule only recognizes its particular left-hand pattern. Semantically equivalent subcircuits with different gates or shapes require more rules. *A MURM entry is semantic: it applies to every subcircuit with unitary U and returns the same smallest representative.*

- **Repeated matching.** Existing superoptimizers search the input circuit separately for every rule. So, matching becomes expensive as the rule set grows. *We reverse the search: enumerate the subcircuits in the input and query the MURM to discover which can simplify.*
- **Rewrite ordering.** Rewrites can expose or hide other matches, so a superoptimizer must choose an application order. Existing systems explore this combinatorial space using cost-guided backtracking, beam search, reinforcement learning, or equality saturation [24, 40, 42, 43]. *We instead make a fast, greedy left-to-right pass and repeat it to a fixpoint.*

Fig. 3 shows the practical consequence of MURMs. A 25-gate subcircuit found in a real input has the same unitary as a five-gate circuit. A rewrite-rule system would need this particular left-hand pattern in its rule library, or would have to serendipitously compose a long sequence of rewrites. A MURM discovers the replacement directly from the subcircuit's unitary.

A left-to-right scan. We scan the circuit from left to right, constructing on the fly an ordered set of subcircuits of a fixed bound on size (e.g., up to 20 gates and 3 qubits). For each subcircuit, we compute its unitary, query the MURM, and install any smaller representative. Because our algorithm is a fast left-to-right scan of a given circuit, we can afford to run it multiple times. Indeed, we run this process to a fixpoint, repeating the left-to-right scan until no further replacements are found. With fixed bounds, only a bounded number of candidates end at each gate, so our formal result bounds a pass by $O(n)$ candidate iterations.

Fig. 4 makes this enumeration concrete on a tiny toy example. The three gates are $g_1 = H(q_0)$, $g_2 = \textsc{cnot}(q_0, q_1)$, and $g_3 = T(q_1)$. With a two-gate bound on subcircuit size, the scan constructs for the queries the five subcircuits shown in the figure: three have one gate and two have two gates. The pair $[g_1, g_3]$ is not included: its gates act on different qubits and therefore do not form a *connected* subcircuit.

Making the scan fast. A direct implementation would reconstruct a unitary from scratch for every subcircuit, repeating most of the same matrix multiplications. Instead, each active subcircuit stores its exact unitary. Extending the subcircuit by one gate then requires only one incremental update, and the resulting unitary can be queried immediately. We also notice that the same subcircuit frequently recurs on different qubits. Renumbering the qubits a subcircuit touches to a canonical form (e.g., $0, 1, \ldots$) makes those recurring circuits syntactically identical, so we memoize the lookup, keyed by the subcircuit in this canonical form and storing its unitary alongside the answer the MURM gave—misses included, so they are never retried. A repeat then costs one hash lookup instead of a matrix product and a map query. Together, these optimizations aim to make MURM lookup as fast as possible.

3 Synthesizing MURMs

This section defines minimal unitary representative maps and explains how to synthesize them. We begin with the necessary foundations for quantum circuits and their semantics.

3.1 Quantum Circuits and Their Semantics

Circuits. A quantum circuit C is a sequence of gates acting on a fixed set of qubits. We write $|C|$ for the number of gates in C and ε for the empty circuit. The *semantics* of C, written U_C, is the unitary matrix obtained by composing its gates in execution order. If C acts on n qubits, then U_C is a $2^n \times 2^n$ complex matrix, and $U_\varepsilon = I$.

For the purposes of this paper, we focus on circuits over a finite gate set, namely, the standard Clifford+T gate set that is popular in fault-tolerant computation. In principle, however, our approach applies to any finite gate set. Fig. 5 gives the semantics of its gates. We write G_i for a one-qubit gate G acting on qubit i, and $\textsc{cnot}_{i,j}$ for a controlled-NOT gate whose control is qubit i and whose target is qubit j.

Circuit equivalence. Two circuits are equivalent when they have the same unitary up to *global phase*: multiplying every amplitude by the same complex number $e^{i\theta}$ does not change the outcome of a computation. Formally,

$$C_1 \equiv C_2 \quad \Longleftrightarrow \quad U_{C_1} = e^{i\theta} U_{C_2} \quad \text{for some } \theta \in \mathbb{R}.$$

This notion of equivalence is preserved by composition, so replacing a subcircuit by an equivalent one preserves the semantics of the entire circuit up to global phase.

It is useful to assign all matrices that differ only by global phase a single canonical form, so we can compare equivalence syntactically. Fix any deterministic rule that picks a nonzero entry of a matrix—say, the first nonzero entry in row-major order—and write u for that *pivot* entry of U. We define

$$\text{canon}(U) = \frac{|u|}{u}\, U,$$

which makes the pivot positive and real. Scaling U by a phase does not change which entries are nonzero, so the rule picks the same position for U and $e^{i\theta}U$; it multiplies the pivot by $e^{i\theta}$, which the factor $|u|/u$ then cancels.

Example 3.1 (Canonicalization in three steps)**.** Let $C = S; H; S; H; S$. Composing its five gates gives

$$U_C = \frac{e^{i\pi/4}}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = e^{i\pi/4}\, U_H,$$

so U_C is U_H scaled by a phase, and comparing the two matrices entry by entry finds no match. The canonicalization rule picks the top-left entry of U_C as the pivot, namely $u = e^{i\pi/4}/\sqrt{2}$, for which $|u|/u = e^{-i\pi/4}$; multiplying through by that factor leaves $\text{canon}(U_C) = U_H$. Running the same

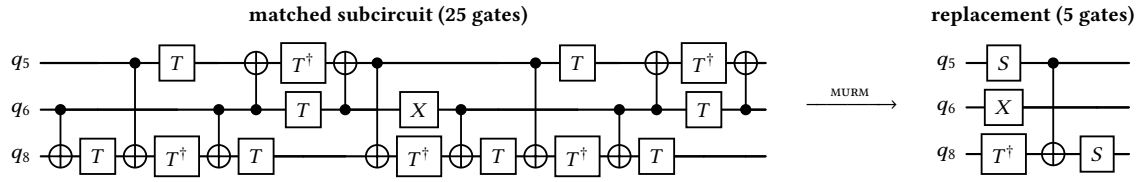


Figure 3. A large semantic rewrite found in practice: a 25-gate subcircuit replaced by a five-gate circuit with the same unitary.

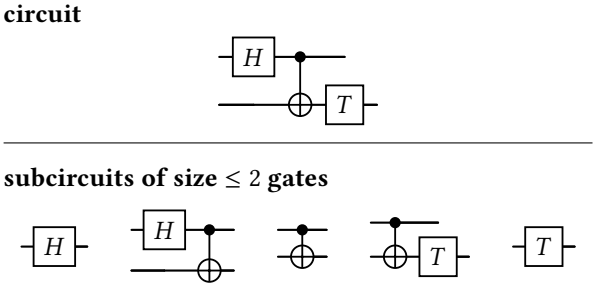


Figure 4. All connected subcircuits of at most two gates.

$$U_H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \qquad U_S = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$$

$$U_T = \begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix} \qquad U_{\mathrm{CNOT}_{1,2}} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$$

Figure 5. Semantics of the Clifford+T gates we use.

process on U_H picks $u = 1/\sqrt{2}$, so $\mathrm{canon}(U_H) = U_H$. The five-gate circuit and the one-gate circuit H therefore share a single MURM entry, whose value is H, and a lookup on C returns it.

Lemma 3.2 (Canonical form)**.** *For all unitaries U and V of equal dimension,*

$$\mathrm{canon}(U) = \mathrm{canon}(V) \iff U = e^{i\theta}V \text{ for some } \theta \in \mathbb{R}.$$

Canonicalization therefore lets us treat circuit equivalence as syntactic equality between canonical matrices. We omit paper proofs throughout: every lemma and theorem we state is machine-checked in Lean [14].

3.2 Synthesizing a MURM

With semantics and canonical forms in hand, we can define the map at the center of our approach and give the algorithm that builds it. A minimal unitary representative map (MURM) answers a single question: given a unitary, what is the smallest circuit implementing it? The map is always taken over some fixed space of circuits, so we begin there.

The space of circuits. Let Circ be the finite set of all circuits over a chosen gate set that act on n qubits and use at most k gates. This is the space the map draws its answers from; we fix n and k until the end of the section.

Algorithm 1 MURM synthesis

```
M ← {canon(I) ↦ ε}
Frontier ← {ε}
for size ← 1 to k do
    Frontier′ ← ∅
    for all C ∈ Frontier and gates g do
        U ← canon(U_{C;g})
        if M(U) is undefined then
            M(U) ← C; g
            add C; g to Frontier′
    Frontier ← Frontier′
return M
```

Defining a MURM. A MURM maps unitary matrices to circuits in Circ. The keys of the MURM are canonicalized: a unitary U is looked up as $\mathrm{canon}(U)$, so all matrices that differ only by global phase share a single entry. The map is partial, since a bounded space of circuits cannot implement every unitary.

Definition 3.3 (MURM)**.** A *minimal unitary representative map* over Circ is a partial map M from unitaries to circuits in Circ such that:

- *Soundness*: If $M(U)$ is defined, $\mathrm{canon}(U_{M(U)}) = U$.
- *Minimality*: $\forall C \in \mathrm{Circ}$ s.t. $\mathrm{canon}(U_C) = U$, $|M(U)| \le |C|$.

Soundness means that the circuit $M(U)$ implements U. Minimality means that no circuit in Circ implements U with fewer gates.

Breadth-first synthesis. Alg. 1 builds a MURM by enumerating circuits in order of size. It keeps a map M and a *frontier* Frontier of circuits of the current size. Each round extends every circuit in the frontier by one gate and inserts the result if its canonical unitary has no entry yet; the newly inserted circuits form the next frontier. Because rounds proceed in order of size, the first circuit inserted for a unitary is a smallest one, which gives minimality. Later rediscoveries are discarded, and the enumeration order breaks ties between distinct smallest circuits.

To keep enumeration from exploding, the algorithm extends only the circuits it stored in M, not every circuit it built.

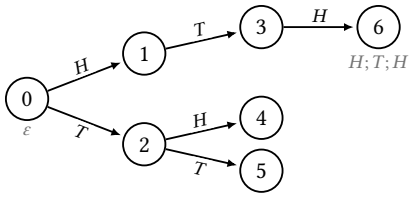


Figure 6. Interned circuits, over the gate set $\{H, T\}$.

Discarding the rest of the circuits does not impact minimality: If two circuits are equivalent, then appending a gate g to both results in equivalent circuits. Every unitary reachable by extending a discarded circuit is reachable by extending the stored representative.

Theorem 3.4 (Synthesis)**.** *Alg. 1 returns a* MURM *over* Circ.

Note that the algorithm can be terminated early, before reaching the size bound k. It still returns a sound MURM over the smaller space of circuits it enumerated.

3.3 Efficiency and Correctness Considerations

There are two challenges with implementing Alg. 1 in practice: (1) the map can be enormous, and (2) the unitary equality for lookup must be decided exactly.

Fingerprints. A MURM holds hundreds of thousands to tens of millions of entries, and its keys (unitaries) are expensive to store. Comparing unitaries entry by entry on every lookup is also slow. We therefore key the map by a *fingerprint* of the canonical unitary—for example, a 64-bit hash of its entries—so that no unitary is stored at all:

$$\widehat{M} : \mathrm{fp}(U) \longmapsto \text{a circuit.}$$

A fingerprint is lossy, since distinct unitaries can have the same fingerprint. Alg. 2 shows how to soundly query $\widehat{M}$: to ensure that no collision occurred, we rebuild the circuit's unitary and compare it against the query unitary. A collision therefore may cost a missed rewrite, not an incorrect one.

Interned circuits. Fingerprinting shrinks the keys of the map; its values (circuits) need the same treatment, since storing c circuits of up to k gates each costs $O(ck)$ space. We exploit the fact that Alg. 1 extends only circuits it has already stored, so the stored circuits are *closed under prefixes*: each one is a shorter stored circuit followed by one gate.

To compactly represent MURM circuits, we *intern* them in a tree data structure: a tree rooted at the empty circuit ε. Each node stores only the last gate of its circuit and the index of its parent, the node holding everything before that gate; the circuit itself is recovered by walking to the root.

Fig. 6 shows six circuits stored this way. Each edge is labeled with the gate that the child adds, so node 6 denotes $H; T; H$ and node 3 denotes $H; T$. No child of node 1 adds H, because $H; H$ recomputes the identity that ε already represents. A prefix shared by a thousand circuits is thus stored once, and a node is a fixed fifteen bytes—eight of fingerprint, four of parent index, three of gate—whatever the length of the circuit it denotes. A map holding c circuits at each width up to n therefore takes $15nc$ bytes.

Algorithm 2 Efficient MURM lookup

Require: a unitary U and a fingerprinted map $\widehat{M}$
1: $V \leftarrow \mathrm{canon}(U)$
2: $C \leftarrow \widehat{M}(\mathrm{fp}(V))$
3: **if** C is undefined **or** $\mathrm{canon}(U_C) \neq V$ **then**
4: **return** undefined
5: **return** C

Exact arithmetic. Checking equivalence of two unitaries can be problematic with floating-point arithmetic. Rounding error accumulates as gates are composed, and a tolerance-based comparison errs in both directions: it can accept two circuits that are not equivalent, and reject two that are.

Since we are dealing with Clifford+T circuits, we can represent their unitaries exactly using a symbolic representation instead of a numerical one. Every entry of a unitary matrix of a Clifford+T circuit lies in the *cyclotomic ring* $\mathbb{Z}[\frac{1}{\sqrt{2}}, i]$ (see, e.g., [16, 23]). Writing $\omega = e^{i\pi/4}$, every element of that ring can be put in the form

$$\frac{a + b\,\omega + c\,\omega^2 + d\,\omega^3}{\sqrt{2}^k},$$

where $a, b, c, d, k \in \mathbb{Z}$ and $k \geq 0$, so a matrix entry is a five-tuple of integers (a, b, c, d, k). We perform all operations in this ring—composing gates, canonicalizing, and the final comparison—so the arithmetic is integer arithmetic and every comparison is exact. Canonicalization is the one step that needs care, since dividing by $|u|$—the modulus of the pivot entry canon picks—would leave the ring. Appendix B gives the construction we use instead and shows that it decides the same equality as Lemma 3.2.

4 Linear-Scan Optimization

We now present the MURM-based optimization algorithm and then show how to implement it efficiently.

4.1 The Algorithm

The optimizer takes a circuit and a MURM, and returns an optimized circuit. First, suppose we have a set of subcircuits, SubCirc, of the input circuit. The optimizer asks the MURM whether it knows something smaller for each $W \in \mathsf{SubCirc}$ and installs the replacements it offers. This presentation is independent of how SubCirc is chosen. Our implementation constructs one such set on the fly using the anchored connected-subcircuit enumeration defined in Appendix A. This is relatively standard and we therefore relegate the details to the appendix.

Algorithm 3 Linear-scan optimization

Require: a circuit C and a MURM M

```
function Optimize(C, M)
    let SubCirc be a set of subcircuits of C
    for all W ∈ SubCirc, ordered by last gate do
        W′ ← M(canon(U_W))
        if W′ is defined and |W′| < |W| then
            replace W with W′ in C
            skip the subcircuits that overlap W
    return C
```

To make the enumeration of subcircuits manageable in practice, we bound the number of qubits a subcircuit may touch by q and the number of gates it may contain by ℓ, which keeps its unitary small enough to compute. (In practice, it is useful to set ℓ larger than the largest circuit in the MURM.)

The linear-scan optimization. Alg. 3 examines the subcircuits in SubCirc in order of their last gate and rewrites greedily. Whenever a MURM lookup returns a strictly smaller circuit, the replacement is installed, and the subcircuits overlapping it are abandoned so that no gate is rewritten twice.

Every replacement is sound. The MURM guarantees $W' \equiv W$, and the condition on SubCirc guarantees that the members of W can be brought together, rewritten, and the skipped gates returned to their places. Thus, installing W' in place of W leaves the surrounding circuit alone (see Appendix A).

Theorem 4.1 (Soundness). *For every circuit C and every MURM M, $\textsc{Optimize}(C, M) \equiv C$ and $|\textsc{Optimize}(C, M)| \leq |C|$.*

For the on-the-fly enumeration of Appendix A and a fixed bound on subcircuit size, the algorithm runs in time linear in the size of the input circuit. This makes the algorithm fast, though one has to implement it carefully, as we see next.

4.2 Incrementality and Fixpoints

Alg. 3 is easy to implement, but a naive implementation can be very slow: it builds every candidate subcircuit and every subcircuit's unitary from scratch. Instead, we can build both *incrementally* and memoize the results of the MURM lookups to avoid repeating them.

Additionally, because the linear scan is fast, we can afford to run Alg. 3 repeatedly until it reaches a fixpoint, as each iteration may expose new opportunities for optimization!

Incremental unitaries. Alg. 3 walks the circuit from left to right and every MURM query needs the queried subcircuit's unitary. Building each of those unitaries from scratch would dominate everything else the pass does—and it would also be redundant, because consecutive subcircuits overlap almost entirely. A subcircuit the scan is about to consider is usually one it has just considered with a single gate appended, so rather than recompute its unitary we take the one we already have and extend it. Concretely, having computed the unitary of $g_1; \ldots; g_n$, the unitary of $g_1; \ldots; g_n; g_{n+1}$ follows by multiplying by the unitary of g_{n+1}.

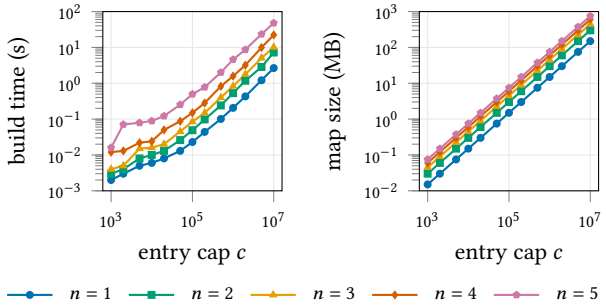


Figure 7. Cost of MURM synthesis per entry cap c and qubit bound n. *Left*: build time. *Right*: size. (Log-log plots.)

Additionally, we observe that the same subcircuit *shape* often recurs heavily. However, the same subcircuit will appear applied to different subsets of the qubits. But these subcircuits ultimately denote the same unitary transformation. Therefore, to minimize MURM lookups, we can canonicalize subcircuits—by giving their qubits canonical names—and memoize the results of past MURM lookups for a given subcircuit shape. A memoization hit skips both the unitary computation and the lookup.

Incremental fixpoint optimization. A single pass of Alg. 3 may expose new optimization opportunities. Since, as discussed, the pass is fast—it's just a linear scan through a circuit—we can afford to run it again. Indeed, we run Alg. 3 repeatedly to a fixpoint, until it leaves the circuit unchanged. In subsequent iterations, the only subcircuits where a rewrite may be possible are those that overlap with a replacement made in the previous iteration—the *diff*. So, to avoid wasted attempts at rewriting in later iterations, we only focus on subcircuits that overlap with the diff.

5 Implementation and Evaluation

We implemented our approach in the TZAP optimizer, written in Rust [1]. It accepts circuits in a subset of OpenQASM 2.0. To ensure the correctness of our Rust implementation, we evaluated TZAP on a large set of small circuits and formally verified equivalence of the optimized and original circuits using QCEC [12].

Research questions. We ask:

RQ1. How long does it take to construct a MURM, how large is the result, and how do both grow with the qubit count and the number of circuits stored?

RQ2. How does TZAP compare to state-of-the-art optimizers, in both optimization quality and runtime?

RQ3. How are TZAP's speed and its reductions affected by the incremental techniques described in Sec. 4.2, iteration to a fixpoint, and the size of the MURM?

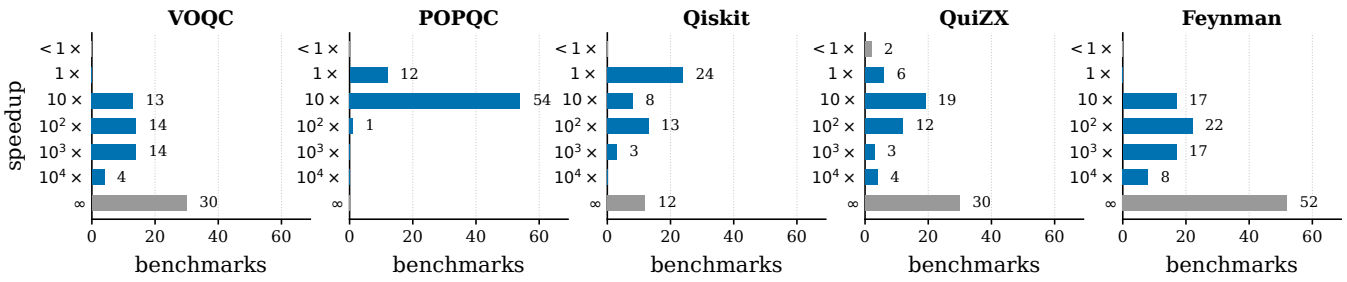


Figure 8. TZAP's speedup over each traditional optimizer, one panel per tool. A bar counts the benchmarks whose speedup falls in that order-of-magnitude band. The gray $<1\times$ bar counts the benchmarks the other tool wins, and ∞ those it returned no circuit for. Benchmarks on which both tools finish inside a second are omitted.

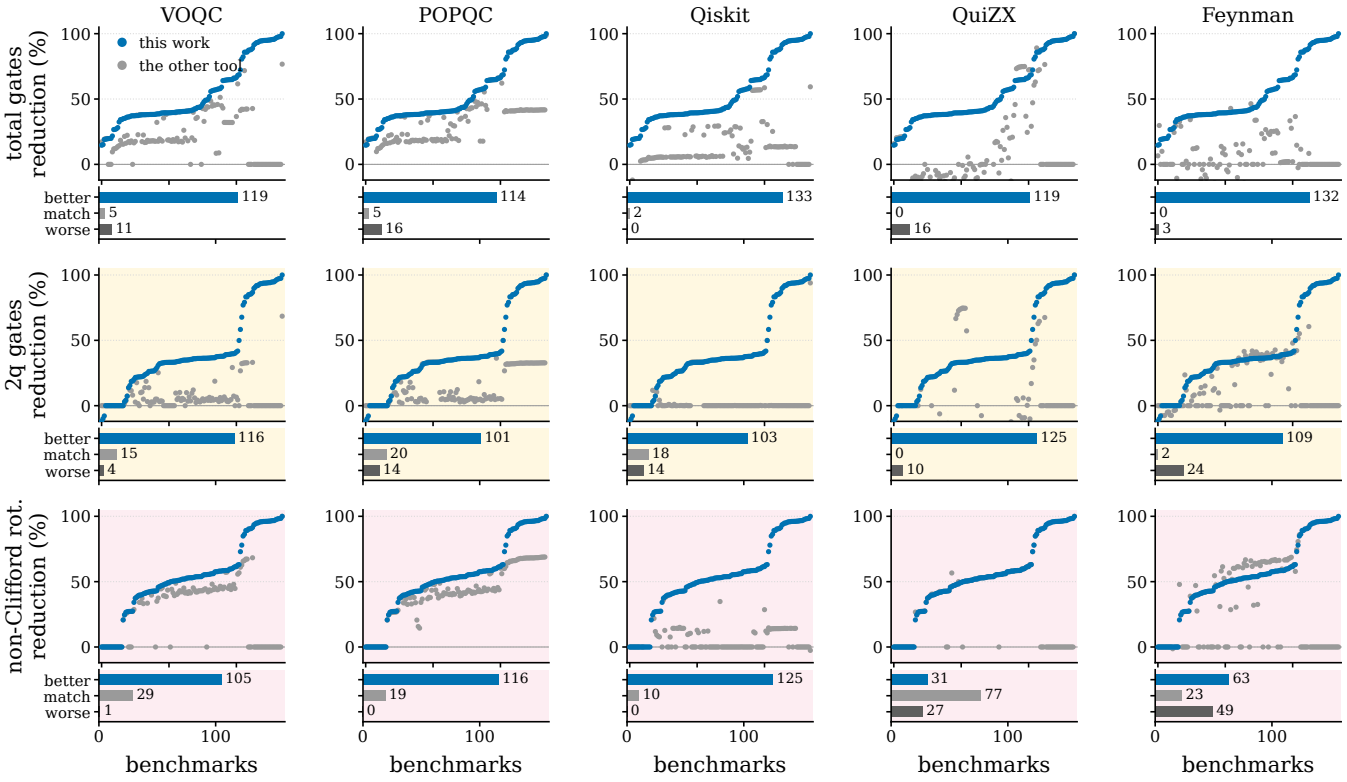


Figure 9. Gate reduction against each traditional optimizer: one column per tool, one row per metric. Blue is TZAP, gray the other tool, and each point is one benchmark. Benchmarks are ordered by TZAP's reduction, so a blue point above a gray one is a benchmark where TZAP removed more. The histogram below each plot counts the benchmarks on which TZAP's reduction is better than, matches, or is worse than the other tool's.

5.1 RQ1: The Cost of Synthesizing a MURM

Summary. Synthesizing a MURM costs time and space linear in the entries it stores, and only takes milliseconds for our default MURM configuration.

Alg. 1 synthesizes a MURM by enumerating circuits in order of size. We measure its time and space while sweeping the entry cap c from 1,000 to 10,000,000 and the qubit bound n from 1 to 5.

Fig. 7 reports the results. Both time and space grow linearly with the cap c at every qubit count. Size grows exactly linearly: interning makes every entry fifteen bytes whatever circuit it stores (Sec. 3.3), so a map is $15nc$ bytes plus a header. Time grows linearly too, because the enumeration extends only the circuits it kept, once per gate in the library, so the work per entry stored is constant. Below $c \approx 50{,}000$ process startup dominates. Appendix C shows similar results for an alternative gate set.

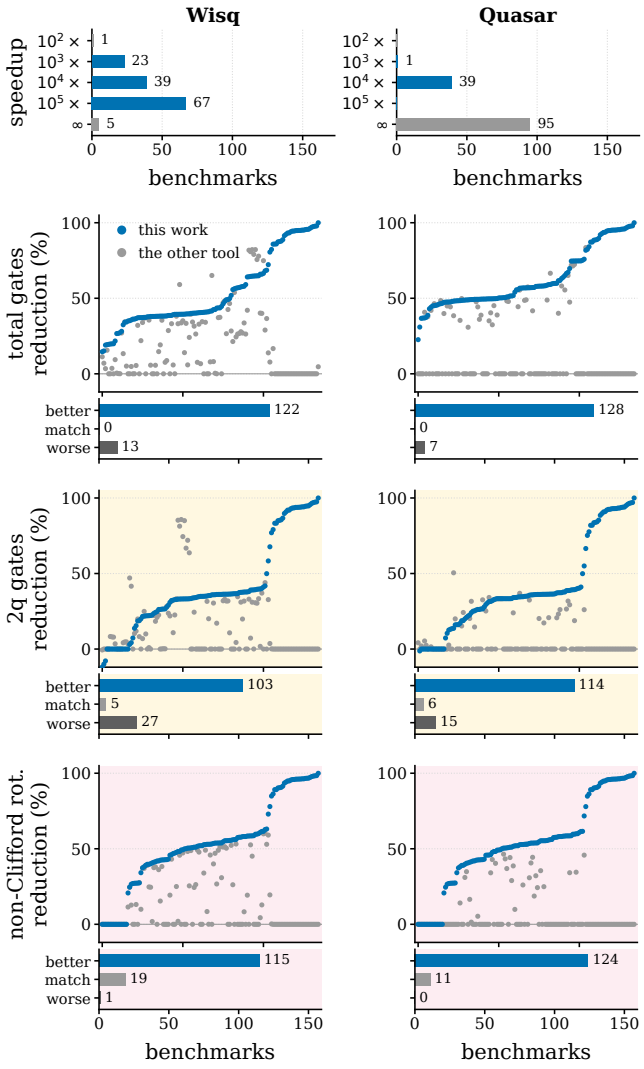


Figure 10. TZAP against the superoptimizers. *Top*: speedup, binned by order of magnitude and timed to each tool's best result, with ∞ counting the benchmarks that returned no circuit. *Bottom*: reduction, drawn as in Fig. 9. Quasar's column uses Quasar's basis on the circuits it was given; Wisq's uses the Clifford+T suite.

Qubit count should in principle cost exponentially more time, since an entry at bound n composes $2^n \times 2^n$ matrices. In practice the MURM needs only small qubit counts, and at that scale the growth is mild: the cost per entry rises from 266 ns at $n = 1$ to 958 ns at $n = 5$, a factor of 3.6, while the matrices grow from 2×2 to 32×32.

In RQ2, the MURM we use only takes 0.15 seconds to synthesize and 9 MB to store.

5.2 RQ2: Comparison with Existing Optimizers

Summary. TZAP finishes all 135 benchmarks at a median of 0.08 seconds, orders of magnitude faster than other tools, while providing the largest reductions in total and two-qubit gate counts. On non-Clifford rotations it is competitive with the two tools built for them.

Benchmarks. We combine four existing benchmark sets: (1) The *Feynman* suite [2] is the standard arithmetic and reversible-logic benchmark set. (2) *Cobble* [45] emits block-encoding circuits for quantum linear algebra. We use it to build incrementally larger block-encoding circuits, from $2K$ to $11M$ gates—the largest circuits in the evaluation. We also take the suites of (3) GUOQ [41], the optimizer we run below through Wisq, and (4) Quasar [43].

Removing circuits under 1,000 gates leaves 135 circuits, spanning $1K$ to $11M$ gates with a median of $18K$; 46 exceed a hundred thousand gates and five exceed ten million (Appendix G). Most circuits are Clifford+T, but some include arbitrary Z-rotations. Like VOQC, Feynman, and POPQC, TZAP handles them with a standard rotation-merging pass [29].

Tools and setup. We compare against three traditional optimizers—Qiskit [20] at optimization level 3, VOQC [18], and POPQC [25], the parallel successor to OAC [7]—and two search-based superoptimizers: Wisq [28, 41] by simulated annealing and Quasar [43] by equality saturation. For non-Clifford rotations, we add Feynman [2] and QuiZX [21, 32], the Rust port of PyZX.

Each tool–circuit combination receives 8 cores, 32 GB of memory, and three hours on identical cluster machines; Appendix J gives the invocations.

TZAP's configuration. We run TZAP with a MURM bounded at three qubits, 200,000 entries per qubit count, and construct subcircuits of at most $\ell = 25$ gates (Sec. 5.3 studies the trade-offs of a larger map.)

Traditional optimizers. Fig. 8 reports the speedup and Fig. 9 the gate reduction, on each of three metrics: total gates, two-qubit gates, and non-Clifford rotations.

POPQC is the fastest tool we compare against, and TZAP is faster than it on every one of the 135 benchmarks and reduces more of the gates on 114 of them (see histograms in Fig. 9). TZAP finishes all 135 in a median of 0.08 seconds against 0.97 for POPQC. TZAP runs single-threaded; POPQC is the one tool here that parallelizes and uses all eight cores, so the 10× gap is against eight times the hardware. The speedup holds at every circuit size and reaches 117× on `ground_state_estimation` (0.12 seconds against 14.4).

TZAP outperforms POPQC on the majority of the benchmarks in terms of gate reduction, usually with a significant gap. Every margin below is a median of *per-benchmark* differences, over the circuits both tools returned a circuit

for. We report margins in percentage points (pp): for example, if TZAP removes 40% of a benchmark's gates and another tool removes 25%, TZAP's margin on that benchmark is 40% − 25% = 15 pp. By this measure, TZAP removes 18.4 pp more of the total gates than POPQC does, 28.9 pp more of the two-qubit gates, and 11.8 pp more of the rotations. The margin widens on large circuits: over the 31 Cobble block-encoding circuits it is 52.8 pp on total gates.

Against VOQC and Qiskit, TZAP is one to four orders of magnitude faster. Both time out on large benchmarks even with the three-hour limit—VOQC on 30 and Qiskit on 12. (Appendix H plots speedup against circuit size, where it climbs with the input circuit size against every tool but POPQC, whose gap stays flat.) TZAP also reduces more: 18.8 pp more of the total gates than VOQC at the median, and 32.8 pp more than Qiskit.

Rotation specialists. Feynman times out on 52 circuits and QuiZX on 26, and both can be orders of magnitude slower than TZAP on the circuits they do finish.

Non-Clifford rotations are the one metric on which either leads. Over the 83 benchmarks Feynman completes, it is 1.5 pp ahead of TZAP at the median, and ahead on 49 of them against our 11. QuiZX draws with TZAP at the median over its 105: the tools tie on 77, QuiZX leads on 27, and TZAP on one.

Both Feynman and QuiZX tend to increase total gate count. Over the 105 circuits QuiZX completes, TZAP is 44.7 pp ahead on total gates and 62.9 pp on two-qubit gates. Feynman, over its 83, is 40.2 pp behind TZAP on total gates and 3.6 pp behind on two-qubit gates.

Superoptimizers. We time each superoptimizer to its best result rather than to exit; Wisq, for example, reaches its best after a median of 64% of its three-hour budget.

Fig. 10 reports the results. TZAP's median speedup over Wisq is 122,959×, and it outperforms Wisq by 29.6 pp on total gates at the median of the 130 benchmarks Wisq returns a circuit for, and by 27.0 pp on rotations—ahead on 110 of them and behind on one.

That margin is almost entirely a matter of size: On the 58 benchmarks between one and ten thousand gates, TZAP's lead narrows to 5.9 pp on total gates and 2.7 pp on two-qubit gates, where Wisq is ahead on 24 of the 58. On the 72 larger ones, Wisq effectively makes no gate reductions at all.

Quasar targets the IBM gate set, $\{\text{CX}, R_z, \text{SX}, X, Z\}$, so we lower every benchmark into that basis and run TZAP in the same basis. Quasar receives 64 GB, twice the other tools: its setup assumes that much, and at 32 GB it fails on most of the suite.

Even so, it optimizes only the smallest 40 circuits and exhausts memory on the other 95. Those 40 cost it 106.5 hours of CPU time; TZAP optimizes them in 9.8 seconds.

On those 40, TZAP removes a median 4.8 pp more of the total gates and 12.5 pp more of the rotations, ahead on 33 and 29 of them, respectively, and behind on 7 and none. Two-qubit gates are the exception: the tools draw at the median, with TZAP ahead on 19 and Quasar on 15. Appendix I compares against Quasar's published results on its own suite of small circuits (median circuit size of 232 gates).

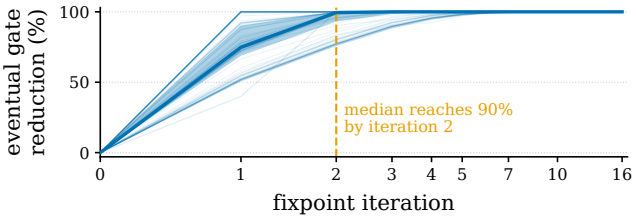


Figure 11. Fraction of a benchmark's eventual gate reduction reached by each iteration. Faint: the 135 benchmarks; solid: their pointwise median; band: the 25th–75th percentiles.

Circuits with large reductions. TZAP reduces twenty-two benchmarks by more than 90%; we checked them all. `ground_state_estimation` collapses from 390,180 gates to 17; we formally verified equivalence of the input and optimized circuits with QCEC [12]. The other twenty-one are Cobble block-encoding circuits, and Cobble's own paper [45] reports the same redundancy, reachable there by optimizing the high-level program instead. For the smaller ones, we simulated the input and optimized circuits on a range of starting states and confirmed that they agree.

5.3 RQ3: Speed and the Reductions

Summary. Incrementality (Sec. 4.2) makes TZAP 10× faster. Iterating to a fixpoint generates sizable gate reductions (9 pp median, 48 pp max). A larger MURM closely matches the reductions of a small MURM and is significantly slower.

Incrementality. We compare against TZAP with the incremental mechanisms (Sec. 4.2) off, so each subcircuit is examined afresh and its unitary built from scratch. The median slowdown is 10×; the whole-suite runtime rises from 1.5 to 18.9 minutes (12.3×).

Recall that the incrementality techniques apply within an iteration (incrementally constructing unitaries) and across iterations (maintaining a *diff* of modified subcircuits). Incrementality *within* an iteration is worth 7.8× at the median on its own. Incrementality *across* iterations of the fixpoint—*diffing*—is worth 1.13×.

Fixpoints. Compared with running Alg. 3 iteratively to a fixpoint, a single iteration of Alg. 3 leaves gates behind on 114 of the 135 benchmarks; running to convergence removes 9.6 pp more of the input at the median and a maximum of 48.2 pp. The median benchmark needs three iterations to

converge, 21 benchmarks need one iteration, and the most needs 16.

Fig. 11 summarizes this data for all benchmarks. At the median, the first iteration removes 74.8% of the total gates that will be removed after a fixpoint is reached; the second iteration reaches 90%.

The size of the MURM. We compare our configuration from RQ2—three qubits, 200,000 entries per qubit count, subcircuit size bound of $\ell = 25$—against one raised on every axis: five qubits, 5,000,000 entries, $\ell = 40$.

The larger map costs 16.2× the optimization time at the median and 5.2× over the suite, counting optimization alone. It also takes up to 1.3 s to load, which the default's map does not, so end-to-end the gap is wider still: Appendix E measures it whole-process and finds 41× at the median. The two sizes of the MURM agree on all three metrics at once, to within 0.5 pp, on 103 benchmarks. The improvements of the larger MURM are small, e.g., a median of 0.81 pp and at most 1.31 pp on total gates, and a median of 1.16 pp on two-qubit gates. In summary, *a large MURM does not appear to be worth the runtime cost for this diverse suite.*

That comparison above moves all three bounds at once. Appendix F separates them, sweeping every combination of qubit bound, entry cap, and subcircuit bound—1,260 configurations over the same 135 benchmarks.

6 Related Work

Superoptimization. We call our approach a superoptimizer because it synthesizes far more rewrites than a hand-written optimizer can encode and matches or exceeds the reductions of search-based superoptimizers (Sec. 5.2).

Unlike other quantum-circuit superoptimizers, we do not search rewrite orderings. QUESO uses beam search [40], Quartz cost-based backtracking [42], GUOQ simulated annealing [41], Quasar equality saturation [43], and QuaRL a learned rewrite policy [24]. (Classical superoptimization similarly searches for short instruction sequences, using exhaustive or synthesis-based techniques [17, 27, 34].) Instead, we greedily scan the circuit to a fixpoint, trading optimality for speed.

Bansal and Aiken's assembly superoptimizer is perhaps the closest prior work [9]. Offline, it harvests x86 instruction windows, enumerates cheaper candidates, and filters them with random-execution fingerprints before SAT validation; online, it canonicalizes each window and queries the database. Like us, it pays for search once to make online optimization a lookup. Later work likewise generates local optimizations or instruction-selection rules from semantics [10, 11, 13, 30]. Its database maps canonicalized source patterns to replacements; a MURM instead maps each exact unitary modulo global phase to a minimal representative.

In reversible-circuit optimization, Shende et al. [36] enumerated optimal circuits for every three-bit reversible function. Prasad et al. [31] use such a library for peephole optimization: they look up a subcircuit's permutation and replace it with the stored minimum-cost circuit. This is the most direct ancestor of the MURM. We generalize from classical permutations to complex-valued unitaries, requiring exact cyclotomic arithmetic and canonicalization modulo global phase. Because even a complete four-bit reversible library is infeasible [31], our capped MURM is optimal for its entries but incomplete.

Unitary synthesis. TZAP applies unitary synthesis [6, 16, 19, 23, 33, 35, 39], as do GUOQ and BQSKit [41, 44]. BQSKit numerically resynthesizes small-qubit blocks; GUOQ repeatedly resynthesizes during its online randomized search. Both procedures can take minutes [41, 44]. A MURM instead provides an optimal but incomplete oracle built offline: a hit returns a minimum-gate circuit and a miss produces no rewrite, making lookup fast and predictable.

General optimization techniques. Our approach could serve as an additional pass in Qiskit, t|ket⟩, or staq [3, 20, 38]. Traditional compilers and specialized optimizers use fixed passes for cancellation, rotation merging, and algebraic or graphical simplification [4, 5, 15, 18, 21, 22, 29].

OAC and POPQC pursue local optimality over bounded contiguous segments using an external oracle, whereas we supply an offline-synthesized MURM and inspect connected subcircuits in a linear scan [7, 25].

7 Conclusion

We have presented a fast and scalable approach for quantum-circuit superoptimization. The key underlying idea is the synthesis of a map from unitaries to minimal circuits implementing them. This map is then used as an oracle driving a fast, greedy optimization algorithm that runs to a fixpoint on a given input circuit.

We see many avenues for future exploration: First, we would like to explore how to extend the MURM to continuous gate sets, e.g., with arbitrary Z-rotations. Second, it would be interesting to investigate the optimization gains from relaxing the greediness of the algorithm, e.g., by exploring many subcircuit orderings in parallel or via reinforcement learning. This would potentially allow us to eke out additional optimization gains at the cost of longer runtime. Third, it would be interesting to ask whether our ideas can be ported to other representations of quantum circuits, e.g., Pauli-based computation, which has become a standard intermediate representation in the fault-tolerance literature.

Acknowledgments

I am grateful for the discussions I've had with Abtin Molavi, Amanda Xu, Swamit Tannu, and Eugene Lin about this work.

References

[1] Aws Albarghouthi. 2026. Linear-Time T-Gate Optimization via Random Abstraction. arXiv:2605.13929 [cs.PL] https://arxiv.org/abs/2605.13929

[2] Matthew Amy. 2019. *Formal Methods in Quantum Circuit Design*. Ph. D. Dissertation. University of Waterloo. Retrieved September 14, 2026 from https://hdl.handle.net/10012/14480

[3] Matthew Amy and Vlad Gheorghiu. 2020. staq—A full-stack quantum processing toolkit. *Quantum Science and Technology* 5, 3 (2020), 034016. doi:10.1088/2058-9565/ab9359

[4] Matthew Amy and Joseph Lunderville. 2025. Linear and non-linear relational analyses for quantum program optimization. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 1072–1103. doi:10.1145/3704873

[5] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014), 1476–1489. doi:10.1109/TCAD.2014.2341953

[6] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 818–830. doi:10.1109/TCAD.2013.2244643

[7] Jatin Arora, Mingkuan Xu, Sam Westrick, Pengyu Liu, Dantong Li, Yongshan Ding, and Umut A. Acar. 2025. Local Optimization of Quantum Circuits. In *IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, Piscataway, NJ, USA, 572–583. doi:10.1109/QCE65121.2025.00069

[8] Ryan Babbush, Robbie King, Sergio Boixo, William Huggins, Tanuj Khattar, Guang Hao Low, Jarrod R McClean, Thomas O'Brien, and Nicholas C Rubin. 2026. Grand Challenge of Quantum Applications. *PRX Quantum* 7, 2 (2026), 020101. doi:10.1103/6r9l-lynr

[9] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 394–403. doi:10.1145/1168857.1168906

[10] Sebastian Buchwald. 2015. Optgen: A Generator for Local Optimizations. In *Compiler Construction (CC 2015) (Lecture Notes in Computer Science, Vol. 9031)*, Björn Franke (Ed.). Springer, Cham, Switzerland, 171–189. doi:10.1007/978-3-662-46663-6_9

[11] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO '18)*. ACM, New York, NY, USA, 300–313. doi:10.1145/3168821

[12] Lukas Burgholzer and Robert Wille. 2021. QCEC: A JKQ Tool for Quantum Circuit Equivalence Checking. *Software Impacts* 7 (2021), 100051. doi:10.1016/j.simpa.2020.100051

[13] Ross Daly, Caleb Donovick, Caleb Terrill, Jackson Melchert, Priyanka Raina, Clark Barrett, and Pat Hanrahan. 2024. Efficiently Synthesizing Lowest Cost Rewrite Rules for Instruction Selection. In *Proceedings of the 24th International Conference on Formal Methods in Computer-Aided Design (FMCAD '24)*, Nina Narodytska and Philipp Rümmer (Eds.). TU Wien Academic Press, Vienna, Austria, 8–17. doi:10.34727/2024/isbn.978-3-85448-065-5_7

[14] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science, Vol. 12699)*. Springer, Cham, Switzerland, 625–635. doi:10.1007/978-3-030-79876-5_37

[15] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. 2020. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum* 4 (2020), 279. doi:10.22331/q-2020-06-04-279

[16] Brett Giles and Peter Selinger. 2013. Exact synthesis of multiqubit Clifford+T circuits. *Physical Review A* 87, 3 (2013), 032332. doi:10.1103/PhysRevA.87.032332

[17] Torbjörn Granlund and Richard Kenner. 1992. Eliminating Branches Using a Superoptimizer and the GNU C Compiler. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 341–352. doi:10.1145/143095.143146

[18] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434318

[19] Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home, and Matthias Christandl. 2016. Quantum Circuits for Isometries. *Physical Review A* 93, 3 (2016), 032318. doi:10.1103/PhysRevA.93.032318

[20] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum Computing with Qiskit. doi:10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[21] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large scale automated diagrammatic reasoning. In *Proceedings 16th International Conference on Quantum Physics and Logic (QPL 2019) (Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 318)*. Open Publishing Association, Waterloo, Australia, 229–241. doi:10.4204/EPTCS.318.14

[22] Aleks Kissinger and John van de Wetering. 2020. Reducing the number of non-Clifford gates in quantum circuits. *Physical Review A* 102, 2 (2020), 022406. doi:10.1103/PhysRevA.102.022406

[23] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. 2013. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates. *Quantum Information & Computation* 13, 7–8 (2013), 607–630. doi:10.26421/QIC13.7-8-4

[24] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. Quarl: A Learning-Based Quantum Circuit Optimizer. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1, Article 114 (April 2024), 28 pages. doi:10.1145/3649831

[25] Pengyu Liu, Jatin Arora, Mingkuan Xu, and Umut A. Acar. 2025. POPQC: Parallel Optimization for Quantum Circuits. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*. ACM, New York, NY, USA, 269–283. doi:10.1145/3694906.3743325

[26] Seth Lloyd. 1996. Universal quantum simulators. *Science* 273, 5278 (1996), 1073–1078. doi:10.1126/science.273.5278.1073

[27] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*. ACM, New York, NY, USA, 122–126. doi:10.1145/36177.36194

[28] Abtin Molavi and Amanda Xu. 2026. Wisq. GitHub repository. Retrieved September 14, 2026 from https://github.com/qqq-wisc/wisq

[29] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 23. doi:10.1038/s41534-018-0072-4

[30] Abdul Rafae Noor, Dhruv Baronia, Akash Kothari, Muchen Xu, Charith Mendis, and Vikram S. Adve. 2025. MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations. *Proceedings of the ACM on Programming Languages* 9, PLDI, Article 198 (June 2025), 24 pages. doi:10.1145/3729301

[31] Aditya K. Prasad, Vivek V. Shende, Igor L. Markov, John P. Hayes, and Ketan N. Patel. 2006. Data Structures and Algorithms for Simplifying Reversible Circuits. *ACM Journal on Emerging Technologies in Computing Systems* 2, 4 (2006), 277–293. doi:10.1145/1216396.1216399

[32] QuiZX Developers. 2022. QuiZX: A Quick Rust Port of PyZX. GitHub repository. Retrieved September 14, 2026 from https://github.com/zxcalc/quizx

[33] Neil J. Ross and Peter Selinger. 2016. Optimal ancilla-free Clifford+T approximation of z-rotations. *Quantum Information & Computation* 16, 11&12 (2016), 901–953. doi:10.26421/QIC16.11-12-1

[34] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. doi:10.48550/arXiv.1711.04422 arXiv:1711.04422 [cs.PL]

[35] Vivek V. Shende, Stephen S. Bullock, and Igor L. Markov. 2006. Synthesis of Quantum-Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 6 (2006), 1000–1010. doi:10.1109/TCAD.2005.855930

[36] Vivek V. Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes. 2003. Synthesis of Reversible Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22, 6 (2003), 710–722. doi:10.1109/TCAD.2003.811448

[37] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Piscataway, NJ, USA, 124–134. doi:10.1109/SFCS.1994.365700

[38] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (2020), 014003. doi:10.1088/2058-9565/ab8e92

[39] Farrokh Vatan and Colin Williams. 2004. Optimal Quantum Circuits for General Two-Qubit Gates. *Physical Review A* 69, 3 (2004), 032315. doi:10.1103/PhysRevA.69.032315

[40] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing quantum-circuit optimizers. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 835–859. doi:10.1145/3591254

[41] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. 2025. Optimizing Quantum Circuits, Fast and Slow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 1. ACM, New York, NY, USA, 777–793. doi:10.1145/3669940.3707240

[42] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433

[43] Ganxiang Yang, Paige Raun, Runzhou Tao, and Ronghui Gu. 2026. Equality Saturation for Quantum Circuit Optimization. *Proceedings of the ACM on Programming Languages* 10, PLDI (2026), 174–199. doi:10.1145/3808254

[44] Ed Younis and Costin Iancu. 2022. Quantum Circuit Optimization and Transpilation via Parameterized Circuit Instantiation. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, Piscataway, NJ, USA, 465–475. doi:10.1109/QCE53715.2022.00068

[45] Charles Yuan. 2026. Cobble: Compiling Block Encodings for Quantum Computational Linear Algebra. *Proceedings of the ACM on Programming Languages* 10, PLDI (2026), 200–225. doi:10.1145/3808255 arXiv:2511.01736 [cs.PL]

A Full Algorithm

Alg. 3 is stated abstractly over an ordered set SubCirc of candidate subcircuits. This appendix defines the particular family used by our implementation and shows how to construct its members on the fly. We call these candidates *anchored connected subcircuits*; they are one efficient choice of SubCirc, rather than the set of all connected subsets of the circuit's dependency graph. We then state the pass together with the incremental construction of Sec. 4.2, in the form we implement.

Dependencies. Two gates are *dependent* when they act on a common qubit. Independent gates act on disjoint qubits.

Fix a circuit $C = g_1; \ldots; g_m$ and refer to its gates by their positions $1, \ldots, m$. Dependency is then a relation on positions: we write $i \bowtie j$ when the gates g_i and g_j are dependent. It is symmetric but not transitive, since two gates can each share a qubit with a third and none with each other.

Anchored connected subcircuits. Every set of positions S determines a *subcircuit* $C|_S$: the circuit made of the gates at those positions, in order.

Definition A.1 (Anchored connected subcircuit)**.** For positions $a \le b$, let $\bowtie_{a,b}$ be the restriction of $\bowtie$ to the gates of $g_a; \ldots; g_b$, and let $\bowtie^*_{a,b}$ be its reflexive transitive closure. Put

$$\mathrm{comp}(a, b) = \{\, j : a \bowtie^*_{a,b} j \,\}.$$

An *anchored connected subcircuit* of C is one of the form $C|_{\mathrm{comp}(a,b)}$ with $b \in \mathrm{comp}(a, b)$; the gate g_a is its *anchor* and g_b is its last gate.

The restriction is taken before the closure. A gate outside $g_a; \ldots; g_b$ never links two members, even if it would link them elsewhere in the circuit.

Example A.2. Take the three-gate circuit $C = X_0;\ H_1;\ T_1$, so $g_1 = X_0$, $g_2 = H_1$, and $g_3 = T_1$. Only g_2 and g_3 are dependent, both acting on qubit 1, so the anchored connected subcircuits of C are

$$X_0, \qquad H_1, \qquad T_1, \qquad H_1;\ T_1.$$

The gate X_0 joins nothing, being dependent on neither of the others.

Replacing g_2 by a CNOT gives $C' = X_0;\ \mathrm{CNOT}_{0,1};\ T_1$, whose middle gate is dependent on both of the others. The closure now links g_1 to g_3 through it, so $\mathrm{comp}(1, 3)$ is all of C'—even though X_0 and T_1 share no qubit.

In $C'' = X_0;\ H_2;\ \mathrm{CNOT}_{0,1}$, on the other hand, $\mathrm{comp}(1, 3) = \{1, 3\}$. The gate H_2 is dependent on neither of the others, so the subcircuit $X_0;\ \mathrm{CNOT}_{0,1}$ skips over it.

The last case is the general one: an anchored connected subcircuit need not occupy a contiguous span. A gate it skips is dependent on no member—otherwise the closure would have included it—so it commutes with all of them, and the members can be brought together without changing what the circuit computes. An anchored connected subcircuit can therefore be rewritten where it sits.

The pass in full. The scan keeps a set A of live anchors, each carrying the positions S_a of the subcircuit it anchors, and a set $\mathcal{R}$ of selected rewrites. Each subcircuit is considered as soon as it grows. We select a rewrite only when it is strictly smaller and no earlier selection has claimed its positions, so selections never overlap. One reconstruction pass at the end applies them all.

B Cyclotomic Ring

Canonicalization is the one step of Sec. 3.3 that cannot be performed literally over the ring. This appendix gives the construction we use instead.

Normal forms in the ring. Write $\omega = e^{i\pi/4}$ and let

$$R = \mathbb{Z}[\tfrac{1}{\sqrt{2}}, i] = \left\{ \frac{a + b\,\omega + c\,\omega^2 + d\,\omega^3}{\sqrt{2}^{\,k}} \;\middle|\; a, b, c, d \in \mathbb{Z},\ k \in \mathbb{N} \right\}$$

be the ring holding every amplitude of a Clifford+T circuit [16, 23]. The representation (a, b, c, d, k) is not unique, since numerator and denominator can be scaled together by $\sqrt{2}$. We reduce k as far as possible and store zero as $(0, 0, 0, 0, 0)$. Two ring elements are then equal exactly when their reduced tuples agree.

Alg. 5 gives the three operations. All of them are integer arithmetic on five-tuples: no floating point appears anywhere, so equality is exact.

Reduction is what makes the representation unique. Since $\sqrt{2} = \omega - \omega^3$, multiplying a numerator by $\sqrt{2}$ is the integer map $(a, b, c, d) \mapsto (b - d,\ a + c,\ b + d,\ c - a)$, and inverting it shows a numerator is divisible by $\sqrt{2}$ exactly when $a + c$ and $b + d$ are both even. Reduce divides out that factor for as long as it is present, so no two tuples of the same value survive with different k.

Addition needs a common denominator, so the element with the smaller k is scaled up by the same $\sqrt{2}$ map until the two agree; the numerators then add componentwise.

Multiplication follows from $\omega^4 = -1$, which folds the six degrees of the product back into four. Multiplication by ω alone is the special case $(a, b, c, d) \mapsto (-d, a, b, c)$: a cyclic shift with one sign flip, leaving k untouched. Canonicalization needs only that case.

Canonicalization over R. The only elements of R with modulus one are the eight powers of ω, and those are exactly the global phases relating two Clifford+T unitaries. Canonicalization is therefore a choice among eight candidates.

Order numerator tuples lexicographically in (a, b, c, d), writing $x \sqsubset y$ for the induced total order. The denominator plays no part: multiplying by ω leaves k untouched, so all eight candidates share one, and canonicalization is a statement about numerators alone. Alg. 6 computes $\mathrm{canon}_R(U)$.

The entry u is chosen by a rule that depends only on which entries are nonzero, so two unitaries differing by a global phase select the same position. The eight candidates are distinct because $u \neq 0$, which makes j^* unique. Multiplying by ω never changes k, so no candidate needs reducing and the comparison is a lexicographic test on four integers. The cost is eight shifts of a single entry plus one pass over the matrix, and every intermediate value stays in R.

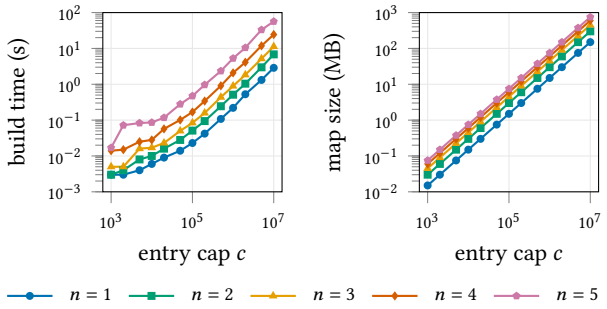


Figure 12. Cost of MURM synthesis over the IBM alphabet, in the layout of Fig. 7. *Left*: build time. *Right*: size. (Log-log plots.)

Theorem B.1 (Ring canonicalization)**.** *Let U and V be the unitaries of two Clifford+T circuits of equal dimension. Then*

$$\mathrm{canon}_R(U) = \mathrm{canon}_R(V) \iff U = e^{i\theta} V \text{ for some } \theta \in \mathbb{R}.$$

So canon_R decides the equality of Lemma 3.2 without ever leaving the ring, and we use it wherever Sec. 3.3 refers to canon: when fingerprinting a unitary and when confirming a candidate match.

C MURM Synthesis for Another Gate Set

Sec. 5.1 measures MURM synthesis over the fault-tolerant alphabet $\{x, z, s, s^\dagger, t, t^\dagger, h, \mathrm{CX}\}$. We synthesize a MURM by enumerating circuits over a chosen alphabet, so differences in the result may arise from the alphabet rather than the construction. We repeat the experiment exactly, changing only the gate set, over the IBM alphabet $\{x, z, s, s^\dagger, t, t^\dagger, \mathrm{SX}, \mathrm{CX}\}$—the basis of Appendix I, which replaces h with SX and leaves the rest alone. The same thirteen entry caps at the same five qubit bounds, five cold builds each, one process per build.

Fig. 12 is the result, drawn in the layout of Fig. 7. Both costs grow linearly in the cap at every qubit count, as before. Tab. 1 puts the two alphabets side by side.

D Sizes of Rewrites

Fig. 13 gives the size of every subcircuit the default configuration replaced.

The 135 benchmarks yield 12,310,782 rewrites, and the distribution is concentrated at the bottom. The median rewrite spans two gates, 68.5% span exactly two, and 73.1% span six or fewer; a rewrite removes 5.0 gates on average.

The rest of the mass sits in two spikes, at 16 gates and at 21. These are primarily artifacts of Cobble benchmarks, which are generated via the same programmatic process.

Algorithm 4 Linear-scan optimization with subcircuit construction

Require: a circuit $C = g_1; \ldots; g_m$, a MURM M, bounds q and ℓ
Ensure: a circuit equivalent to C with no more gates

$A \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
for $b \leftarrow 1$ **to** m **do**
  for all $a \in A$ with $i \bowtie b$ for some $i \in S_a$, in increasing order of a **do**
    if g_b acts only on qubits that $C|_{S_a}$ already uses **then**
      $S_a \leftarrow S_a \cup \{b\}$
    else
      $S_a \leftarrow \mathrm{comp}(a, b)$ ▷ g_b bridges
    if $C|_{S_a}$ touches more than q qubits or has more than ℓ gates **then**
      $A \leftarrow A \setminus \{a\}$
    else
      CONSIDER(S_a)
  $S_b \leftarrow \{b\}$; $A \leftarrow A \cup \{b\}$; CONSIDER($S_b$)
return C with $C|_S$ replaced by W' for every $(S, W') \in \mathcal{R}$

function CONSIDER(S)
  $W \leftarrow C|_S$; $W' \leftarrow M(\mathrm{canon}(U_W))$
  if W' is defined **and** $|W'| < |W|$ **and** no position of S is claimed **then**
    add (S, W') to $\mathcal{R}$ and claim every position of S

Algorithm 5 Arithmetic in $R = \mathbb{Z}[\frac{1}{\sqrt{2}}, i]$, on tuples (a, b, c, d, k) denoting $(a + b\omega + c\omega^2 + d\omega^3)/\sqrt{2}^k$

function REDUCE(a, b, c, d, k)
  if $a = b = c = d = 0$ **then**
    return $(0, 0, 0, 0, 0)$ ▷ one representative for zero
  while $k > 0$ **and** $a + c$ even **and** $b + d$ even **do**
    $(a, b, c, d, k) \leftarrow \left(\frac{b-d}{2},\ \frac{a+c}{2},\ \frac{b+d}{2},\ \frac{c-a}{2},\ k-1\right)$
  return (a, b, c, d, k)

function ADD($(a_1, b_1, c_1, d_1, k_1)$, $(a_2, b_2, c_2, d_2, k_2)$)
  while $k_1 < k_2$ **do** ▷ scale by $\sqrt{2}$ until the denominators agree
    $(a_1, b_1, c_1, d_1, k_1) \leftarrow (b_1 - d_1,\ a_1 + c_1,\ b_1 + d_1,\ c_1 - a_1,\ k_1 + 1)$
  while $k_2 < k_1$ **do**
    $(a_2, b_2, c_2, d_2, k_2) \leftarrow (b_2 - d_2,\ a_2 + c_2,\ b_2 + d_2,\ c_2 - a_2,\ k_2 + 1)$
  return REDUCE($a_1{+}a_2,\ b_1{+}b_2,\ c_1{+}c_2,\ d_1{+}d_2,\ k_1$)

function MUL($(a_1, b_1, c_1, d_1, k_1)$, $(a_2, b_2, c_2, d_2, k_2)$) ▷ $\omega^4 = -1$
  $a \leftarrow a_1a_2 - b_1d_2 - c_1c_2 - d_1b_2$
  $b \leftarrow a_1b_2 + b_1a_2 - c_1d_2 - d_1c_2$
  $c \leftarrow a_1c_2 + b_1b_2 + c_1a_2 - d_1d_2$
  $d \leftarrow a_1d_2 + b_1c_2 + c_1b_2 + d_1a_2$
  return REDUCE($a, b, c, d, k_1 + k_2$)

E What a Larger MURM Buys

Sec. 5.3 reports that a larger MURM—five qubits, 5,000,000 entries, $\ell = 40$—is a poor trade. Drawing it as Sec. 5.2 draws the superoptimizers shows that trade per benchmark.

The default is faster on every benchmark, with a median speedup of 41×: the $< 1\times$ band is empty, and the suite takes 210 seconds against 1,345. These are whole-process times, measured as in Sec. 5.2, so each invocation loads its own map: no benchmark finishes in under two seconds with the larger

Algorithm 6 Canonicalization over R, up to global phase

Require: a matrix U over R, in reduced tuples, with $U \neq 0$
Ensure: the unique representative of $\{\, \omega^j U : 0 \le j < 8 \,\}$

```
function CanonR(U)
    u ← the first nonzero entry of U in row-major order        ▷ any fixed order will do
    j* ← 0;   v ← u
    for j ← 1 to 7 do
        u ← (−u_d, u_a, u_b, u_c, u_k)                         ▷ u ← ωu: shift, one sign flip, k fixed
        if u ⊏ v then
            v ← u;   j* ← j
    return ω^{j*} U                                            ▷ the same shift applied to every entry
```

Table 1. MURM synthesis over the two alphabets at the largest cap swept, $c = 10^7$ per qubit count. *Slope* is the log–log exponent of build time against c, fitted over caps of 50,000 and up where startup no longer dominates; one is linear.

		ns per entry		slope	
n	entries	Clifford+T	IBM	Clifford+T	IBM
1	10^7	266	288	1.01	1.03
2	2×10^7	361	342	1.06	1.04
3	3×10^7	338	376	1.04	1.04
4	4×10^7	558	610	1.06	1.06
5	5×10^7	958	1131	1.00	1.03

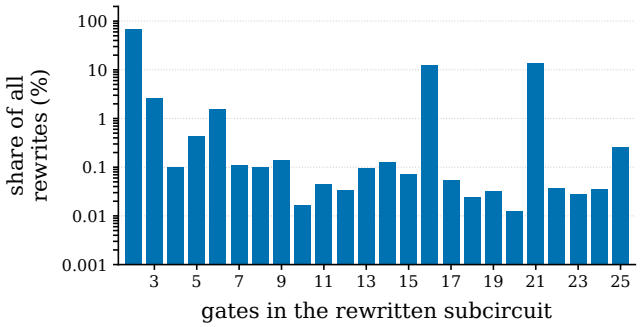


Figure 13. Sizes of the subcircuits TZAP rewrites, over the 12,310,782 rewrites the default configuration installs across the 135 benchmarks of Sec. 5.3. One count per rewrite, so the largest circuits contribute the most counts, and the share axis is logarithmic.

map, against 34 ms with the default, and the tool reports up to 1.3 s of that as map loading. The 16.2× in Sec. 5.3 counts optimization alone, with the map loaded once for the whole suite; the two figures differ by that fixed cost, which is itself part of what a larger map buys you. The larger map buys almost no additional reduction for that time. Fig. 15 shows the two series lying on top of one another, and the counts bear that out. Counting a benchmark as a match when the two reductions agree to within 0.5 pp, the configurations

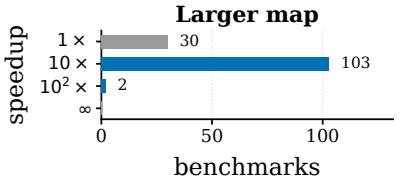


Figure 14. Speedup of the default MURM over the larger one, binned by order of magnitude, drawn as Fig. 10. Benchmarks on which both configurations finish inside a second are omitted as noise, so the bars total less than 135.

match on 113 of the 135 benchmarks on total gates, on 105 on two-qubit gates, and on 134 of 135 on non-Clifford rotations; 103 agree on all three at once. Where they differ the larger map usually leads—it wins 20 benchmarks on total gates and 27 on two-qubit gates—but not uniformly: the default still wins 2 and 3. In aggregate the gap all but disappears. Over the whole suite the larger map removes 91.13% of gates against the default's 91.07%, and 91.34% of two-qubit gates against 91.29%—differences of 0.06 and 0.05 pp—with the two indistinguishable on non-Clifford rotations. Its outputs are 0.7% smaller in total, for 6.4× the time. A wider window is not a superset of a narrower one, since the scan claims gates greedily and one long window taken early can crowd out two shorter ones that together removed more.

F Sweeping the MURM's Bounds

Appendix E compares the default MURM with one larger configuration. To isolate the effect of each bound and test alternatives to both configurations, we sweep their full Cartesian product. The sweep uses qubit bounds $n \in \{2, 3, 4, 5\}$; entry caps from 10,000 through 190,000 in increments of 20,000, then from 200,000 through 5,000,000 in increments of 200,000; and window lengths $\ell \in \{5, 10, \ldots, 45\}$. These 1,260 configurations each run over the 135 benchmarks of Sec. 5.2, for 170,100 optimizations.

Tab. 2 isolates one bound at a time and includes a joint change for comparison. The full sweep supports three conclusions.

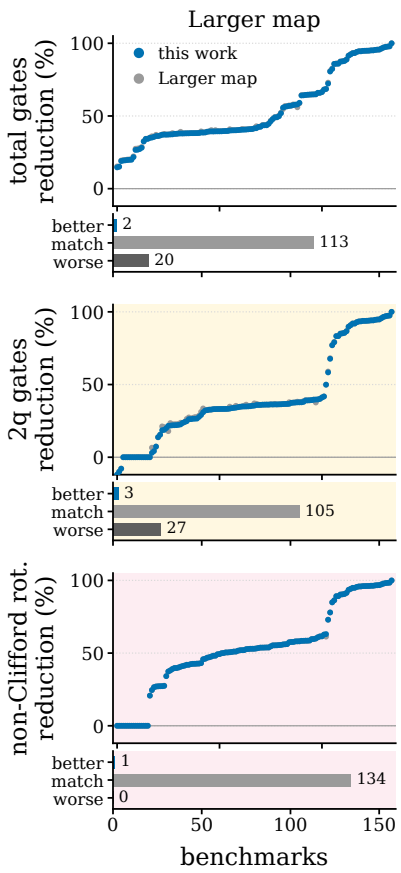


Figure 15. Reduction of the default MURM against the larger one, drawn as Fig. 10 except that the strip counts a benchmark as a match when the two reductions agree to within 0.5 pp. The two series lie on top of each other on every metric, which is the result: the gray is hidden because the larger map returns almost the same circuit.

Increasing one bound rarely improves reduction. Raising the qubit bound from three to five improves only one benchmark, on two-qubit gates. Raising the window length from 25 to 45 improves none. Raising the entry cap to $5{\times}10^6$ has the largest individual effect, improving total-gate reduction on 8 benchmarks and two-qubit-gate reduction on 15. Raising all three bounds improves 20 and 27 benchmarks on those metrics, respectively, which suggests that the bounds interact. None of these changes improves the reduction of non-Clifford rotations on more than one benchmark.

No swept configuration substantially improves on the default. The best configuration for total gates ($n = 5$, $c = 4.6{\times}10^6$, $\ell = 45$) wins on 21 benchmarks, draws on 112, and loses on 2. The best configuration for two-qubit gates ($n = 5$, $c = 5{\times}10^6$, $\ell = 45$) wins on 27, draws on 105, and loses on 3. Across the entire grid, no configuration improves the reduction of non-Clifford rotations on more than one benchmark. Thus, increasing these bounds yields little additional reduction on this benchmark suite.

Table 2. Effect of changing bounds from the default MURM ($n = 3$, $c = 200{,}000$, $\ell = 25$). Each cell reports *wins*/*draws*/*losses* for the modified configuration against the default over 135 benchmarks. A win or loss requires the gate reduction to differ by at least 0.5 percentage points; smaller differences are draws.

	total gates	two-qubit gates	non-Clifford rotations
raising one bound			
qubits → 5	0/135/0	1/134/0	0/135/0
entries → $5{\times}10^6$	8/124/3	15/116/4	0/134/1
window → 45	0/135/0	0/135/0	0/135/0
all three (Appendix E)	20/113/2	27/105/3	0/134/1
lowering one bound			
qubits → 2	1/91/43	0/84/51	0/98/37
entries → 10,000	1/119/15	1/124/10	0/132/3
window → 10	0/110/25	0/125/10	0/134/1
all three	1/81/53	1/80/54	0/98/37

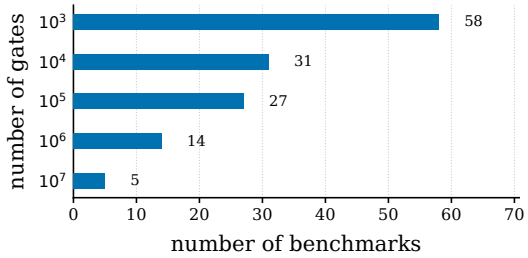


Figure 16. Benchmark sizes, one bar per order of magnitude.

The qubit bound matters most when reducing the configuration. Lowering only the entry cap to 10,000 loses 15 benchmarks on total gates; lowering only the window length to 10 loses 25. In contrast, lowering the qubit bound to two loses 43 benchmarks on total gates, 51 on two-qubit gates, and 37 on non-Clifford rotations. With all three reduced as shown in Tab. 2, the configuration still draws with the default on 81 benchmarks for total gates, 80 for two-qubit gates, and 98 for non-Clifford rotations.

G The Benchmark Suite

Fig. 16 gives the size distribution of the 135 benchmarks of Sec. 5.2. The suite spans four orders of magnitude, with 46 circuits past a hundred thousand gates and five past ten million.

H Speedup against Circuit Size

Fig. 17 plots the speedups of Fig. 8 against the size of the input circuit. The advantage grows with the circuit against

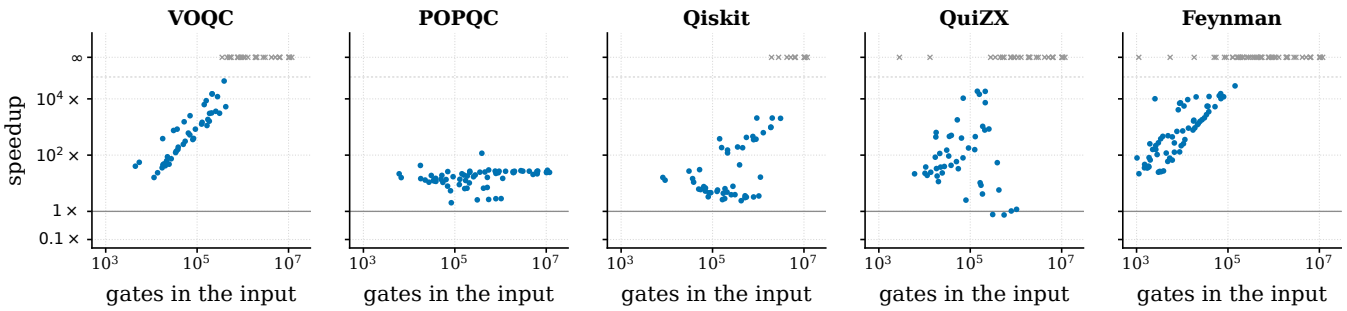


Figure 17. Fig. 8's speedups against the size of the input circuit. Runs that returned no circuit are drawn as crosses on the ∞ line. Benchmarks where both tools finish inside a second are omitted as noise.

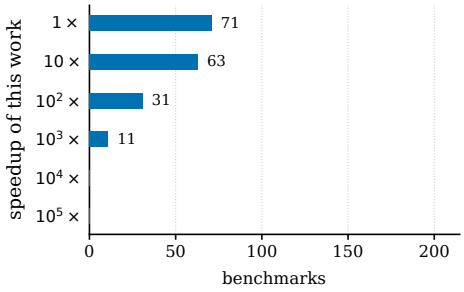


Figure 18. Speedup of TZAP over Quasar on Quasar's own 176 benchmarks, binned by order of magnitude. Quasar's time is where its search stopped improving, from its published timelines; the 1× row holds the benchmarks where it is faster, all of them small.

every tool but POPQC. From the benchmarks under ten thousand gates to those above a hundred thousand, the median speedup rises from 48.0× to 3,388× over VOQC and from 118× to 28,916× over Feynman; Qiskit climbs more gently, 14.5× to 30.7×. POPQC is nearly flat, 18.8× against 23.8×, and the one benchmark where it costs 117× our time is a single outlier.

I Quasar's Benchmarks

Here, we compare TZAP against Quasar using the 176 circuits in its evaluation and its reported results.

Both maps. Quasar spends hours on a circuit, so we report TZAP with both the small MURM of Sec. 5.2—three qubits, 200,000 entries, $\ell = 25$—and the large one of Sec. 5.3—five qubits, 5,000,000 entries, $\ell = 40$. Tab. 3 gives both. The figures draw the large map, which is the stronger of the two and still finishes the suite in four minutes against Quasar's twelve hours.

The two maps differ least where TZAP is strongest. On non-Clifford rotations they are indistinguishable—38.0% against 38.1%, both ahead on the same 108 benchmarks—so that result costs nothing. The large map earns its place on the other two metrics, nine more wins on total gates and seven on two-qubit gates, and it is the only configuration whose two-qubit total comes in under Quasar's: 15,934 against 16,154, where the small map gives 16,298. It costs 52× the time to do it, which is why Sec. 5.3 keeps the small map for the main suite. Both maps beat Quasar's twelve hours by orders of magnitude.

Table 3. TZAP against Quasar on Quasar's own 176 benchmarks, with each MURM. Reductions are medians; *wins* counts benchmarks where TZAP reduces more than Quasar at all.

	small MURM		large MURM		Quasar
	red.	wins	red.	wins	red.
total gates	37.1%	88	38.0%	97	37.0%
two-qubit gates	16.7%	35	19.4%	42	25.0%
non-Clifford rotations	38.0%	108	38.1%	108	24.2%
suite time	4.7 s		243 s		12.2 h
median speedup	737×		15×		

Results. Figs. 18 and 19 show the per-benchmark shape of the large map's row. The two tools are level on total gates. Quasar is the better tool on two-qubit gates, which is the objective its search minimizes. TZAP is the better tool on the non-Clifford rotations that dominate cost under fault tolerance, reducing them by a median of 38.1% compared with Quasar's 24.2%, and it is ahead there on 108 of the 176 benchmarks without losing one.

J How Each Tool Was Configured

Every tool runs on the same cluster, pinned to a single CPU model so the times are comparable. Each job takes one circuit, runs in the tool's own container with eight cores and 32 GB, and gets three hours. Quasar is the exception at 64 GB: its documentation assumes that figure and its runner caps address space there. At half that, it fails on most of the suite. Only POPQC is parallel—TZAP and the rest run single-threaded whatever they are allocated. Output circuits are measured afterwards by a single program, so no number here is a tool's own report.

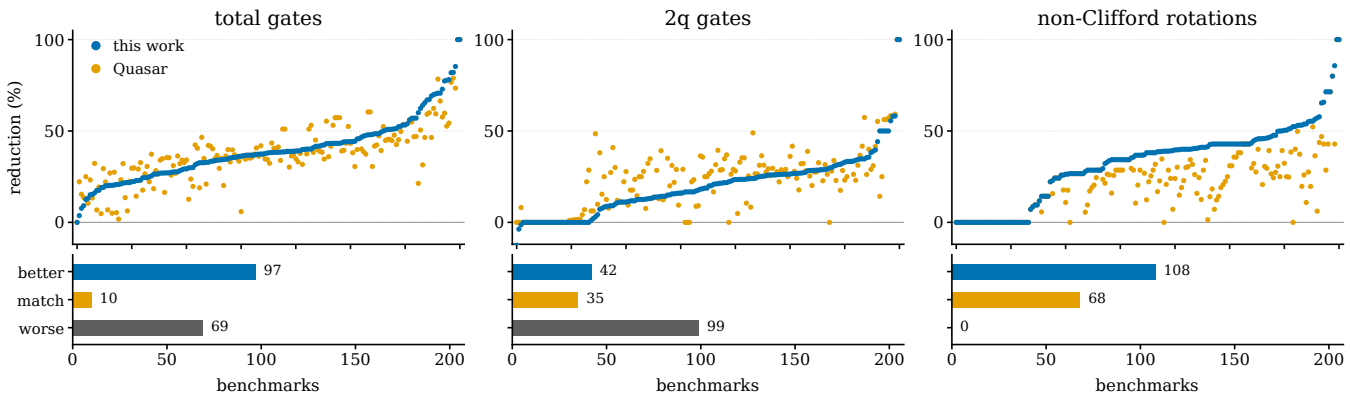


Figure 19. Reduction on Quasar's own 176 benchmarks, drawn as in Fig. 9. TZAP is run under `-ibm`, so both tools are measured in Quasar's own basis.

Each tool runs in its strongest configuration for this task. Where a tool ships a script that produced its published numbers, we run that script.

Qiskit 2.5.1. `transpile(circuit, basis_gates=BASIS, optimization_level=3, seed_transpiler=0)`, level 3 being its strongest preset, with the seed fixed because level 3 uses randomized passes.

VOQC. Its default configuration.

POPQC. The configuration shipped in its repository, with `n_threads` following the job's eight cores.

***QuiZX.*.** `quizx opt`, its default simplification.

Feynman. `feynopt -purecircuit -O3`. `-purecircuit` is required for soundness: without it the tool assumes the input starts in $|0\rangle$.

Quasar. We run its own `scripts/run_main.sh` configuration from the artifact root.

Wisq. `-mode opt` for optimization only; `-resynth NONE` makes it rewrite-only, as this comparison asks. We use total gate count as the optimization objective (`-optimization_objective TOTAL`). `-opt_timeout 10500` sits just under the job's three hours, so it writes its result before the kill. The gate set is chosen per circuit to match the input, as its documentation directs.