Reqomp: Space-constrained Uncomputation for Quantum Circuits

Anouk Paradis, Benjamin Bichsel, and Martin Vechev

ETH Zurich, Switzerland

Quantum circuits must run on quantum computers with tight limits on qubit and gate counts. To generate circuits respecting both limits, a promising opportunity is exploiting *uncomputation* to trade qubits for gates.

We present Reqomp, a method to automatically synthesize correct and efficient uncomputation of ancillae while respecting hardware constraints. For a given circuit, Reqomp can offer a wide range of trade-offs between tightly constraining qubit count or gate count.

Our evaluation demonstrates that Reqomp can significantly reduce the number of required ancilla qubits by up to 96%. On 80% of our benchmarks, the ancilla qubits required can be reduced by at least 25% while never incurring a gate count increase beyond 28%.

1 Introduction

Quantum computers will remain tightly resource-constrained for the foreseeable future, both in terms of available qubits and number of operations applicable before an error occurs. Running quantum programs hence requires compiling them to circuits with a limited qubit and gate count. A promising opportunity to achieve this goal is to exploit the need for *uncomputation* as an opening to trade qubits for gates.

What is Uncomputation? Just as classical programs, quantum circuits often leverage temporary values, called *ancilla variables*. Whereas classical programs can discard temporary values whenever convenient, temporary values in quantum circuits must be carefully managed to avoid side effects on other values through entanglement [1, §3]. Uncomputation is the process of preventing such side effects by reverting ancilla variables to state $|0\rangle$ after their last use, thus ensuring that they are disentangled from the remainder of the state. For instance, Fig. 1 shows a circuit implementing $CCCCH$: the H gate on qubit t with four control qubits o, p, q, and r. Fig. 1a uses three ancillae variables a, b, c, stored in the respective *ancilla qubits* u_0, u_1, u_2. The first ancilla a holds $o \cdot p$, b holds $o \cdot p \cdot q$, and c holds $o \cdot p \cdot q \cdot r$. We then use this last ancilla c to control the H gate on t, only applying H if all of o, p, q, and r hold state $|1\rangle$. In Fig. 1a, these ancilla variables are not uncomputed, and may result in unexpected interactions if this circuit is used as part of a bigger computation. They must therefore be uncomputed, as shown in Fig. 1b: the operations applied to each of them are reverted at the end of the circuit, ensuring that all ancilla qubits are reset to $|0\rangle$.

Anouk Paradis: anouk.paradis@inf.ethz.ch
Benjamin Bichsel: benjamin.bichsel@inf.ethz.ch
Martin Vechev: martin.vechev@inf.ethz.ch

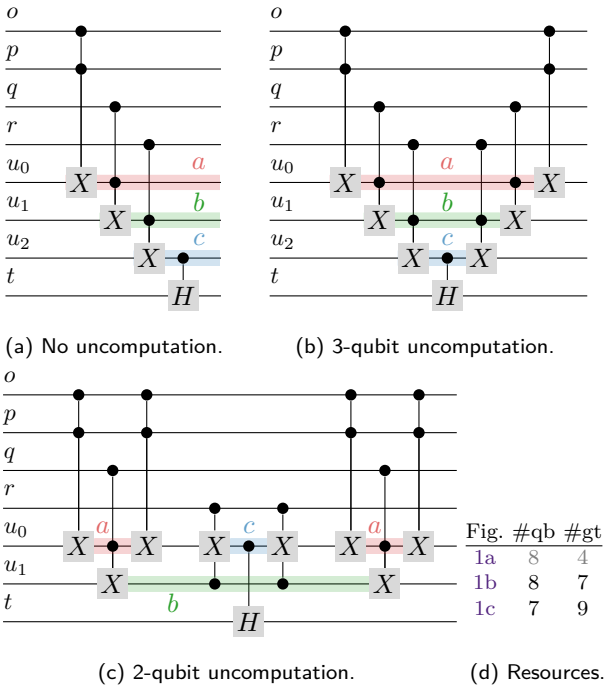


Figure 1: Two uncomputation strategies for $CCCCH$.

Reducing Qubits. After uncomputing an ancilla variable, its qubit can be reused by another ancilla variable, therefore reducing the overall number of qubits used by the circuit. Sometimes, it is even beneficial to uncompute an ancilla variable (too) early, allowing its qubit to be reused at the cost of later *recomputing* the ancilla variable when it is needed again.

Fig. 1b simply uncomputes ancilla variables in the reverse order of their computation, namely c–b–a. As no ancilla qubit can be reused, Fig. 1b requires 8 qubits and 7 gates overall (see Fig. 1d). Fig. 1c shows an alternative implementation of $CCCCH$ leveraging *recomputation*. It uncomputes ancilla variable a early, making its qubit u_0 free for the computation of ancilla

variable c. However, uncomputing b requires a again, forcing us to *recompute* it and subsequently uncompute it for a second time, at the cost of 2 additional gates. Overall, Fig. 1c thus trades qubits for gates compared to Fig. 1b, as summarized in Fig. 1d.

Correctness. Clearly, uncomputation is only useful if it correctly resets ancillae to $|0\rangle$ without modifying the remainder of the state. However, this is difficult to achieve, as uncomputing an ancilla may require some preprocessing on its controls, to ensure that they are in the right state (for details, see §7.1). Synthesizing the right gates to achieve uncomputation is thus a fundamental challenge, as evidenced by correctness issues in Square [2] which attempts to automate the placement of programmer-defined computation and uncomputation blocks (see §7.1).

Our Work. We present Reqomp, a method to automatically synthesize and place correct yet efficient uncomputation while respecting hardware constraints. Reqomp takes as inputs a quantum circuit C without uncomputation (such as Fig. 1a), its ancilla variables and a space constraint specifying the number of available ancilla qubits. If possible, Reqomp extends C to a circuit $\overline{C}$ which uncomputes all ancilla variables, using only the number of ancilla qubits specified.

To ensure Reqomp is both correct (whenever a circuit $\overline{C}$ is returned, it is a correct uncomputation of C) and practical (for most input circuits C and size constraints, it finds a circuit $\overline{C}$), we build it out of two distinct components. First, to ensure correctness, we introduce *well-valued circuit graphs*, which extend circuit graphs [1] (a graph representation of quantum circuits) by the new concept of *value indices* tracking the state of qubits. We further formalize evolveVertex, a method to safely introduce computation or uncomputation in these graphs. Second, to ensure practicality, we use multiple heuristics picking which steps of computation and uncomputation should be inserted (through calls to evolveVertex) and in which order.

Evaluation. Our experimental evaluation shows that Reqomp can significantly reduce the number of required ancilla qubits by up to 96% compared to the most relevant previous work Unqomp [1]. Many algorithms are amenable to a significant reduction: for 16 of 20 benchmarks, Reqomp can reduce the number of ancilla qubits by 25% compared to Unqomp, without incurring a gate count increase beyond 28%. For the remaining 4 examples, Reqomp strictly outperforms Unqomp, albeit by a smaller margin. In some cases, Reqomp achieves an impressive ancilla qubit reduction at very low cost: for one example, by 75% at the cost of increasing gate count by 17.6%.

Note that Unqomp already showed that manual uncomputation is both error-prone and less efficient than automatically synthesized uncomputation [1, §7].

Main Contributions. Our main contributions are:

- Well-valued circuit graphs, a graph representation of circuits allowing for accurate value tracking, and a method evolveVertex to modify them (§3);
- Reqomp, a method using well-valued circuit graphs to synthesize and place uncomputation in circuits under space constraints (§4);
- A correctness proof for Reqomp (§5);
- An implementation[1] and evaluation of Reqomp demonstrating it outperforms previous work (§6).

2 Background

We now introduce the necessary background on quantum computation.

Quantum States. We write the quantum state φ of a system with qubits p and q as:

$$\sum_{j=0}^{1}\sum_{k=0}^{1}\gamma_{j,k}\left|j\right\rangle_p\otimes\left|k\right\rangle_q=\sum_{l\in\{0,1\}^2}\gamma_l\left|l\right\rangle_{pq}\in\mathcal{H}_2,\quad(1)$$

where $\gamma_{j,k},\gamma_l\in\mathbb{C}$ and $\otimes$ is the Kronecker product. If φ factorizes into $\left(\sum_j\gamma'_j\left|j\right\rangle_p\right)\otimes\left(\sum_k\gamma''_k\left|k\right\rangle_q\right)$, p and q are *unentangled*, otherwise they are *entangled*. Whenever convenient, we omit $\otimes$ and write $|j\rangle$ instead of $|j\rangle_p$. We use latin letters $|j\rangle$ to denote computational basis states from the canonical basis $\{|0\rangle,|1\rangle\}$ and greek letters φ to denote arbitrary states.

Gates. A gate applies a unitary operation to a quantum state. Here, we only consider gates with a single target qubit in state φ and potentially multiple control qubits $C=\{c_1,...\}$ in state $|j\rangle$ for $j\in\{0,1\}^m$, mapping $|j\rangle_C\otimes\varphi$ to $|j\rangle_C\otimes\phi$, where the mapping from φ to ϕ may depend on the control j. Specifically, only the value of the target qubit may be changed, while control qubits are preserved. Note that this mapping can be naturally extended to superpositions (i.e., linear combinations as in Eq. (1)) by linearity. Further, because any circuit can be decomposed into single-target gates, not considering multi-target gates is not a fundamental restriction.

A gate is *qfree* if its mapping can be fully described by operations on computational basis states, i.e., if for control qubits C and target qubit t it is of the form

$$\left|j\right\rangle_C\left|k\right\rangle_t\mapsto\left|j\right\rangle_C\left|F(j,k)\right\rangle_t,$$

for $F\colon\{0,1\}^m\times\{0,1\}\to\{0,1\}$. For example, the NOT gate X, the controlled NOT gate CX, and the Toffoli gate CCX are qfree, while the Hadamard gate H and the controlled Hadamard gate CH are not qfree. Qfree gates are known to be critical for synthesizing uncomputation [1, 3, 4].

[1] Reqomp is publicly available at https://github.com/eth-sri/Reqomp.

Uncomputation. The task of uncomputation is to revert all ancilla variables in a circuit to their initial state $|0\rangle$, while preserving the circuit effect on the other variables. Formally, given a circuit C, we want to synthesize $\overline{C}$ which resets ancillae variables to $|0\rangle$ without affecting the remainder of the state:

Definition 2.1 (Correct Uncomputation, [1, 3])**.** *$\overline{C}$ correctly uncomputes the ancillae A in C if whenever*

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![C]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j |j\rangle_A \otimes \phi_j, \text{ then}$$

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![\overline{C}]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j |0\cdots 0\rangle_A \otimes \phi_j.$$

Here, $[\![C]\!]$ denotes the semantics of circuit C acting on a given input state. We refer to [1] for a more thorough introduction to uncomputation.

3 Circuit Graphs

As discussed in §1, Reqomp does not work directly on circuits, but instead relies on *well-valued circuit graphs*. In this section, we first intuitively introduce these graphs (§3.1) and the method evolveVertex to manipulate them (§3.2). Finally, we formalize the definition of well-valued circuit graphs and show how evolveVertex preserves their well-formedness (§3.3).

Circuit graphs were introduced and formalized in Unqomp [1]. We here extend them to precisely track qubits values. We discuss the differences between well-valued circuit graphs and Unqomp circuit graphs in §3.3.

3.1 Circuit Graph Intuition

As an example, we consider the circuit depicted in Fig. 2a. This circuit uses an ancilla qubit a to compute some output on qubit t, based on the value of qubit s. The circuit graph G corresponding to this circuit is shown in Fig. 2c.

Vertices and Edges. We first focus on the structure of the circuit graph G. G contains one *init vertex* per qubit (e.g., $s_{0.0}$ for qubit s), and one *gate vertex* per gate (e.g., $s_{1.0}$ for the first X gate on s). It also connects consecutive vertices on the same qubit by a *target edge*, e.g., $s_{0.0} \to s_{1.0}$. Further, as $a_{1.0}$ represents a CX gate controlled by qubit s, the circuit graph G also contains a *control edge* between the corresponding vertices on s and a: $s_{1.0} \bullet\!\!\to a_{1.0}$. Finally, the circuit graph G also contains *anti-dependency edges* to enforce correct ordering between otherwise unordered vertices. For example, $a_{1.0} \dashrightarrow s_{0.1}$ ensures that the second X gate on s (represented by $s_{0.1}$) can only be applied after the CX gate targeting a (represented by $a_{1.0}$). Anti-dependency edges can be reconstructed from the target and control edges: for any three vertices n, c, d such that $c \to d$ and $c \bullet\!\!\to n$, there is an edge $n \dashrightarrow d$ ensuring that n must be applied before d.

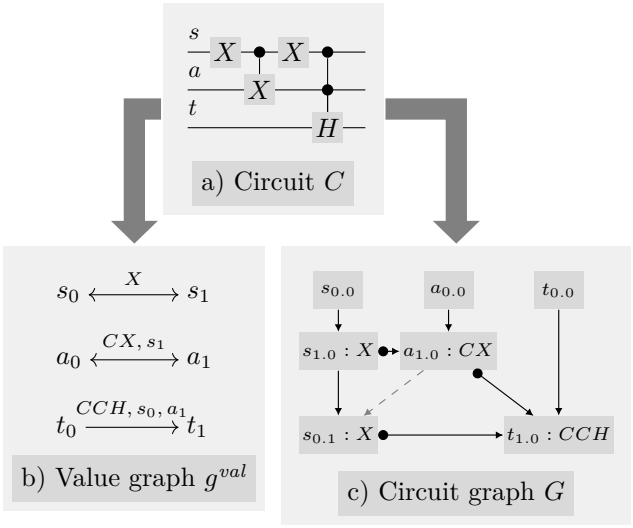


Figure 2: A Circuit C and the corresponding value graph g^{val} and circuit graph G

Valid Circuit Graphs. A circuit graph is *valid* iff it corresponds to a valid circuit. Most importantly, all valid circuit graphs must be acyclic[2]. Any valid circuit graph G can be converted into a circuit. The resulting circuit has one qubit per init vertex in G. We then pick any total order on the gate vertices of G that is consistent with the partial order induced by its edges, and add gates to the circuit following this total order. [1] showed that any of the circuits G can be converted to (depending on the choice of total order) have equivalent semantics. We hence define the semantics of a valid circuit graph G, denoted $[\![G]\!]$, as the semantics of any circuit it can be converted to.

Tracking Values. While the above construction follows Unqomp [1], we additionally introduce a new vertex naming convention to track qubit values. Specifically, each vertex (e.g., $s_{1.0}$) is identified by its qubit (here s), its *value index* (here 1) and its *instance index* (here 0). The value index is chosen such that intuitively, two vertices with the same qubit and value index hold the same value, even in the presence of entanglement. The instance index is used to ensure uniqueness of vertex names. For example, as X is self-inverse, the value on qubit s is the same in the very beginning of the circuit ($s_{0.0}$) as after applying the two X gates to s ($s_{0.1}$). More precisely, if the input state to the circuit is $|0\rangle_s \otimes \varphi$, after the two X gates on s have been applied, the final state is $|0\rangle_s \otimes \varphi'$ for some φ'. Reflecting this in the circuit graph, vertices $s_{0.0}$ and $s_{0.1}$ share the same value index 0.

[2]We recall the definition of valid circuit graphs from [1] in Def. C.1

Value Graph g^{val}. To track value indices during circuit graph construction and later during uncomputation, we rely on the value graph g^{val}, shown in Fig. 2b. It records for each qubit and value index the possible value transitions. g^{val} contains one init vertex per qubit but without an instance index, for example s_0 for qubit s. When encountering a new gate, for example the first X gate on qubit s, we pick a fresh value index for this qubit and extend g^{val}. The value graph g^{val} also records which operations can be safely uncomputed. For instance, as X is a qfree gate, the X on s_0 can be uncomputed: applying X on s_1 yields s_0 (note that X is self-inverse). We materialize this with the reverse edge $s_1 \to s_0$, giving:

$$s_0 \overset{X}{\longleftrightarrow} s_1$$

Similarly, the CX gate from a_0 with control s_1 yields a_1 (see Fig. 2b). Note that we do not specify the instance index of s_1: as any vertex on qubit s with value index 1 carries the same value, any of them can be used as a control. As CX is qfree, we also record in g^{val} the reverse edge from a_1 to a_0, with gate CX and control s_1. In contrast, the CCH gate on qubit r cannot be safely uncomputed as it is not qfree [3]. We therefore only record the forward edge from t_0 to t_1.

3.2 Modifying a Circuit Graph

We now discuss the core operation of Reqomp, the safe extension of a circuit graph in a stepwise manner through function evolveVertex.

evolveVertex. We show the algorithm for evolveVertex in Fig. 3a. As its name suggests, it is used to evolve vertices, i.e., to bring qubits from one value index to another. It uses the value graph g^{val} as a guide, and iteratively modifies a circuit graph $\overline{G}$. In Fig. 3, $\overline{G}$ is a copy of the circuit graph G from Fig. 2c, and its value graph g^{val} is shown in Fig. 2b. Note that Reqomp, and hence also evolveVertex, never modifies the circuit graph G corresponding to its input circuit. Instead, they both work on a new circuit graph $\overline{G}$, built by following G, as we will explain in §4.

Calling evolveVertex. evolveVertex takes as input three arguments. First, *qb* is the qubit on which we will insert the uncomputation. Second, *nVId* is the value index we will evolve this qubit to. The last argument is I, a set of qubits on which vertices are currently being added—this argument is needed to avoid infinite recursion (see also Lin. 2, discussed later). In Fig. 3b, we demonstrate the example call evolveVertex(a, 0, $\emptyset$), which uncomputes a single gate on qubit a: it will bring qubit a from its current value index 1 to 0. The argument $\emptyset$ indicates that no vertices on any other qubit are in the process of being added to $\overline{G}$.

Building the New Vertex. evolveVertex proceeds as follows. Lin. 3 gets the last vertex $\overline{last}$ on qubit a, that is the lowest one following target edges. Lin. 4 stores its value index in variable *oVId*. Here $\overline{last}$ is $a_{1.0}$ and *oVId* is 1. Lin. 5 then checks that *nVId*, the value index we want to add to the graph, here 0, can be reached in just one gate step. This is the case as a_0 is just one CX gate away from a_1, as evidenced by the edge $a_0 \xrightarrow{CX,b_0} a_1$ in g^{val} (see Fig. 2b). Lin. 8 then inserts a new vertex in $\overline{G}$ on qubit a with value index 0 and gate CX. As there is already a vertex $a_{0.0}$ in $\overline{G}$, it picks a new instance index, resulting in vertex $a_{0.1}$. Lin. 9 finally links it to its predecessor with a target edge and adds the resulting anti-dependency edge $t_{1.0} \dashrightarrow a_{0.1}$. This results in the second graph in Fig. 3b (ignoring the red edges $s_{1.0} \to a_{0.1} \dashrightarrow s_{0.1}$).

Adding Control Edges. To ensure $a_{0.1}$ indeed uncomputes $a_{1.0}$, we must control $a_{0.1}$ with qubits holding the same values as were used to control $a_{1.0}$. More precisely, $a_{0.1}$ should be controlled by vertices with the same qubit and value index as those controlling $a_{1.0}$. As the set *ctrls* contains exactly those qubits and value indices (see Lin. 6), Lin. 10 simply iterates over all controls c in *ctrls*. Through a call to the auxiliary function getAvailCtrl it then gets a (potentially new) vertex $\overline{c}$ (Lin. 11), which should have the same value index and qubit as c and be available as a control for $\overline{v}$ (that is adding the control edge $\overline{c} \bullet\!\!\to \overline{v}$, and resulting anti-dependency edges does not create a cycle in $\overline{G}$). Many implementations of getAvailCtrl are possible, each following different heuristics. The only restriction is that any modification to $\overline{G}$ must be done through a call to evolveVertex. This and the assertion in Lin. 12 are enough to ensure correctness of evolveVertex as we will discuss in §3.3.

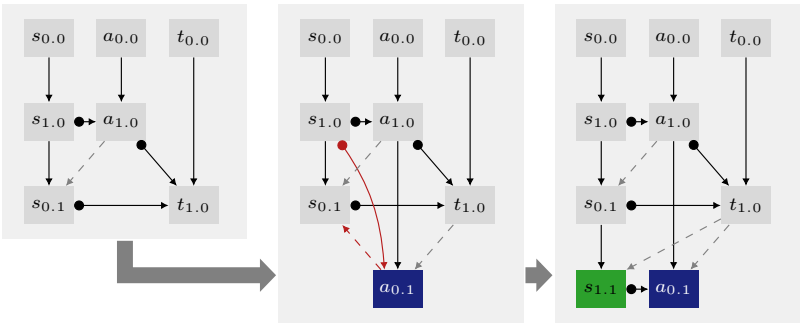
Choosing a Control. Let us manually follow the implementation of getAvailCtrl[3]. The only control in *ctrls* is s_1. We first check if an existing vertex in $\overline{G}$ with qubit s and value index 1 could be used. This is not the case, as using $s_{1.0}$ as control for $a_{0.1}$ would result in a cycle, as shown in the second graph in Fig. 3b. We must hence compute a new vertex on qubit s with value index 1. We do so by calling evolveVertex($s, 1, \{a\}$). Note how the set of qubits under construction contains a, as this is a recursive call within the computation of $a_{0.1}$. We can finally link $s_{1.1}$ to $a_{0.1}$ with a control edge, concluding the computation and yielding the third graph in Fig. 3b.

Avoiding Infinite Recursion. The assertion in Lin. 2 ensures that we never call evolveVertex recursively on the same qubit. This avoids infinite recursion where two qubits keep triggering recomputation of the other. To this end, we propagate the set I of qubits currently under construction through getAvailCtrl to potential recursive calls into evolveVertex (see Lin. 11).

[3] We describe this function in §4.3 and show it in Fig. 12.

```
1: func evolveVertex(qb, nVId, I)
2:     assert qb ∉ I
3:     last ← getVertex_G(qb, "last")
4:     oVId ← last.valIdx
5:     assert ∃qb_oVId --gt,ctrls--> qb_nVId in g^val
6:     define gt, ctrls as above
7:     assert WELL_VALUED_VERTEX
8:     v ← addVertex_G(gt, qb, nVId)
9:     addTargetEdge(last, v)
10:    for c ∈ ctrls do
11:        c ← getAvailCtrl(c, v, I ∪ {qb})
12:        assert CORRECT_CONTROL
13:        addControl_G(c, v)
14:    return v
```

(a) Function evolveVertex



(b) Using evolveVertex to uncompute $a_{1.0}$ in $\overline{G}$

Figure 3: evolveVertex algorithm and demonstration.

Modified Control. Here the uncomputation of ancilla a (introducing $a_{0.1}$ in $\overline{G}$) resulted in the modification of its control qubit s (introducing $s_{1.1}$). The circuit $\overline{C}$ corresponding[4] to $\overline{G}$ is hence not a correct uncomputation of C. While ancilla a has been correctly brought back to its initial value 0, the value of qubit s has been modified and does not hold the same value as it would in C. We will show in §4.3 how Reqomp notices and fixes such a value mismatch to ensure correct uncomputation.

3.3 Formalizing Value Indices

So far, we relied on an intuitive understanding of well-valued circuit graphs, and used it to build uncomputation for a circuit. Let us now formalize this intuition in Def. 3.1.

Definition 3.1 (Well-valued Circuit Graph)**.** *We say a valid circuit graph is well valued iff:*

(i) all vertex names are of the form $q_{s.i}$ where q is the name of the vertex qubit, s and i are natural numbers

(ii) there are no duplicate vertices

(iii) the init vertex on each qubit is named $q_{0.0}$ and for any $q_{s.i}$ in G, $q_{s.0}$ is in G

(iv) any gate vertex $q_{s.i}$ with $s > 0$ satisfies one of the following:

> ***(fwd)*** *$valIdx(pred(q_{s.i})) = valIdx(pred(q_{s.0}))$ and $q_{s.i}$ and $q_{s.0}$ have the same gate and same control vertices (up to their instance indices)*
>
> ***(bwd)*** *if we denote $s' = valIdx(pred(q_{s.i}))$, we have that (i) $valIdx(pred(q_{s'.0})) = s$, (ii) $q_{s.i}.gate$ is qfree and equal to $q_{s'.0}.gate^\dagger$, and (iii) both $q_{s.i}$ and $q_{s'.0}$ have the same controls (up to instance indices).*

[4]This is a slight simplification. Multiple circuits $\overline{C}$ may correspond to the circuit graph $\overline{G}$. They all have the same semantics, and will all be incorrect uncomputations of C.

Here, $pred(v)$ is the unique v' such that $v' \to v$ and $valIdx(v)$ is the value index of v. We now give some intuition of the last condition (iv). Case **(fwd)** corresponds to a (forward) computation, for instance $s_{1.0}$ and $s_{1.1}$ in the last graph in Fig. 3b. Here, case (fwd) ensures that $s_{0.0} \to s_{1.0}$ and $s_{0.1} \to s_{1.1}$ apply the same operation to the same starting state, i.e. in both cases qubit s holds the same value before the operation is applied. This is the case as both $s_{1.0}$ and $s_{1.1}$ have the same gate X, and their predecessors ($s_{0.0}$ and $s_{0.1}$) have the same value index 0. Case **(bwd)** corresponds to a (backward) uncomputation, for instance $a_{1.0}$ and $a_{0.1}$. Here, case (bwd) ensures that the operations $a_{0.0} \to a_{1.0}$ and $a_{1.0} \to a_{0.1}$ are exact inverses of each other. Specifically, it ensures that (i) $a_{0.1}$ and the predecessor of $a_{1.0}$ (here $a_{0.0}$) have the same value index 0, (ii) the gates of $a_{1.0}$ and $a_{0.1}$ are inverses of each other, and (iii) their controls ($s_{1.0}$ and $s_{1.1}$) have the same qubit and value index.

Preserving Values. Any circuit graph verifying this definition ensures the following: if a qubit is in some basis state at a value index, then if the qubit reaches the same value index at a later point in time, it will again be in this same basis state. Or, more formally: for every pair of vertices $q_{s.i}$ and $q_{s.i'}$, applying all gates G' between these vertices should preserve q, in the following sense:

$$\begin{matrix}\forall b \in \{0,1\}. \\ \forall \varphi \in \mathcal{H}_{n-1}.\end{matrix} \exists \psi \in \mathcal{H}_{n-1}. |b\rangle_q \otimes \varphi \xmapsto{[\![G']\!]} |b\rangle_q \otimes \psi, \quad (2)$$

where $\mathcal{H}_{n-1}$ denotes the set of quantum states over $n-1$ qubits. As we can write any state as a sum of computational basis states, Eq. (2) allows us to reason about any state. We show in App. C that any well-valued circuit graph ensures Eq. (2) (more precisely, Lem. C.3 implies this).

evolveVertex. In §3.2, we claimed that evolveVertex preserves the well-valuedness of a circuit graph. More precisely, if evolveVertex terminates without error, the resulting circuit graph $\overline{G}$ is still well-valued (assuming it was before the call). This is ensured through the two assertions in Lin. 7 and Lin. 12. We now specify

those assertions. The first, WELL_VALUED_VERTEX requires that one of the following conditions is satisfied:

(fwd) there exists i such that $qb_{oVId.i} \to qb_{nVId.0}$ is in $\overline{G}$ and gt and $ctrls$ correspond to the gate and controls of $qb_{nVId.0}$;

(bwd) there exists i such that $qb_{nVId.i} \to qb_{oVId.0}$ is in $\overline{G}$, gt is qfree and equal to $qb_{oVId.i}.gate^\dagger$ and $ctrls$ correspond to the controls of $qb_{oVId.0}$;

The second, CORRECT_CONTROL requires that $\overline{c}$ has the same qubit and value index as c, and that adding the control edge $\overline{c} \bullet\!\!\to \overline{v}$ (and all resulting anti-dependency edges) does not create a cycle in $\overline{G}$. Taken together, these assertions correspond exactly to the definition of a well-valued graph, and ensuring that it stays acyclic. Further, neither of those assertions refer to the value graph g^{val} and both avoid reliance on the function getAvailCtrl. This allows for a self-contained definition of well-valued graphs and simpler correctness proof, which we discuss in §5.

Contributions to Circuit Graphs. In the following, we briefly elaborate on the main differences between the circuit graphs introduced in Unqomp and our new notion of a well-valued circuit graph. Unqomp does not use value indices. Instead, uncomputation is built by adding to the circuit graph built from the input circuit an uncomputation vertex for each vertex on an ancilla. Correctness of the uncomputation then relies on this one computation vertex to one uncomputation vertex correspondence in the final circuit graph. This one to one correspondence fundamentally does not allow for recomputation, where three or more vertices may correspond to the same value. In contrast, we introduce the notion of value indices, and prove formally (see §5) that they accurately track values in qubits. We further introduce the notion of a value graph and the function evolveVertex, which leverages value indices to build correct computation and uncomputation in a circuit graph.

4 Reqomp

The previous section presented our notion of well-valued circuit graphs, and how they can be used to insert computation or uncomputation on any qubit. We now take a step back and present the complete Reqomp procedure, and how it leverages well-valued circuit graphs and the evolveVertex function to tackle the problem of ancilla variables uncomputation under space constraints. As mentioned in §1, Reqomp takes as input a quantum circuit and a number of available ancilla qubits. If successful, it returns a quantum circuit where all ancilla variables from the original circuit are uncomputed and all other variables are preserved, using only the number of available ancilla qubits.

Example Circuit. Fig. 4 gives an overview of Reqomp and applies it to an example circuit with five ancilla variables, a, b, c, d, and e, and two non ancilla variables r and t. We note that while this circuit does not implement a relevant algorithm, it allows showcasing the key features of Reqomp on a simple example.

Reqomp Workflow. Fig. 4 shows the steps performed by Reqomp, which we detail below.

First, Reqomp converts the circuit C into a circuit graph G and a value graph g^{val} (see Fig. 4b). Using this representation, Reqomp identifies the dependencies among ancilla variables in the circuit (see §4.1 and Fig. 4c), and uses them to derive an uncomputation strategy respecting the number of available ancilla qubits (see §4.2 and Fig. 4d). Reqomp then applies this strategy to build a new circuit graph $\overline{G}$ containing uncomputation (see §4.3 and Fig. 4e). Finally, Reqomp converts the resulting circuit graph into a circuit $\overline{C}$ (see §4.4 and Fig. 4f).

4.1 Identifying Ancilla Variables Dependencies

The first step of Reqomp is converting the input circuit into a circuit graph G, as we described in §3. Using this circuit graph G, Reqomp then identifies ancilla dependencies.

On the circuit graph G from Fig. 4b, Reqomp identifies all ancilla variables vertices (highlighted in red) and their dependencies (highlighted in blue), and extracts the ancilla dependencies shown in Fig. 4c. There, each vertex corresponds to an ancilla variable and each solid edge corresponds to a control edge among gate vertices between these respective ancilla variables. We will discuss the dotted edge shortly.

4.2 Deriving the Uncomputation Strategy

Based on the ancilla dependencies derived above, Reqomp will derive an *uncomputation strategy*.

What is an Uncomputation Strategy? The uncomputation strategy describes in which order the ancillae in the circuit should be computed and uncomputed, to satisfy the space constraints that were given as input, while minimizing the number of gates in the circuit. For instance Fig. 1 showcases two different such strategies. The first one, shown in Fig. 1b, is to compute ancilla a, then b, then c, then uncompute c, then b, then finally a. We typically write such a strategy as $a, b, c, c^\dagger, b^\dagger, a^\dagger$, where we write a to denote "computing ancilla a", and $a^\dagger$ to denote "uncomputing ancilla a". The second strategy, shown in Fig. 1c is $a, b, a^\dagger, c, c^\dagger, a, b^\dagger, a^\dagger$.

Partitioning Ancillae. The first step in deriving such an uncomputation strategy from the ancilla dependencies is to distinguish groups of ancillae variables that depend on each other; in other words, partitioning the ancillae according to their dependen-

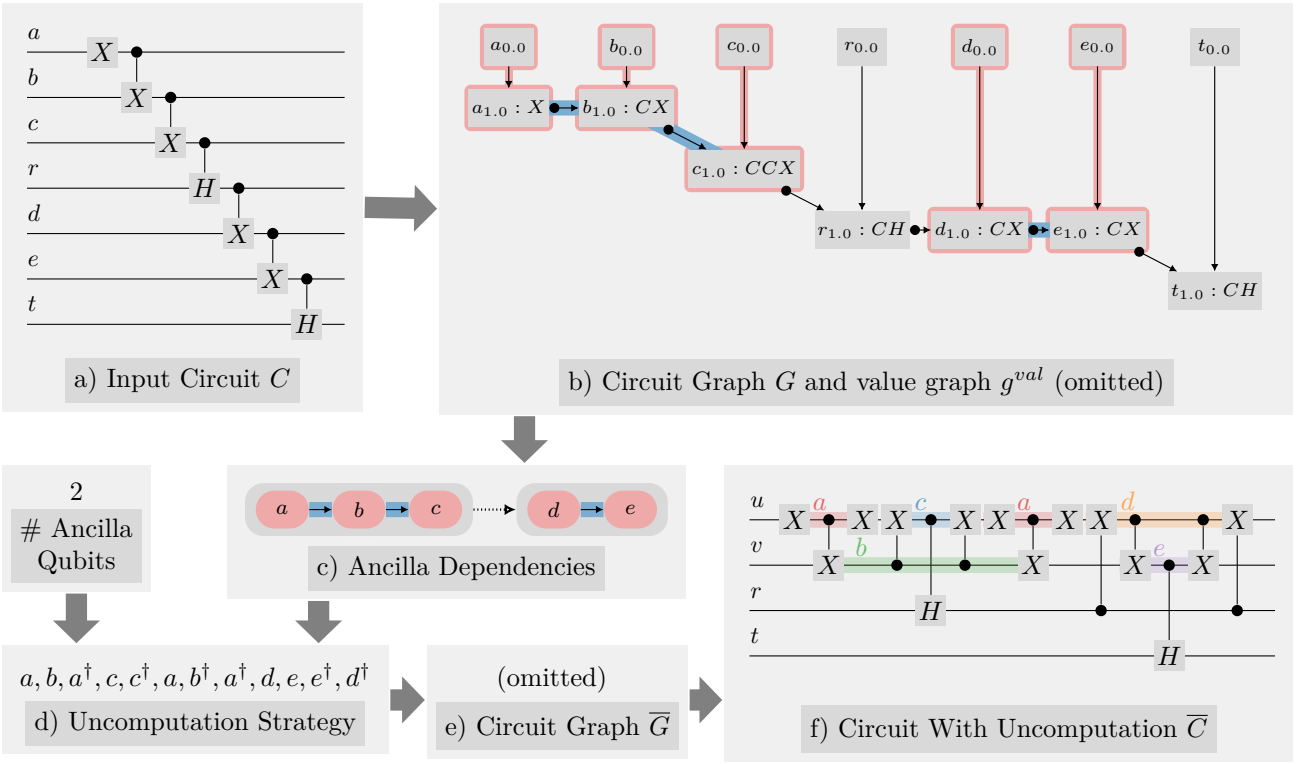


Figure 4: Overview of Reqomp

cies. More precisely, Reqomp identifies ancilla variables that do not interact with each other (i.e., lie in different connected components of the ancilla dependency graph), and can therefore be computed and uncomputed independently. For instance in Fig. 4c, ancillae variables a, b and c belong to the same connected component highlighted in dark gray, while ancillae variables d and e are in another component.

Why Partition Ancillae? Reqomp aims at balancing ancilla qubits and gates. For two ancillae that do not interact, such a trade-off is easy: we should always uncompute the first ancilla early, making its qubit available for the latter one. As the ancillae are independent, the latter one does not need the earlier one, so the early uncomputation will not induce extra gates, i.e., no recomputation is necessary. For instance, in Fig. 4c, ancillae $\{a, b, c\}$ and $\{d, e\}$ are independent. Therefore, it is strictly better to uncompute a, b and c before computing d and e, thereby reusing the physical ancilla qubits initially holding a, b and c for d and e. This is in contrast to ancillae that are part of the same partition. For instance, in Fig. 1, we saw that for the 3 linked ancillae a, b, c, uncomputing early yields a different trade-off than uncomputing late.

Strategy for a Connected Component Now that we have split the ancillae variables in two components, let us derive the uncomputation strategy for each of them. We first note that within each of the components in Fig. 4c, the ancilla variables exhibit a linear dependency. Formally, we say ancillae $a^1, \ldots, a^n$ are linearly dependent if all gates targeting a^i for $i > 1$ are only controlled by a^{i-1} and non-ancillae. This corresponds to a component that forms a simple path. In this case, we can derive an optimal uncomputation strategy (in terms of number of computation/uncomputation steps) using dynamic programming [5] [5]. For the first component ($\{a, b, c\}$) on at most two ancilla qubits, the following optimal strategy is found:

$$a, b, a^\dagger, c, c^\dagger, a, b^\dagger, a^\dagger. \tag{3}$$

For the second component ($\{d, e\}$), also on at most two ancilla qubits, the following optimal strategy is found:

$$d, e, e^\dagger, d^\dagger. \tag{4}$$

If within a component the ancilla variables are not linearly dependent, we abort the current procedure, and fall back on an alternative one, Reqomp-Lazy, which we describe in §4.5. We note that we avoid solving the general problem of finding an uncomputation strategy for any ancilla dependency, as it is P-SPACE complete [6].

[5] We show our implementation of this method in Fig. 13.

```
15: func ApplyingStrategy(stages)
16:     G̅ ← initGraph(G.qbs)
17:     for anc, fwd ∈ stages do
18:         if fwd then
19:             ancQb ← getFreeAncillaQubit()
20:             i ← freshInstanceId_G̅(anc, 0)
21:             addVertex_G̅(anc_{0.i}, ancQb)
22:             addEdge_G̅(getLastOnQb_G̅(ancQb) → anc_{0,i})
23:         if fwd then
24:             evolveVtxUntil(anc, max{s | anc_s ∈ g^val})
25:         else
26:             evolveVtxUntil(anc, 0)
27:         if s is last fwd stage in stages acting on anc then
28:             for t_s such that ∃t_s', t_s' --gate,..,anc_i--> t_s in g^val do
29:                 if t is not an ancilla and t_{s.0} ∉ G̅ then
30:                     evolveVtxUntil(t, s)
31:         if not fwd then
32:             freeAncillaQubit(ancQb)
33:     for v_{s.i} final vertex in G do
34:         if v not an ancilla then
35:             evolveVtxUntil(v, s)
36:     assertFullyEvolved()
37:
38: func evolveVtxUntil(var, to)
39:     from ← lastOnQb_G̅(var).valIdx
40:     steps ← getPath(var_from, var_to, g^val)
41:     for step ∈ steps do
42:         evolveVertex(anc, step, ∅)
```

Figure 5: Applying the uncomputation strategy. We assume $G, \overline{G}$, and g^{val} are globally available.

Combining Strategies After determining the optimal uncomputation strategy for each connected component, we must combine those strategies to yield our complete strategy. To this end, we determine in which order the components should be processed, ensuring that if an ancilla d transitively depends on ancilla c, c's component is processed before d's component (captured by edges ⋯▸ in Fig. 4c). In our case, we must process the component with ancilla variables a, b, c before the one with ancilla variables d, e, as d depends on c through the qubit r. Combining the respective strategies in this order, we finally get the complete uncomputation strategy shown in Fig. 4d. If ordering the connected components is impossible due to cycles, we again fall back to Reqomp-Lazy.

4.3 Applying the Uncomputation Strategy

We showed in the previous section how Reqomp derives the uncomputation strategy for a given circuit. Further, we have shown in §3.2 how we could use evolveVertex to insert computation or uncomputation in a circuit graph. However, there is a gap between the uncomputation strategy and evolveVertex: the former does not mention any non-ancilla variables, nor any value indices, which are a required argument of evolveVertex. Fig. 5 bridges this gap by showing how Reqomp translates the uncomputation strategy into a series of calls to evolveVertex, which will build a new circuit graph $\overline{G}$ with uncomputation.

A New Circuit Graph. Reqomp does not insert the uncomputation directly in G, the circuit graph built from the input circuit C. Instead, it builds a new graph $\overline{G}$ from scratch, adding computation and uncomputation on all qubits step by step. $\overline{G}$ is initialized in Lin. 16. Initially, it contains one init vertex for each non ancilla qubit in G. For the circuit graph G shown in Fig. 4b, this results in the following graph $\overline{G}$:

Qubits Allocation. The strategy consists of a sequence of stages, where each stage fully computes an ancilla or fully uncomputes it. Each stage is described by the ancilla it concerns, and a boolean *fwd* that is true for a computation step, and false for an uncomputation one. For instance the first stage in the strategy shown in Fig. 4d is (a, True), that is computing ancilla a. For a computation stage, the first step is to allocate a qubit (Lin. 19) and create a new vertex on this qubit (Lin. 20–21). This new vertex is then linked to the last vertex on the same qubit, if it exists, with a target edge in Lin. 22. This is typically the case if the qubit was previously used to compute and uncompute another ancilla. Further, at the very end of each stage, if it was an uncomputation stage, the qubit is marked as freed and therefore can be reused for later stages, see Lin. 32.

Detailed Steps for Ancilla (Un)computation. Now that the ancilla has been allocated a qubit if necessary, Reqomp computes the detailed computation or uncomputation steps the current stage requires, in Lin. 23–26. First, Reqomp decides on what is the objective value index for the current stage. If this is an uncomputation, it is 0, as the ancilla should be uncomputed back to its initial value. If this is a computation step, the ancilla should be computed until its maximum value index in G. For instance, for ancilla a, this maximum value is 1.

EvolveVtxUntil. To compute an ancilla to the objective value index chosen above, Reqomp relies on the function evolveVtxUntil, shown in Lin. 38. This function first determines what the current value index of the variable is in $\overline{G}$, that is to say what is the value index of the last vertex with qubit *var*. Using g^{val}, the function then determines the intermediate computation steps required to bring *var* from its current value index *from* to the objective one *to*. This is simply the shortest path in g^{val} from var_{from} to var_{to}. Those steps can then be applied in order, using the function evolveVertex, which we discussed in §3.2.

Non Ancilla Variables. We explained above how the uncomputation strategy can be detailed for non ancilla variables. Let us now describe how non ancilla variables are computed. This is done in two places. First, at the end of the last forward stage on an ancilla *anc*, anything controlled by this ancilla is computed.

More precisely, we want to compute all t_s, where t is a non ancilla variable and s a value index such that the computation of t_s is controlled by some anc_i, that is to say that there exists some edge $t_{s'} \xrightarrow{gate,..,anc_i} t_s$ in g^{val}. This is shown in Lin. 28–30, and the required computation steps are again computed and applied through the function evolveVtxUntil[6]. Non ancilla variables are also computed at the very end of the strategy, in Lin. 33–35. Here any non ancilla variable whose final value index in $\overline{G}$ is not the same as in G is computed to its final value index in G.

getAvailCtrl. When we introduced evolveVertex in §3.2, we mentioned that it relies on an auxiliary function getAvailCtrl to get the controls required for a vertex. Any heuristic can be used for this function, as long as it only modifies $\overline{G}$ through evolveVertex. Reqomp uses the following heuristic[7]. Suppose we need a control c_s (that is value index s on qubit c) to be used to control some vertex $\overline{v}$. We first find the latest (that is the lowest following target edges) vertex with this qubit and value index in $\overline{G}$. If this vertex is available for $\overline{v}$, that is adding a control edge from this vertex to $\overline{v}$ does not create a cycle in $\overline{G}$, we return it. If this vertex is not available, or if no such vertex exists, we recursively call evolveVertex to build a new vertex on qubit c with value index s from the latest vertex on qubit c.

Asserting Uncomputation is Complete. After the uncomputation strategy has been applied as described above, Lin. 36 performs a final check [8]. It asserts that all variables are fully evolved, either back to their initial state index 0 (for ancilla variables), or to their final state index in G (for non-ancillae). If not, Reqomp falls back to the alternative Reqomp-Lazy strategy (see §4.5).

4.4 Obtaining the Final Circuit

If the above check succeeded, the final step of the algorithm converts the circuit graph $\overline{G}$ to a circuit $\overline{C}$. During this step, we perform a generic post-processing optimization, previously discussed in [1]. Specifically it replaces in $\overline{G}$ all CCX gates which are later uncomputed by RCCX gates. While RCCX gates introduce an additional phase change, replacing pairs of CCX gates ensures that this phase change is also reverted.

As RCCX gates can be implemented more efficiently than CCX gates (the latter require more T gates), this can lead to a substantial efficiency improvement. This is particularly appealing in our setting, were we encounter many CCX gates, and most of them are uncomputed.

[6]As detailed in App. B.2 (Fig. 12), we may force extra steps in the computation to ensure some values are computed at least once for non ancilla variables.

[7]Fig. 12 shows this implementation of getAvailCtrl.

[8]Fig. 12 shows the implementation of assertFullyEvolved.

```
43: func Reqomp-Lazy()
44:     G̅ ← DeepCopy(G)
45:     U ← {q_{s.0} ∈ G̅ | q ancilla in G̅.qbs and s > 0}
46:     for v ∈ reverse(topologicalSort(U)) do
47:         l ← last_{G̅}(v.qubit)
48:         assert l.valIdx == v.valIdx
49:         evolveVertex(v.qubit, v.valIdx − 1, ∅)
50:     assertFullyEvolved()
51:     reuseAncillaRegisters()
```

Figure 6: Fallback algorithm closely following Unqomp [1].

We note that Unqomp could only apply this optimization to gates it had itself uncomputed, whereas Reqomp can also identify uncomputation that is already in place in the original circuit, by leveraging value indices.

The updated circuit graph $\overline{G}$ is then converted back to a circuit, as described in §3.1. For our example, this results in the circuit $\overline{C}$ in Fig. 4f. Importantly, the resulting circuit uses the same physical ancilla qubit to hold both a and c, saving one qubit at the cost of an extra uncomputation and recomputation of qubit a. The same physical qubit is also used to hold d, at no extra recomputation cost, as d does not depend directly on a.

4.5 Fallback procedure: Reqomp-Lazy

In general, uncomputation according to Def. 2.1 is not always physically possible [1, §6.2]. Because we cannot always achieve uncomputation, Reqomp applies heuristics to succeed as frequently as possible. However, we must accept that they may fail in some cases. First, the ancillae within a partition may not be linearly dependent, or the ancillae partitions may have cyclic dependencies. In such cases, Reqomp falls back to the heuristic Reqomp-Lazy, which we will describe shortly. Second, assertions in evolveVertex or assertFullyEvolved may fail, both in Reqomp and Reqomp-Lazy. In such cases, Reqomp returns an error. When this happens, it may indicate that no approach can achieve uncomputation, hinting at a possible implementation mistake or misconception by the programmer. If uncomputation is possible, but no available approach can synthesize it automatically, a programmer can always uncompute manually instead.

Overview. Reqomp-Lazy is inspired by Unqomp [1], but leverages the augmented circuit graphs and evolveVertex. In particular, Reqomp can uncompute and recompute controls for a vertex when they are not directly available. In contrast, Unqomp would have returned an error anytime this happens. We provide an example in Fig. 9 (§6).

Fig. 6 shows the algorithm Reqomp-Lazy. Lin. 44 initializes $\overline{G}$ with a copy of G, to be extended by adding vertices that perform uncomputation. Lin. 45 defines U as the set of all vertices to be uncomputed: it contains the first instance of each value index on

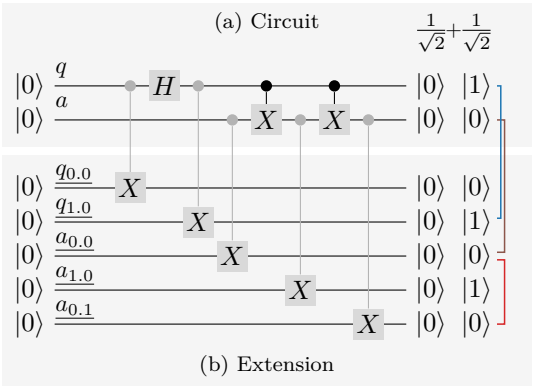


Figure 7: Intuition on the correctness of Reqomp.

each ancilla qubit. Then, Lin. 46–49 step through U in reverse topological order and revert all operations on ancillae one step at a time by calling evolveVertex (Lin. 49). Then, analogously to Reqomp, Lin. 50 asserts all qubits are fully evolved. Finally, as specified in the original version of Unqomp [1, §5.4], Lin. 51 allocates ancillae to the same physical qubits if their lifetimes do not overlap. The resulting circuit graph is then finally converted to a circuit, using the procedure detailed in §4.4.

Custom Control Strategy. While Reqomp-Lazy reuses evolveVertex, it uses a different implementation of getAvailCtrl. This new implementation aims at using controls that are as early (in terms of target edges) as possible, therefore keeping later controls available for later uncomputations. Specifically, to find a control c_s for a vertex $\overline{v}$, it finds the earliest vertex in $\overline{G}$ on qubit c and value index s that is available for $\overline{v}$. Recall that in contrast, we used the latest such vertex when using evolveVertex to apply an uncomputation strategy (see §4.3). If no such control vertex can be found, we recursively call evolveVertex to evolve the last vertex on qubit c until it has the state index s, just as we did in §4.3.

5 Correctness

We prove in App. C that Reqomp synthesizes correct uncomputation. In this section, we provide an intuition of this proof.

Value Index Assertions. The correctness of Reqomp relies on value indices. At the end of the algorithm (Lin. 36 when ApplyingStrategy succeeded, Lin. 50 when Reqomp falls back to Reqomp-Lazy), we assert that the last vertex on all ancilla qubits has value index 0, and that for any non-ancilla qubit, the value indices of the last vertex are the same for the original graph and the synthesized graph. Intuitively, this ensures that ancillae are reset to $|0\rangle$, while other qubits are preserved.

Correctness hence relies on the precise formal interpretation of value indices. Intuitively, we claim that two vertices on the same qubit with the same value index *hold the same value*.

Extended Circuits. To formally define this notion, we introduce the notion of an extended circuit. We conceptually *extend* a given circuit to allow us to compare the value of all vertices occurring in the circuit.

Fig. 7 exemplifies this by extending the example circuit in Fig. 7a, which applies an H gate and two controlled X gates to qubits q and a. Overall, the circuit in Fig. 7a yields state

$$\tfrac{1}{\sqrt{2}}\,|0\rangle_q\,|0\rangle_a + \tfrac{1}{\sqrt{2}}\,|1\rangle_q\,|0\rangle_a\,,$$

which we write in a column-by-column format in Fig. 7a (right).

Fig. 7b shows our extension of Fig. 7a, copying[9] the value of each vertex from the corresponding circuit graph to a fresh qubit. The name of these *copy qubits* is the same as their corresponding vertex but underlined, e.g., $\underline{q_{0.0}}$ holds the initial state of q, corresponding to vertex $q_{0.0}$.

Value Index. Intuitively, copy qubits with the same value index and qubit hold the same value. More precisely, if we write the state produced by the extended circuit as a sum of computational basis states, in each summand (with a non-null coefficient), copy qubits with the same value index and qubit hold the same value. For example, in every summand (i.e., column) of the final state in Fig. 7b, $\underline{a_{0.0}}$ and $\underline{a_{0.1}}$ hold value $|0\rangle$ (see red bracket in Fig. 7).

Similarly, each qubit holds the same value as its last copy qubit. For example, in every summand (i.e., column) of the final state in Fig. 7b, q and $\underline{q_{1.0}}$ both hold either value $|0\rangle$ or $|1\rangle$ (see blue bracket in Fig. 7).

In Lem. C.3 (App. C) we formally prove that these two facts hold for any well-valued circuit graph, as defined in Def. 3.1.

We further show in App. C that any circuit graph built with evolveVertex (Fig. 3a) is well-valued.

Final Values in the Extended Graph. The assertion in Lin. 36 (resp. Lin. 50) ensures that in the circuit graph $\overline{G}$ built by applying the uncomputation strategy (resp. the circuit graph $\overline{G}$ built by Reqomp-Lazy), the last vertex on all ancilla qubits has value index 0. Hence, those qubits hold the same value as the initial value of that qubit, i.e., $|0\rangle$. More precisely, consider a circuit graph G with ancilla qubits A and non ancilla qubits Q, and denote $\overline{G}$ the circuit graph after uncomputation. We then have that any summand in the final state after applying the extended version of $\overline{G}$ is of the form $|0...0\rangle_A \otimes |i\rangle_Q \otimes |...\rangle_{\overline{V}}$, where we use $\overline{V}$ to denote all the copy qubits in $E(G)$.

The assertions in Lin. 36 and Lin. 50 further check that the value indices of non-ancilla qubits

[9] Note that copying using a controlled X gate does not violate the no-cloning theorem.

match their respective last vertices in G. As we show more formally in App. C, this means that if the effect of the extended version $E(G)$ of G on some initial state can be written as

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![E(G)]\!]} \sum_{\substack{j\in\{0,1\}^{|A|}\\ k\in\{0,1\}^{|Q|}}} \gamma_{jk}\,|j\rangle_A \otimes |k\rangle_Q \otimes |\ldots\rangle_{\underline{V}},$$

then the effect of the extended version $E(\overline{G})$ of $\overline{G}$ on the same state can be written as:

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![E(\overline{G})]\!]} \sum_{\substack{j\in\{0,1\}^{|A|}\\ k\in\{0,1\}^{|Q|}}} \gamma_{jk}\,|0\cdots 0\rangle_A \otimes |k\rangle_Q \otimes |\ldots\rangle_{\underline{V'}},$$

where we denote $\overline{V'}$ the set of copy qubits in $E(\overline{G})$.

Circuit Graph Semantics. Importantly, the semantics of the unextended circuit follows straightforwardly from the semantics of the extended circuit. In Fig. 7, simply ignoring the rows from Fig. 7b yields the correct final state. If we similarly ignore the values of the copy qubits $\underline{V}$ and $\underline{V'}$ in the two equations above, we recover the correct uncomputation theorem, for circuits C and $\overline{C}$:

Definition 2.1 (Correct Uncomputation, [1, 3])**.** *$\overline{C}$ correctly uncomputes the ancillae A in C if whenever*

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![C]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j\,|j\rangle_A \otimes \phi_j, \textit{ then}$$

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![\overline{C}]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j\,|0\cdots 0\rangle_A \otimes \phi_j.$$

Multiple Graphs. Note that here we assumed that both G and $\overline{G}$ have the same effect, as they apply the same gates for the same value indices. Proving this formally requires extra work, done in Lem. C.4 (App. C).

6 Evaluation

We have evaluated Reqomp on an existing benchmark to answer the following research questions:

Q1 Circuit Efficiency: Can Reqomp create efficient circuits in terms of number of qubits and gates, while allowing to trade one for the other?

Q2 Usability: Is Reqomp fast and directly applicable to a wide range of circuits?

Implementation. We implemented Reqomp as a language extension of Qiskit, using Qiskit's built-in `AncillaRegister` type to mark ancilla variables in the circuit. As Qiskit, our extension is implemented in Python.

6.1 Benchmarks and Baseline

To evaluate Reqomp, we used the benchmark from Unqomp [1]. The first column in Table 1 summarizes the circuits in our benchmark, separated into "small" and "big" circuits. While the "small" circuits were taken directly from Unqomp, we have generated the "big" circuits by re-parametrizing the original circuits to yield bigger circuits. This allows us to demonstrate the Reqomp also performs well on larger circuits.

For completeness, we provide the exact parameters for each circuit in App. D, including the resulting circuit sizes.

Circuits. To provide an intuition on our benchmark, we explain selected circuits (see [1, §7.1] for details).

IntegerComparator takes a constant parameter n and multiple input qubits encoding a value v, and flips its output qubit if and only if $v \geq n$. MCX flips its output qubit if and only if all its input qubits are one. MCRY applies a rotation to its output qubit if and only if all its control qubits are one. PiecewiseLinearR applies a rotation $f(x)$ to its output qubit, where x is the value on its input qubits and f is piecewise linear. PolynomialPauliR works analogously, but for polynomial f. WeightedAdder takes as parameters a list of weights $\lambda_0, \ldots \lambda_n$ and outputs $\sum \lambda_i q_i$ where the q_i are the input values.

Selecting a Baseline. We provide a thorough overview of related work in §7. Of the many works discussed there, only four can take circuits as input: Square [2], Quipper [7], ReQWire [4] and Unqomp [1]. Of these, ReQWire can only verify uncomputation in circuits and not synthesize it[10], and we show in §7.1 that due to various shortcomings Square is not a viable option for uncomputation. This leaves only Quipper and Unqomp. As [1] showed that Unqomp generally outperforms Quipper, we choose Unqomp as our baseline.

Other works take as input boolean formulas (to be compiled to circuits) [8, 9, 10], focus on building uncomputation strategies without explaining how to apply them [11, 12], or do not compile to circuits [13].

6.2 Q1: Circuit Efficiency

We now discuss the efficiency of circuits produced by Reqomp in terms of qubits and gates and compare them to circuits produced by Unqomp [1].

Approach. For each circuit, we ran Reqomp targeting all possible number of ancilla qubits `nAncillaQubits`. We then recorded, for all calls that terminated without error, the number of qubits and gates of the resulting circuit (with uncomputation).

We note that Reqomp had to fall back to Reqomp-Lazy for circuits Multiplier and WeightedAdder, as

[10] It can synthesize circuits with uncomputation from a boolean formula but we focus here on its possibilities when working directly on circuits.

Table 1: Reqomp results when targeting a specific ancilla qubit reduction compared to Unqomp (e.g., -66.7 indicates a reduction by 66.7%). Gate counts are reported as compared to Unqomp (e.g., 70.5 indicates an increase by 70.5%). Columns **Max** and **Min** report the results for the most aggressive settings, respectively optimizing only for number of qubits and optimizing only for number of gates. Columns **-75%**, **-50%**, and **-25%** report the gate counts when achieving the respective ancilla qubit reductions. Entries "x" indicate that a given ancilla qubit reduction was not achieved.

	Ancilla Reduction						
	Max		**-75%**	**-50%**	**-25%**	**Min**	
Algorithm	anc (as %)	gates (as %)	gates (as %)	gates (as %)	gates (as %)	anc (as %)	gates (as %)
Small							
Adder	-66.7	70.5	x	39.2	15.7	-8.3	0.0
Deutsch-Jozsa	-50.0	40.9	x	40.9	20.4	0.0	0.0
Grover	-33.3	12.9	x	x	12.9	0.0	0.0
IntegerComparator	-63.6	47.1	x	36.1	11.6	0.0	-5.2
MCRY	-63.6	80.0	x	53.3	26.7	0.0	0.0
MCX	-60.0	55.2	x	46.0	27.6	0.0	0.0
Multiplier	0.0	-5.1	x	x	x	0.0	-5.1
PiecewiseLinearR	-50.0	7.5	x	7.5	2.5	0.0	-3.3
PolynomialPauliR	-33.3	9.5	x	x	9.5	0.0	0.0
WeightedAdder	0.0	-9.7	x	x	x	0.0	-9.7
Big							
Adder	-93.0	314.9	64.7	42.9	21.0	-1.0	0.0
Deutsch-Jozsa	-92.9	327.1	69.0	45.7	23.3	0.0	0.0
Grover	-50.0	37.8	x	37.8	16.2	0.0	0.0
IntegerComparator	-92.9	310.8	60.2	37.6	14.7	0.0	-6.7
MCRY	-96.0	515.4	75.3	50.2	25.1	0.0	0.0
MCX	-96.0	509.9	74.9	49.8	25.1	0.0	0.0
Multiplier	0.0	-5.4	x	x	x	0.0	-5.4
PiecewiseLinearR	-85.0	47.1	17.6	10.2	2.8	0.0	-3.8
PolynomialPauliR	-50.0	14.4	x	14.4	1.5	0.0	0.0
WeightedAdder	0.0	-8.0	x	x	x	0.0	-8.0

the ancilla dependencies of these circuits are not linear. While Reqomp-Lazy succeeds on these circuits and even outperforms Unqomp, it cannot offer multiple space-time trade-offs.

Results. Table 1 summarizes our results. Note that gate counts are expressed as a percentage of Unqomp gate counts. For all examples, using the maximum number of ancilla qubits (column **Min** as this is the minimal reduction) yields better results than Unqomp for 10 circuits, and equivalent results for the remaining 10 circuits. For example, Reqomp saves 5.2% of gates on circuit IntegerComparator, without requiring additional qubits. This is because Reqomp can identify uncomputation *already present in the original circuit*, allowing it to avoid unnecessary operations when uncomputing or recomputing an ancilla or even a control. Analogous effects occur for PiecewiseLinearR, WeightedAdder, and Multiplier, where the last two are handled by Reqomp-Lazy.

More importantly, Table 1 demonstrates that Reqomp can significantly reduce the number of ancilla qubits compared to Unqomp: by up to 96% for two examples, and by at least 25% for 16 out of 20 circuits. Importantly, this reduction comes at only a moderate cost in gate count, below 28% for qubit reductions of 25%. As most quantum computers are more limited in terms of qubits than gates, these trade-offs are highly favorable. Further, for some examples the reduction in qubits comes at almost no cost in gates: for PiecewiseLinearR, reducing by 75% the number of ancilla qubits only increases the number of gates by 17.6%.

Trade-Offs. To further demonstrate the gate count cost incurred by these reductions, Figs. 8a–8b show a more fine-grained visualization of the trade-offs between ancilla qubits and gate count.

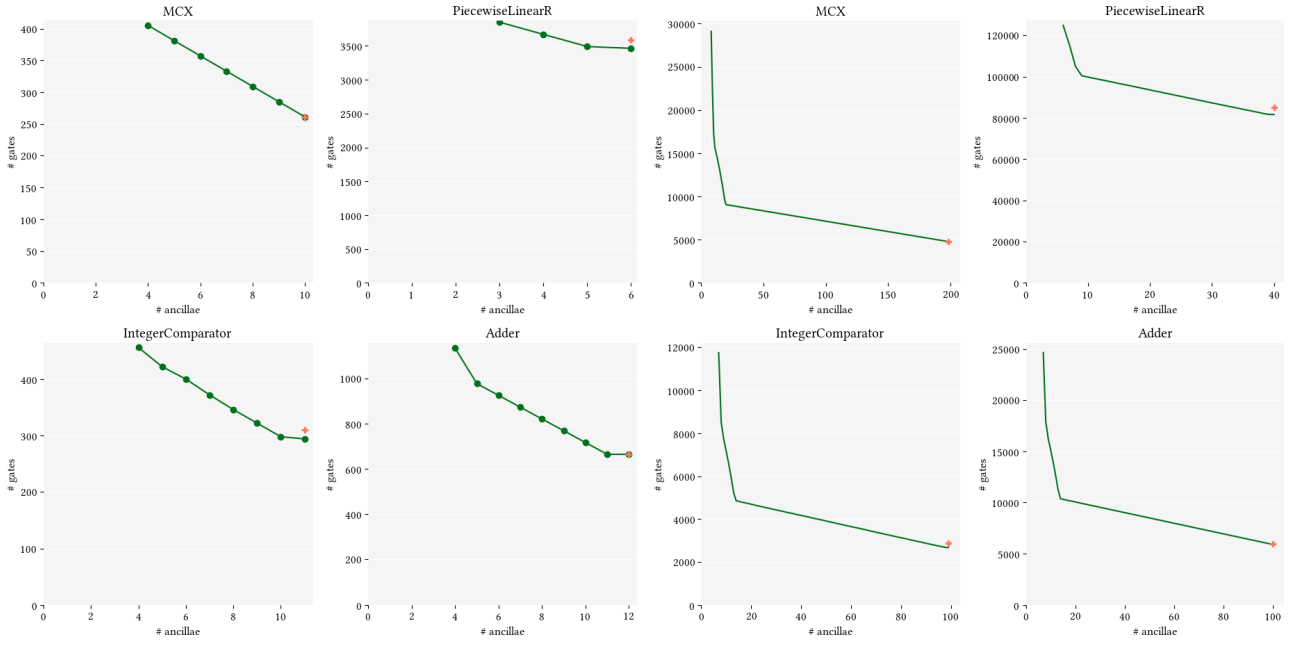
Overall, we immediately observe that on all circuits, reducing the number of available ancilla qubits can only increase (and never decrease) the gate count of the resulting circuit. However, the rate of this increase varies among the different circuits, as discussed next.

For some benchmarks such as PiecewiseLinearR (Figs. 8a–8b) and PolynomialPauliR (Table 1), Reqomp can drastically reduce the number of ancillae at almost no cost in terms of gates.

For other benchmarks such as MCX (Figs. 8a–8b) and MCRY (Table 1), Reqomp can still reduce the number of ancillae substantially, but at a significant cost in terms of gates. In such cases, the appropriate ancilla reduction depends on the available hardware—a programmer with access to Reqomp can then systematically select the right trade-off.

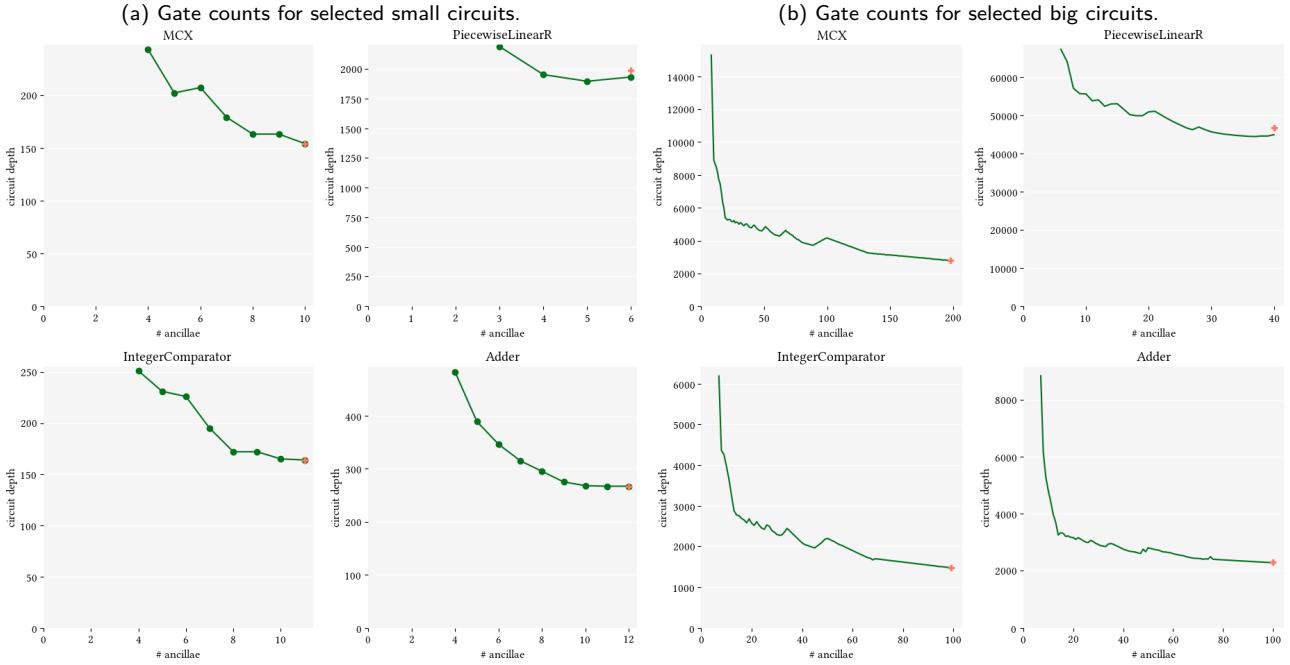
Other circuits fall somewhere between these two categories (Figs. 8a–8b and Table 1): Reqomp can reduce the number of ancilla qubits, at a non-negligible cost in terms of gates.

Very Small Number of Qubits. Fig. 8 further demonstrates that enforcing a very small number of ancillae typically increases the number of applied gates significantly. For instance, MCX with 200 controls can be implemented with only 8 ancilla qubits, but this requires a staggering 21 831 gates, compared to only 3579 when 200 ancillae are used.



(a) Gate counts for selected small circuits.

(b) Gate counts for selected big circuits.



(c) Circuit depth for selected small circuits.

(d) Circuit depths for selected big circuits.

Figure 8: Gate counts (a–b) and circuit depths (c–d) for given numbers of ancillae, using Reqomp (▬) and Unqomp (+).

Overall, we conclude that enforcing very small number of ancilla qubits is typically not a good approach.

Depth. For completeness, Figs. 8c–8d shows the trade-off between ancilla qubits reduction and circuit depth. As we do not optimize for circuit depth, reducing the number of ancillae sometimes yields *shorter* circuits. Still, overall, circuit depth behaves analogously to gate count, generally increasing for reduced ancilla qubits counts, at different rates depending on the circuit.

Interestingly, in some cases, we can reduce the ancilla count at almost no cost in circuit depth, even though there is a cost in gate count. For example, reducing ancillae from 99 to 25 on Adder only increases depth by 31%, even though it increases the gate count by 64%.

6.3 **Q2**: Reqomp Usability

We also investigated the usability of Reqomp, showing that it is both fast and directly applicable to many quantum circuits.

Reqomp Runtime. Our evaluation indicated that Reqomp is fast: it synthesized uncomputation for all circuits in Table 1 within five seconds.

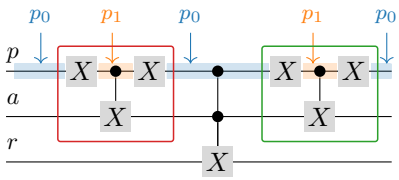


Figure 9: Gate requiring definition in Unqomp.

Furthermore, running Reqomp typically takes as much time as decomposing the resulting circuit to basic gates using Qiskit's built-in `decompose()` function. We hence believe that Reqomp can be integrated into the programmer's workflow without incurring a significant slowdown.

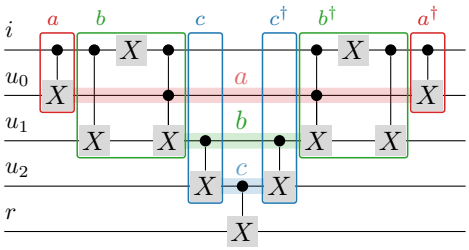
Applicability. Recall that even for a circuit where uncomputation is possible in principle, Reqomp may raise an error. We therefore investigated how frequently Reqomp succeeds in practice, comparing it to other tools:

	Qfree only	Unqomp	Reqomp
% examples covered	$\leq 50\%$	60%	100%

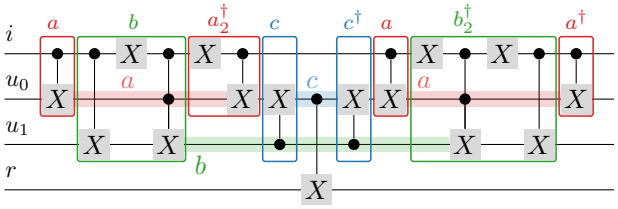
We find that Reqomp (with the fallback strategy Reqomp-lazy) finds a circuit with uncomputation for all input circuits. In contrast, Unqomp can only cover 60% of those circuits directly. We will explain shortly how we tweaked Unqomp to also cover the remaining 40%. Furthermore, only 50% of the circuits in our benchmark are purely classical, hence any tool that exclusively supports qfree gates can at most be used on 50% of the examples.

Unqomp Limitations. Unqomp can only handle 60% of the circuits in our evaluation directly, because it cannot accurately handle uncomputation that already occurs in the input circuit. Fig. 9 illustrates this on a circuit applying a $\overline{C}X$ gate (see red box on the left), where the bar over C indicates that the control is inverted. To invert the controls, the circuit applies an X gate to invert the control, and another X gate to restore the value of the control. To uncompute a it is hence necessary to track that after two X gates, p is back to its original value shown as p_0 in Fig. 9, and therefore applying a third X gate will bring its value to p_1 again, allowing to uncompute a. Value indices allow Reqomp to precisely track those value changes, and insert the uncomputation gates (in the green box on the right). In contrast, Unqomp fundamentally cannot allow for recomputation, as its correctness relies on each operation being computed and uncomputed exactly once. It further does not recognize uncomputation or recomputation already present in the original circuit. Therefore, in Fig. 9, Unqomp cannot recognize that the second X gate recovers the original value of p. Even if it did, it could not recompute p_1 to uncompute a.

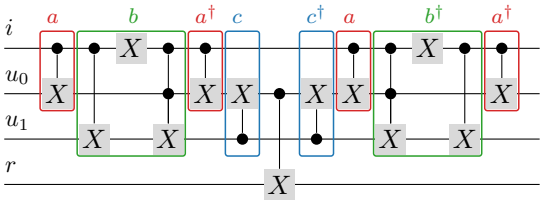
In our evaluation (Table 1), we bypassed this type of issue by defining the red block on the left as a custom gate controlled by p. Unqomp then never decomposes this new gate, assumes it keeps p constant, and places it to uncompute a. Unfortunately, this approach makes Unqomp harder to use, and in some cases makes the resulting circuit less efficient.



(a) Uncomputation using 3 qubits.



(b) Correct uncomputation using 2 qubits.



(c) Incorrect uncomputation using 2 qubits.

Figure 10: Circuit with varying uncomputation.

7 Related Work

We now discuss works related to Reqomp.

7.1 Square

Even though it cannot synthesize uncomputation code, Square [2] looks very closely related to Reqomp at first sight. Specifically, it presents "a compiler that automatically [places uncomputation] in order to manage the trade-offs in qubit savings and gate costs" [2, §1]. Unfortunately, Square suffers from various shortcomings that prevent a meaningful comparison to Reqomp.

Square Problem Statement. Square takes as input a program defining a qfree circuit (non qfree gates are not supported). In this program, each function consists of the three blocks *Compute* (indicating forward computation), *Store* (indicating computation of outputs), and *Uncompute* (indicating uncomputa-

tion). SQUARE then compiles this program to a circuit by arranging these blocks, possibly repeating blocks when recomputation is helpful.

SQUARE defines three different strategies for interleaving the blocks. Lazy (uncompute as late as possible), Eager (uncompute as early as possible), and finally SQUARE itself, using a custom heuristic. For the example $CCCH$ in Fig. 1, Lazy would correspond to the 3-qubit strategy shown in Fig. 1b and Eager to the 2-qubit strategy shown in Fig. 1c. We now present the main shortcomings of SQUARE.

Constant Compute/Uncompute Blocks. As mentioned in §1, the gates needed to uncompute an ancilla variable may depend on where this uncomputation occurs in the circuit. It is hence impossible to define fixed *Compute* and *Uncompute* blocks to be applied anywhere.

For instance, consider the circuit in Fig. 10a. It uses three ancilla variables a, b, and c to compute the output variable r from the input i. Fig. 10a highlights the Compute and Uncompute blocks SQUARE would consider, namely blocks a, b and, c for computation and blocks $a^\dagger$, $b^\dagger$, and $c^\dagger$ for uncomputation. Note how the value of qubit i is changed by block b, and restored later by block $b^\dagger$, ensuring that qubit i has the same value for the CX gate in block $a^\dagger$ as it had in block a. Now, if we want to save one ancilla qubit by uncomputing ancilla variable a early, we get the circuit shown in Fig. 10b. Here, when uncomputing a for the first time, the value of i has been changed in block b and is not yet restored. To correctly uncompute a in the block $a_2^\dagger$ (different from the block $a^\dagger$), it is hence necessary to restore i using an X gate before using it as a control to uncompute a. Similarly, block $b_2^\dagger$ must change the value of i again.

Not accounting for the above, SQUARE assumes that no matter its placement, uncomputation code can be kept unchanged. In particular, its eager strategy would use the Compute and Uncompute blocks from Fig. 10a, yielding Fig. 10c. This is clearly incorrect as this circuit has different semantics than the one in Fig. 10a. For example, for input $|0\rangle_i |0\rangle_t$, Fig. 10a produces state $|0\rangle_i |0\rangle_t$ while Fig. 10c produces state $|0\rangle_i |1\rangle_t$ (assuming ancillae are in state $|0\rangle$).

We note that SQUARE does not exclude such patterns—in fact its `little-belle` benchmark contains an analogous pattern. [11]

Incomplete Uncomputation. Besides only supporting fixed uncomputation code, SQUARE may also skip uncomputation of some ancilla variables. For some examples evaluated in [2], the implementation of the lazy strategy does not insert any uncomputation code at all, leaving all ancilla variables dirty, while the eager strategy uncomputes all of them. Specifically, we believe that the reported differences between strategies in the SQUARE publication ([2, Tab. III]) on the benchmarks[12] RD53, 6SYM, 2OF5, and ADDER4 are only due to leaving some ancillae dirty—as these benchmarks do not contain nested uncomputation, the order of uncomputation should not make a difference.

Additional Parameters. Finally, the implementation of SQUARE is inconsistent with the system described in [2]. Specifically, using the interface to specify *Compute* blocks requires providing 7 parameters, and some benchmarks evaluated in [2] also contain *Unrecompute* and *Recompute* blocks not mentioned in the publication [2]. Even though the authors provided us with brief explanations of these parameters on request, we could not confidently derive correct parameters for new benchmarks.

7.2 Purely Classical Circuits

Most works synthesizing uncomputation cannot handle non-qfree gates [4, 7, 8, 9, 14]. [13] It has already been established [1] that using such works on quantum circuits by separating out the qfree subparts typically yields inefficient circuits, and is sometimes even impossible.

In the following, we discuss works which only support qfree gates, and define a custom strategy allowing to trade qubits for gates. We have already discussed SQUARE in §7.1.

Boolean Functions. REVS [8, 9] translates irreversible classical functions to reversible circuits. It focuses on optimization possibilities during the translation from boolean functions to reversible circuits, but also offers an uncomputation strategy, however without the option of trading qubits for gates.

Similarly, [10] also translates boolean specifications to reversible circuit. While it introduces another uncomputation heuristic, it also cannot trade qubits for gates.

We expect that both of those strategies could be incorporated into Reqomp, possibly yielding more efficient circuits.

Pebble Games. Multiple works present uncomputation strategies for classical reversible computation, which can be reduced to solving *pebble games* [12]. Importantly, while pebble games operate on dependency graphs on values, Reqomp operates on quantum circuits. In particular, pebble games assume all values can be uncomputed, which is incorrect for non-qfree gates. Further, a direct translation of circuits to

[11] Benchmark `little-belle` is available at https://github.com/epiqc/Benchmarks/blob/master/bench/square-cirq/synthetic/little_belle.py. We note that different uncomputation strategies do not yield different results on it, as it does not contain gates modifying the output and hence is semantically equivalent to the identity.

[12] Available at https://github.com/epiqc/Benchmarks/tree/master/bench/square-cirq/application.

[13] [4] can verify uncomputation for non qfree circuits, but can synthesize it only for qfree ones.

such graphs would ignore repeated values, leading to issues analogous to Fig. 9. In contrast, conflating repeated values can lead to cyclic dependencies, which are not supported by pebble games.

Knill [5] provides an optimal yet efficient solution for linear dependencies. As most circuits we encounter in practice exhibit linear dependencies, Reqomp uses the same uncomputation strategy. Meuli et al. [11] suggest using a SAT-solver to handle arbitrary dependencies, which may be a possible extension of Reqomp.

7.3 Non-Qfree Circuits

We now discuss works offering uncomputation for non-qfree circuits.

Language Level. Quantum languages like Quipper [7] and Q# [13] offer convenience functions to automatically insert uncomputation. However, these functions are often tedious to use, and may insert incorrect uncomputation (see [1, §8] for details).

Silq [3] uses a type system to detect which variables can be safely uncomputed, but does not synthesize this uncomputation. Overall, none of those works can constrain the number of ancillae used.

Circuit Level. We are aware of only two works supporting uncomputation for non-qfree circuits. ReQWire [4] can only verify user supplied uncomputation (in the case of non-qfree circuits). Unqomp [1] allows to synthesize uncomputation for quantum circuits, but cannot trade qubits for gates. Further, as discussed in §6, it uses a notion of circuit graphs that does not allow to track qubit values and therefore is unable to uncompute directly many examples that Reqomp can handle.

8 Conclusion

We introduced Reqomp, a method to synthesize and place efficient uncomputation for quantum circuits with space constraints. Reqomp is proven correct and can easily be integrated into circuit based quantum languages such as Qiskit. We demonstrate in our evaluation that Reqomp is widely applicable and yields wide ranges of trade-offs in space and time, for instance allowing to generate tightly space constrained circuits by using only a few ancilla qubits.

References

[1] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. "Unqomp: synthesizing uncomputation in Quantum circuits". In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Pages 222–236. Association for Computing Machinery, New York, NY, USA (2021).

[2] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. "Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation". In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Pages 570–583. IEEE (2020).

[3] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. "Silq: A High-level Quantum Language with Safe Uncomputation and Intuitive Semantics". In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. Pages 286–300. PLDI 2020New York, NY, USA (2020). Association for Computing Machinery.

[4] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. "ReQWIRE: Reasoning about Reversible Quantum Circuits". Electronic Proceedings in Theoretical Computer Science **287**, 299–312 (2019).

[5] Emanuel Knill. "An analysis of Bennett's pebble game". Technical Report arXiv:math/9508218. arXiv (1995).

[6] Siu Man Chan, Massimo Lauria, Jakob Nordstrom, and Marc Vinyals. "Hardness of approximation in pspace and separation results for pebble games". In 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. Pages 466–485. (2015).

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. "Quipper: A scalable quantum programming language". In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Page 333–342. PLDI '13New York, NY, USA (2013). Association for Computing Machinery.

[8] Alex Parent, Martin Roetteler, and Krysta M. Svore. "Reversible circuit compilation with space constraints". Technical Report arXiv:1510.00377. arXiv (2015).

[9] Alex Parent, Martin Roetteler, and Krysta M. Svore. "REVS: A Tool for Space-Optimized Reversible Circuit Synthesis". In Iain Phillips and Hafizur Rahaman, editors, Reversible Computation. Pages 90–101. Lecture Notes in Computer ScienceCham (2017). Springer International Publishing.

[10] Debjyoti Bhattacharjee, Mathias Soeken, Srijit Dutta, Anupam Chattopadhyay, and Giovanni De Micheli. "Reversible Pebble Games for Reducing Qubits in Hierarchical Quantum Circuit Synthesis". In 2019 IEEE 49th International Sympo-

Symbol	Meaning
i	Imaginary unit
$o, p, q, r, t, u, \ldots$	Qubits
$a, a^{(0)}, b, c, d, \ldots$	Ancilla qubits
n	Number of qubits
C	Circuit
$G = (V, E)$	Circuit graph
$g^{val} = (V^{val}, E^{val})$	Value graph
$\overline{G} = (\overline{V}, \overline{E})$	Synthesized circuit graph
$v, v', \overline{v}, w, \ldots$	Vertex
s	State index
i	Instance index
$q_{s.i}, p_{0.1}, r_{1.0}, \ldots$	Vertex with explicit q, s, i
$c, \overline{c}$	Control vertex
$\varphi, \phi, \psi, \sum_j \gamma_j \lvert j\rangle$	Quantum state
$\gamma, \gamma', \overline{\gamma}, \lambda, \lambda', \overline{\lambda}, \ldots$	Complex coefficient (see above)
j, k, l	Variables to sum over (see above)
Q	Set of qubits
A	Set of ancilla qubits
R	Set of non-ancillae qubits (rest)
$F\colon \{0,1\}^{n+1} \to \{0,1\}$	Classical function defining qfree gate with n controls
U	(Unitary) gate (e.g., X or CX)
$[\![G]\!]$	Semantics of a circuit graph, as a function over quantum states
$E(G)$	Extended graph of G
$\underline{q_{s.i}}$	Qubit in $E(G)$ holding a copy of $q_{s.i}$
$(\!\lvert G \rvert\!)_p$	Coefficient for the projection p of the semantics of $E(G)$

Table 2: Notational conventions used throughout this work.

sium on Multiple-Valued Logic (ISMVL). Pages 102–107. (2019).

[11] Giulia Meuli, Mathias Soeken, Martin Roetteler, Nikolaj Bjorner, and Giovanni De Micheli. "Reversible pebbling game for quantum memory management". In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). Pages 288–291. IEEE (2019).

[12] Charles H. Bennett. "Time/Space Trade-Offs for Reversible Computation". SIAM Journal on Computing **18**, 766–776 (1989).

[13] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. "Q#: Enabling scalable quantum computing and development with a high-level dsl". In Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018New York, NY, USA (2018). Association for Computing Machinery.

[14] Matthew Amy, Martin Roetteler, and Krysta M. Svore. "Verified Compilation of Space-Efficient Reversible Circuits". In Rupak Majumdar and Viktor Kunčak, editors, Computer Aided Verification. Volume 10427, pages 3–21. Springer International Publishing, Cham (2017).

A Notational Conventions

Table 2 summarizes notational conventions used in this work.

B Algorithms

B.1 Partitioning

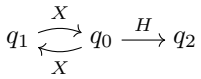
Fig. 11 shows the algorithm for partitioning the input graph.

B.2 Reqomp Convenience Methods

Fig. 12 shows convenience functions omitted from Reqomp.

GetPath. The function getPath used by evolveVertexUntil is shown in Fig. 12. For ancilla variables, it simply returns the shortest path between the two values in the value graph. However for non ancilla variables, it forces the computation of intermediate values that may not have been computed yet. This could happend for a circuit such as:

Here the value graph is:



Therefore, if we want to compute q_2 from q_0, the shortest path is simply $q_0 \to q_2$. However as H is not qfree, once q_2 has been computed, it can never be uncomputed again, and therefore, we can never compute q_1, which may be needed for some later computation. To correct this, we introduce q_1 (if it has not already been computed in $\overline{G}$) in the path, giving:

$$q_0 \to q_1 \to q_0 \to q_2$$

B.3 Linear Steps

Fig. 13 shows getLinearStrat. It is adapted from [5]: we added the *uncLast* parameters that allows us to apply it to ancillae only (that is we want all qubits to be computed once then uncomputed whereas the original algorithm did not uncompute the last qubit in the dependency line).

C Formal Correctness Proof

In the following, we provide a formal proof that Reqomp synthesizes correct uncomputation according to Def. 2.1.

```
52: func PartitionAncillae()
53:     G_anc ← AncillaDependencies(G)
54:     comps ← ConnectedComponents(G_anc)                    ▷ Subgraphs of G_anc
55:     G_ancdeps ← (comps, {})                               ▷ Each comp is a vertex of G_ancdeps
56:     for comp ∈ comps do
57:         for comp′ ∈ comps do
58:             if ∃ path from c ∈ comp to c′ ∈ comp′ in G then
59:                 addEdge_{G_ancdeps}(comp, comp′)
60:     assert G_ancdeps has no cycles
61:     return comps in topological order according to G_ancdeps
62:
63: func AncillaDependencies(G)                               ▷ Dependency graph on ancilla qubits
64:     V_a ← {v.qbit | v ∈ G, v.isAnc}
65:     E_a ← {(v.qbit, v′.qbit) | v •→ v′ ∈ G} ∩ V_a × V_a
66:     G_a ← (V_a, E_a)
```

Figure 11: Partitioning the uncomputation problem into independent subproblems.

```
67: func getAvailCtrl(c: Vertex, v̄: Vertex, I:Set[Qb])
68:     c̄ ← getVertex_Ḡ(c.qbit, "last", c.valIdx)
69:     if isAvailable_Ḡ(c̄, v̄) then
70:         return c̄
71:     else
72:         c̄′ ← getVertex_Ḡ(c.qbit, "last")
73:         c̄ ← evolveVertexUntil(c̄′, c.valIdx, I)
74:         return c̄
75:
76: func assertFullyEvolved()
77:                     ▷ Abort if values were evolved incorrectly
78:     for v ∈ final vertices in G do
79:         v̄ ← getVertex_Ḡ(v.qbit, "last")
80:         if v̄.isAnc then
81:             assert v̄.valIdx = 0
82:         else
83:             assert v̄.valIdx = v.valIdx
84:
85: func getPath(q: Qubit, from: int, to: int)
86:     p ← shortestPathInValueGraph(q, from, to)
87:     if q.isAnc then
88:         return p
89:     p′ ← []
90:     r ← [from + 1, to] if from < to else [to, from − 1]
91:     v ← from
92:     for i in r do
93:         if i ∉ p and q_{i.0} ∉ Ḡ then
94:             p′ ← p′ + shortestPathInValueGraph(q, v, i)
95:             v ← i
96:     p′ ← p′ + shortestPathInValueGraph(q, v, to)
97:     return p
```

Figure 12: Convenience functions leveraged by Reqomp.

C.1 Definitions and Helper Lemmas

We first define what we consider to be a valid circuit graph, following [1]:

Definition C.1 (Valid Circuit Graph)**.** *A circuit graph is valid iff*

(i) its init vertices have no incoming target edge while gate vertices have exactly one,

(ii) all its vertices have at most one outgoing target edge

(iii) its anti-dependency edges can be reconstructed from its control and target edges according to the rule discussed in §3.1,

(iv) the number of incoming control edges of each gate vertex v matches the number of controls of the gate of v

(v) G is acyclic.

In a valid circuit graph, we can define for any non init vertex n its predecessor $\text{pred}(n)$ as the only vertex m such that $m \to n$ (the target edge from m goes to n). We can also define for any qubit q its last vertex $\text{last}(q)$: it is the only vertex on qubit q with no outgoing target edge.

We now recall the well-valued circuit graph definition.

Definition C.2 (Well-valued Circuit Graph)**.** *We say a valid circuit graph is well valued iff:*

(i) all vertex names are of the form $q_{s.i}$ where q is the name of the vertex qubit, s and i are natural numbers

(ii) there are no duplicate vertices

(iii) the init vertex on each qubit is named $q_{0.0}$ and for any $q_{s.i}$ in G, $q_{s.0}$ is in G

(iv) any gate vertex $q_{s.i}$ with $s > 0$ satisfies one of the following:

(fwd) *$valIdx(pred(q_{s.i})) = valIdx(pred(q_{s.0}))$ and $q_{s.i}$ and $q_{s.0}$ have the same gate and same control vertices (up to their instance indices)*

(bwd) *if we denote $s' = valIdx(pred(q_{s.i}))$, we have that (i) $valIdx(pred(q_{s'.0})) = s$, (ii) $q_{s.i}.gate$ is qfree and equal to $q_{s'.0}.gate^\dagger$, and (iii) both $q_{s.i}$ and $q_{s'.0}$ have the same controls (up to instance indices).*

Vertices in a well-valued circuit graph are of the shape $q_{s.i}$, where we call s its value index (*valIdx* in the algorithms) and i its instance index. i is 0 for the first occurrence of q_s in the graph, but otherwise we only use its value to ensure uniqueness of the vertex names.

Due to the following lemma, it suffices to only consider valid and well-valued circuit graphs:

```
98: func getLinearStrat(cc: Qubit, n_qbits: int)
99:     sortedAncillae ← topoSort(cc)
100:    return [(sortedAncillae[i], b) for (i, b) ∈ stepsDP(|sortedAncillae|, n_qbits, false)]
101:
102: func stepsDP(n_anc: int, n_qbits: int, uncLast: bool)
103:    if return value was computed previously then
104:        return previously computed value                                   ▷ memoization
105:    if n_anc = 0 then
106:        return []
107:    if n_anc = 1 then
108:        if n_qbits = 0 then
109:            return null
110:        if uncLast then
111:            return [(0, true), (0, false)]
112:        else
113:            return [(0, true)]
114:    for m ∈ {1, ..., n_anc − 1} do
115:        if uncLast then
116:            toM ← stepsDP(m, n_qbits, false)                               ▷ 0 → m
117:            fromM ← [(i + m, b) for (i, b) ∈ stepsDP(n_anc − m, n_qbits − 1, true)]   ▷ m ⇄ n_anc
118:            cleanM ← [(i, ¬b) for (i, b) ∈ reverse(stepsDP(m, n_qbits, false))]       ▷ 0 ← m
119:        else
120:            toM ← stepsDP(m, n_qbits, false)                               ▷ 0 → m
121:            fromM ← [(i + m, b) for (i, b) ∈ stepsDP(n_anc − m, n_qbits − 1, false)]  ▷ m ⇄ n_anc
122:            cleanM ← [(i, ¬b) for (i, b) ∈ reverse(stepsDP(m, n_qbits − 1, false))]   ▷ 0 ← m
123:        steps_m ← toM + fromM + cleanM
124:    return arg min_{steps_m} cost(steps_m)
```

Figure 13: Optimal uncomputation strategy for linear graphs.

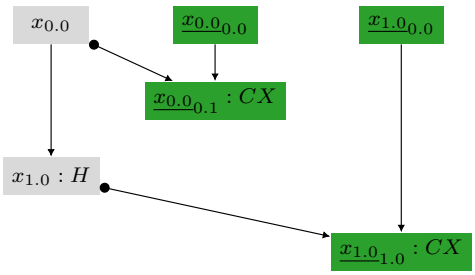


Figure 14: Extended graph example, copy vertices are shown in green.

Lemma C.1 (evolveVertex Correctness)**.** *For a valid and well-valued circuit graph G, any number of calls to evolveVertex results in a valid and well-valued circuit graph $\overline{G}$ such that (i) $\{q_{s.0} \in \overline{G}\}$ is a subset of $\{q_{s.0} \in G\}$ and (ii) for any $q_{s.0}$ in $G \cap \overline{G}$, it has the same gate and control vertices (up to instance index) in both graphs.*

Proof. By induction on the depth of calls to evolveVertex. □

We then define the extended graph $E(G)$ of a circuit graph G. Roughly, we want $E(G)$ to keep a copy of every vertex $q_{s.i}$ in G, saved on a fresh qubit $\underline{q_{s.i}}$. For a graph G with one qubit and two vertices, we show $E(G)$ in Fig. 14.

Definition C.3 (Extended Graph)**.** *For any circuit graph $G = (V, E)$, we define its extended graph $E(G) = (V_e, E_e)$ as follows:*

$$\begin{aligned} V_e =& V \cup \left\{ \underline{q_{s.i}}_{0.0}, \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \\ E_e =& E \cup \left\{ \underline{q_{s.i}}_{0.0} \to \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \\ & \cup \left\{ q_{s.i} \bullet\!\!\to \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \end{aligned}$$

For each $q_{s.i}$ in V, we have added a new qubit $\underline{q_{s.i}}$, with one init vertex and one gate vertex CX controlled by $q_{s.i}$. In the following we refer to those added qubits as $\underline{V}$. Note that while $\underline{q_{s.i}}_{1.0}$ is a vertex, $\underline{q_{s.i}}$ is qubit.

As the extended graph is a valid graph, it corresponds to a circuit and therefore its semantics $[\![E(G)]\!]$ is well defined. For a given input state φ to G, this allows us to define:

Definition C.4 (Projected Coefficients)**.** *For a fixed input state φ to the circuit graph $G = (V, E)$, we define the projected coefficients of G as the unique complex numbers $(\!|G|\!)_p$ such that:*

$$[\![E(G)]\!]\varphi \otimes |0...0\rangle_{\underline{V}} = \sum_{p:E(G).qbs \to \{0,1\}} (\!|G|\!)_p \, |p(G.qbs)\rangle_{G.qbs} \otimes |p(\underline{V})\rangle_{\underline{V}}$$

where $p(Q) = (p(q^{(1)}), ..., p(q^{(n)}))$ for qubits $Q = \{q^{(1)}...q^{(n)}\}$.

Using these coefficients, we can prove the following three lemmas. First, the semantics of the circuit graph G can be expressed in terms of its projected coefficients $(\!|G|\!)_p$:

Lemma C.2 (Projected Coefficients for Graph Semantics)**.** *For a circuit graph G we have:*

$$[\![G]\!]\varphi = \sum_{p:E(G).qbs\to\{0,1\}} (\!|G|\!)_p \,|p(G.qbs)\rangle$$

Proof. We can prove this by induction on the number of gates in G. □

Second, copies have consistent values. Specifically, for a given qubit q and valIdx s, all $\underline{q_{s.i}}$ hold the same value as $\underline{q_{s.i}}$, and the value of q is the same as the copy of the last vertex on q:

Lemma C.3 (Null Projected Coefficients)**.** *For a valid and well-valued circuit graph $G = (V, E)$ and $p : E(G).qbs \to \{0,1\}$, we have $(\!|G|\!)_p = 0$ if*

(i) $p(\underline{q_{s.i}}) \neq p(\underline{q_{s.0}})$ for some $q_{s.i}$, or

(ii) $p(q) \neq p(\underline{last(q)})$ for some qubit q.

Proof. We prove Lem. C.3 in App. C.3. □

Finally, if $(\!|G|\!)_p \neq 0$, it depends only on the gates used for the first computation of each $q_{s.0}$.

Lemma C.4 (Projected Coefficients Values)**.** *For a circuit graph G and $p : E(G).qbs \to \{0,1\}$, we have that if $(\!|G|\!)_p \neq 0$ then:*

$$(\!|G|\!)_p = \alpha_{p(q_{0.0}^{(0)}\cdots q_{0.0}^{(n)})} \prod_{\substack{q_{s.0}\in G\\ s\neq 0}} \gamma^{q_{s.0}}$$

Here the α describe the initial state:

$$\varphi = \sum_{k\in\{0,1\}^m} \alpha_k \,|k\rangle\,.$$

and the γ are gate coefficients defined such that $[\![g]\!]\,|c\rangle\,|t\rangle = \sum_{t'=0}^{1} \gamma^g_{t,c\to t'}\,|c\rangle\,|t'\rangle$ for a gate g and t in $\{0,1\}$ and $c \in \{0,1\}^m$. We have further shortened

$$\gamma^{q_{s.0}} = \gamma^{q_{s.0}.gate}_{p(\underline{pred(q_{s.0})}),p(\underline{ctrls(q_{s.0})})\to p(\underline{q_{s.0}})}$$

Proof. We prove Lem. C.4 in App. C.3. □

C.2 Main Proof

Using Lem. C.1–C.4, we can prove the correctness of Reqomp:

Theorem C.1 (Correctness)**.** *Have G a circuit graph built from a circuit with n qubits, of which m are ancilla variables. Without loss of generality, we can assume that those ancilla variables $A = (a^{(1)}, \ldots, a^{(m)})$ are the first m qubits of G. Let $\textsc{Reqomp}(G, A) = \overline{G}$. If*

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k\in\{0,1\}^m} \lambda_k\,|k\rangle_A \otimes \phi_k, \text{ then} \tag{5}$$

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![\overline{G}]\!]} \sum_{k\in\{0,1\}^m} \lambda_k\,|0\cdots 0\rangle_A \otimes \phi_k. \tag{6}$$

Note that this is an equivalent rewrite of Def. 2.1.

Proof. We first make the values of the non-ancilla qubits explicit, and denote $R = G.qbs\backslash A$. This allows us to rewrite Eq. (5) as :

$$[\![G]\!]\,|0\cdots 0\rangle_A \otimes \varphi = \sum_{\substack{k\in\{0,1\}^m\\ k'\in\{0,1\}^{n-m}}} \lambda_{kk'} \quad |k\rangle_A\,|k'\rangle_R \tag{7}$$

Similarly for $\overline{G}$ we can write:

$$[\![\overline{G}]\!]\,|0\cdots 0\rangle_A \otimes \varphi = \sum_{\substack{k\in\{0,1\}^m\\ k'\in\{0,1\}^{n-m}}} \overline{\lambda_{kk'}} \quad |k\rangle_A\,|k'\rangle_R \tag{8}$$

Note that here we use $\overline{\lambda}$ to refer to a coefficient in $\overline{G}$, and not to the complex conjugate of λ.

To prove the theorem, it is hence enough to prove that for all k',

$$\overline{\lambda_{kk'}} = \begin{cases} 0 & \text{if } k \neq 0 \quad \text{(i)}\\ \sum_k \lambda_{kk'} & \text{if } k = 0 \quad \text{(ii)}\end{cases}$$

To do so, we first identify Eq. (8) with Lem. C.2. This gives us that:

$$\overline{\lambda_{kk'}} = \sum_{\substack{p:E(\overline{G}).qbs\to\{0,1\}\\ p(\overline{G}.qbs)=kk'}} (\!|\overline{G}_p|\!) \tag{9}$$

The assertion at Lin. 36 in the Reqomp algorithm (Fig. 5) and Lem. C.3 then give that for any ancilla qubit $a^{(i)}$, if $p(a^{(i)}) \neq p(\underline{a^{(i)}_{0.0}})$, then $(\!|\overline{G}_p|\!)$ is null. As $\underline{a^{(i)}_{0.0}}$ copies the initial state of the ancilla, we then get that if $k \neq 0$, then $\overline{\lambda_{kk'}} = 0$, proving (i).

To prove (ii), we first note that Eq. (9) holds analogously for G, allowing us to derive the following. Here, we denote $V_0 = \{q_{s.0} \in V\}$. We then have for any k' in $\{0,1\}^{n-m}$:

$$\sum_{k\in\{0,1\}^m} \lambda_{kk'} = \sum_{k\in\{0,1\}^m} \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(G.qbs)=kk'}} (\!|G_p|\!) \tag{10}$$

$$= \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(R)=k'}} (\!|G_p|\!) \tag{11}$$

$$= \sum_{p_0:V_0\to\{0,1\}} \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(R)=k'\\ p_{|V_0}=p_0}} (\!|G_p|\!) \tag{12}$$

Using Lem. C.3, we have that for any $p_0 : V_0 \to \{0,1\}$, there is a unique $p_0^+ : E(G).qbs \to \{0,1\}$ such that $p_{|V_0} = p_0$ and $(\!|G|\!)_p$ is not known to be null. We can hence further rewrite Eq. (12):

$$\sum_{k\in\{0,1\}^m} \lambda_{kk'} = \sum_{\substack{p_0:V_0\to\{0,1\}\\ p_0(R_0)=k'}} (\!|G|\!)_{p_0^+}, \tag{13}$$

where we denoted $R_0 = \{\underline{q_{s.0}} \mid \text{last}(q) = q_{s.i}, q \in R\}$. Similarly, we write the same equation for $\overline{G}$ and $\overline{\lambda_{kk'}}$:

$$\sum_{k\in\{0,1\}^m} \overline{\lambda_{kk'}} = \sum_{\substack{p_0:\overline{V_0}\to\{0,1\} \\ p_0(\overline{R_0})=k'}} (\!|\overline{G}|\!)_{p_0^+} \tag{14}$$

Using that $\overline{\lambda_{kk'}}$ is null if $k \neq 0$, we finally get:

$$\overline{\lambda_{0k'}} = \sum_{\substack{p_0:\overline{V_0}\to\{0,1\} \\ p_0(\overline{R_0})=k'}} (\!|\overline{G}|\!)_{p_0^+} \tag{15}$$

Now, if $V_0 = \overline{V_0}$, as gates are the same for any $q_{s.0}$ in G and $\overline{G}$, Eq. (13) and Eq. (15) combined with Lem. C.4 and Lem. C.1 give us (ii).

If this is not the case, there must exist some $q_{s.0}$ in $V_0 \backslash \overline{V_0}$. For instance this could happen if G contains $q_{0.0} \to q_{1.0} \to q_{0.1}$, and $\overline{G}$ only contains $q_{0.0}$: it was not necessary to compute $q_{1.0}$ to reach the same final state as in G. The crucial observation is that this vertex $q_{s.0}$ gate is qfree. If q is an ancilla, this is clear as Reqomp would have raised an error otherwise. Indeed, Reqomp computes all ancillae (in Lin. 23–26) and check that they are all later uncomputed (Lin. 36). All operations on ancillae are hence uncomputed, and therefore their gate must be qfree (this is checked in Lin. 7). If q is not an ancilla, it means then $q_{s.0}$ must have been uncomputed in G (as the final state of non ancilla qubits in both graphs is the same). As qfree gates coefficients γ are either 0 or 1, having an extra qfree gates does not change the result of the sum in Eq. (13), concluding this proof. □

C.3 Proofs of Helper Lemmas

We now prove Lem. C.3 and Lem. C.4 by induction on the number of gates in G.

Proof. For a circuit graph G with no gates, an immediate induction on the number of qubits gives both lemmas.

Now suppose both lemma holds for any G with at most l gates. Now have $G' = (V', E')$ with $l+1$ gates. We can write G' as $G' = G \cdot q_{s.i}$ where $G = (V, E)$ has l gates and $q_{s.i}$ can be applied last in G'. To simplify notations, we assume $q_{s.i}$ has only one control vertex $c_{t.j}$. If it has 0 or more controls, the reasoning is analogous.

By definition, we know that:

$$[\![E(G)]\!]\varphi = \sum_{p:E(G).qbs\to\{0,1\}} (\!|G|\!)_p \, |p(G.qbs)\rangle_{G.qbs} \, |p(\underline{V})\rangle_{\underline{V}}$$

If we now apply $q_{s.i}$ and $\underline{q_{s.i}}_{0.1}$ (the CX gate copying $q_{s.i}$ to a new qubit) to one state of the sum above, we get for any $p : E(G).qbs \to \{0,1\}$:

$$[\![q_{s.i} \cdot \underline{q_{s.i}}_{0.1}]\!] \, |p(G.qbs)\rangle_{G.qbs} \, |p(\underline{V})\rangle_{\underline{V}} = \sum_{b\in\{0,1\}} \gamma^{q_{s.i}.gate}_{p(q),p(c)\to b} \, |p(G.qbs\backslash\{q\}), b\rangle_{G.qbs} \, |p(\underline{V}), b\rangle_{\underline{V'}}$$

Here b appears first as the value on the qubit q, and second as the value on the copy qubit $\underline{q_{s.i}}$.

As $E(G') = E(G) \cdot q_{s.i} \cdot \underline{q_{s.i}}_{0.1}$, we can use the above to compute $(\!|G'|\!)_{p'}$ for any $p' : E(G').qbs \to \{0,1\}$. We first notice that if $p'(q) \neq p'(\underline{q_{s.i}})$, then $(\!|G'|\!)_{p'} = 0$, giving us in Lem. C.3 (ii) for q. In the following, we hence only consider p' such that $p'(q) = p'(\underline{q_{s.i}})$. We then get:

$$(\!|G'|\!)_{p'} = \sum_{b\in\{0,1\}} (\!|G|\!)_{p'_{|E(G).qbs}[q\mapsto b]} \gamma^{q_{s.i}.gate}_{b,p'(c)\to p'(q)} \tag{16}$$

Using the recursion hypothesis, this immediately gives that if for any $q' \neq q$ if $p'(q') \neq p'(\underline{last(q')})$, then $(\!|G'|\!)_{p'} = 0$, giving us Lem. C.3 (ii) for $q' \neq q$. Together with the above, we hence get Lem. C.3 (ii).

We now work on proving both Lem. C.3 (i) and Lem. C.4. The recursion hypothesis gives us that for any $(q', s', i') \neq (q, s, i)$, if $p'(\underline{q'_{s'.i'}}) \neq p'(\underline{q'_{s'.0}})$, then again $(\!|G'|\!)_{p'} = 0$. We hence only need to establish that if $p'(q_{s.i}) \neq p'(q_{s.0})$ then $(\!|G'|\!)_{p'} = 0$, and the value of this coefficient when it is not null (that is to say Lem. C.4). To do so, we now consider p' consistent with what we have already proven, i.e., p' such that for any q' in $G.qbs$, $p'(q') = p'(\underline{last(q')})$ and for any $(q', s', i') \neq (q, s, i)$, $p'(q'_{s'.i'}) = p'(\underline{q'_{s'.0}})$.

We first notice that the recursion hypothesis gives that if $b \neq p'(\underline{pred(q_{s.i})})$, then $(\!|G|\!)_{p'_{|E(G).qbs} \oplus q\mapsto b} = 0$. Hence one of the summands in Eq. (16) is null:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{p'_{|E(G).qbs} \oplus q\mapsto p'(\underline{pred(q_{s.i})})} \gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(c)\to p'(q)}$$

Using the constraints on p', we can rewrite this to:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{p'_{|E(G).qbs} \oplus q\mapsto p'(\underline{pred(q_{s.i})})} \gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})}$$

Now we distinguish two cases. If this is the first occurence of q_s, that is to say $i = 0$, we immediately get Lem. C.3 (i), as $i = 0$. For Lem. C.4, by rewriting the equation above using that $i = 0$

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{p'_{|E(G).qbs} \oplus q\mapsto p'(\underline{pred(q_{s.0})})} \gamma^{q_{s.0}.gate}_{p'(\underline{pred(q_{s.0})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.0}})}$$

and using the induction hypothesis, we can conlude.

On the other hand, if $i \neq 0$, we again need to distinguish two possibilities: cases **fwd** and **bwd** in Item (iv) of Def. 3.1. We focus on the later case, as the first is simpler. We denote $q_{s'.i'} = \text{pred}(q_{s.i})$. We can hence rewrite:

$$\gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})} = \gamma^{q_{s.i}.gate}_{p'(\underline{q_{s'.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})}$$

As we know that G' is well-valued, we have that $q_{s.i}.gate = q_{s'.0}.gate^\dagger$ and that both gates are qfree. Generally, the coefficient for the reverse of a qfree gate g is

$$\gamma^{g^\dagger}_{t,c\to t'} = \gamma^{g}_{t',c\to t},$$

as a qfree gate coefficient can only be 0 or 1.

We can hence again rewrite the above coefficient as:

$$\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}.$$

Here, we used that $\left(g^\dagger\right)^\dagger = g$. Overall this gives us that:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{\substack{p'_{|E(G).qbs} \\ \oplus q\mapsto p'(\underline{pred(q_{s.0})})}} \gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$$

Now if $p'(\underline{q_{s.i}}) \neq p'(q_{s.0})$, let us prove that $(\!|G'|\!)_{p'}$ is null. If $(\!|G|\!)_{\substack{p'_{|E(G).qbs} \\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ is null this is clear, otherwise using Lem. C.4 we get that $(\!|G|\!)_{\substack{p'_{|E(G).qbs} \\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ contains $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$. We hence have that $(\!|G'|\!)_{p'}$ is a product which includes the factors $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ and $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$. As $q_{s'.0}.gate$ is qfree, one of those coefficients is null, and hence so is $(\!|G'|\!)_{p'}$.

Finally, if $p'(\underline{q_{s.i}}) = p'(q_{s.0})$, we get Lem. C.3 (i) trivially. If $(\!|G|\!)_{\substack{p'_{|E(G).qbs} \\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ is null, Lem. C.4 holds trivially. Otherwise, we use as above that $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ is in $(\!|G|\!)_{\substack{p'_{|E(G).qbs} \\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$. As $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ is 0 or 1, it is equal to its squared value, and the recursion hypothesis allows us to conclude. □

D Evaluation Values

We show in Table 4 the absolute numerical results on which the relative values in Table 1 are based. Table 3 further shows the exact parameters of each circuit used in our evaluation.

Table 3: Parameters for all examples in Table 1 and Table 4.

Algorithm	**Parameters**
Small	
Adder	12 qubits per operand
Deutsch-Jozsa	10 control qubits, with oracle MCX, returning true iff the value is 1111111111
Grover's algorithm	5 control qubits, with oracle MCX, returning true iff the value is 1111111111
IntegerComparator	12 control qubits, comparing to $i = 463$
MCRY	12 control qubits, with rotation angle $\theta = 4$
MCX	12 control qubits
Multiplier	5 qubits for each operand, and 5 for the result
PiecewiseLinearR	6 control qubits, function breakpoints are [10, 23, 42, 47, 51, 53, 63], slopes are [39, 32, 77, 27, 77, 4, 74] and offsets are [174, 40, 110, 163, 100, 185, 130]
PolynomialPauliR	5 control qubits, polynomial coefficients are [2, 2, 2, 2, 2]
WeightedAdder	10 controls qubits, values for sum are [0, 1, 1, 5, 2, 10, 4, 4, 9, 3]
Big	
Adder	100 qubits per operand
Deutsch-Jozsa	100 control qubits, with oracle MCX, returning true iff the value is 1111111111
Grover's algorithm	10 control qubits, with oracle MCX, returning true iff the value is 1111111111
IntegerComparator	100 control qubits, comparing to $i = 878234040205782925887743338143$
MCRY	200 control qubits, with rotation angle $\theta = 4$
MCX	200 control qubits
Multiplier	16 qubits for each operand, and 5 for the result
PiecewiseLinearR	40 control qubits, function breakpoints are [63870600266, 81180069351, 185076947411, 350818281077, 590566882159, 677977056232, 866030640015, 949186564661, 978976427282], offsets are [46, 59, 40, 48, 54, 67, 21, 71, 22] and coefficients are [60, 59, 6, 45, 83, 44, 34, 130, 130]
PolynomialPauliR	10 control qubits, polynomial coefficients are [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
WeightedAdder	20 controls qubits, values for sum are [9, 0, 9, 10, 2, 6, 10, 6, 8, 5, 8, 7, 8, 4, 0, 0, 5, 7, 5, 6]

Table 4: Reqomp results for the reductions presented in Table 1. We also report Unqomp results. Columns **Max** and **Min** report the results for the most aggressive settings, respectively optimizing only for number of qubits and optimizing only for number of gates. Columns **-75%**, **-50%**, and **-25%** report the gate counts when achieving the respective ancilla reductions. Entries "x" indicate that a given ancilla reduction was not achieved. Q is total number of qubits, A is number of ancillae, CX is number of CX gates, G is total number of gates and D is circuit depth.

	Ancilla Reduction														
	Max					**-75%**					**-50%**				
Algorithm	Q	A	CX	G	D	Q	A	CX	G	D	Q	A	CX	G	D
Small															
Adder	28	4	326	1132	482	x	x	x	x	x	30	6	270	924	346
Deutsch-Jozsa	15	4	78	331	162	x	x	x	x	x	15	4	78	331	162
Grover	8	2	192	839	456	x	x	x	x	x	x	x	x	x	x
IntegerComparator	17	4	110	456	251	x	x	x	x	x	18	5	100	422	231
MCRY	17	4	122	486	263	x	x	x	x	x	18	5	104	414	225
MCX	17	4	102	405	243	x	x	x	x	x	18	5	96	381	202
Multiplier	24	20	420	1500	550	x	x	x	x	x	x	x	x	x	x
PiecewiseLinearR	10	3	1000	3851	2188	x	x	x	x	x	10	3	1000	3851	2188
PolynomialPauliR	8	2	360	1381	846	x	x	x	x	x	x	x	x	x	x
WeightedAdder	25	40	689	2606	1184	x	x	x	x	x	x	x	x	x	x
Big															
Adder	207	7	6824	24664	8832	225	25	2820	9792	3005	250	50	2470	8492	2806
Deutsch-Jozsa	108	7	2700	10999	5632	125	24	1038	4351	2289	150	49	888	3751	2041
Grover	15	4	3600	15312	7734	x	x	x	x	x	15	4	3600	15312	7734
IntegerComparator	108	7	2720	11758	6190	125	24	1042	4584	2452	150	49	892	3938	2177
MCRY	209	8	7358	29430	15268	250	49	2096	8382	4646	300	99	1796	7182	4134
MCX	209	8	7278	29109	15283	250	49	2088	8349	4653	300	99	1788	7149	4141
Multiplier	79	64	4688	16768	5764	x	x	x	x	x	x	x	x	x	x
PiecewiseLinearR	47	6	31064	124834	67353	51	10	25252	99754	55636	61	20	23812	93512	50941
PolynomialPauliR	15	4	30322	119245	65640	x	x	x	x	x	15	4	30322	119245	65640
WeightedAdder	38	80	1857	7042	3259	x	x	x	x	x	x	x	x	x	x

	Ancilla Reduction										**Unqomp**				
	-25%					**Min**									
Algorithm	Q	A	CX	G	D	Q	A	CX	G	D	Q	A	CX	G	D
Small															
Adder	33	9	228	768	275	35	11	200	664	267	36	12	200	664	267
Deutsch-Jozsa	17	6	66	283	142	19	10	54	235	126	19	8	54	235	126
Grover	8	2	192	839	456	9	3	168	743	386	9	3	168	743	378
IntegerComparator	21	8	82	346	172	24	12	68	294	164	24	11	68	310	164
MCRY	21	8	86	342	177	24	12	68	270	161	24	11	68	270	161
MCX	20	7	84	333	179	23	12	66	261	154	23	10	66	261	154
Multiplier	x	x	x	x	x	24	20	420	1500	550	24	20	460	1580	573
PiecewiseLinearR	11	4	958	3671	1952	13	6	906	3465	1932	13	6	906	3583	1986
PolynomialPauliR	8	2	360	1381	846	9	3	330	1261	693	9	3	330	1261	735
WeightedAdder	x	x	x	x	x	25	40	689	2606	1184	25	40	749	2886	1322
Big															
Adder	275	75	2120	7192	2494	299	99	1784	5944	2291	300	100	1784	5944	2291
Deutsch-Jozsa	174	73	744	3175	1563	199	100	594	2575	1386	199	98	594	2575	1386
Grover	17	6	3000	12912	6279	19	8	2550	11112	5902	19	8	2550	11112	5852
IntegerComparator	175	74	742	3284	1662	200	100	596	2670	1484	200	99	596	2862	1484
MCRY	350	149	1496	5982	3145	400	200	1196	4782	2793	400	199	1196	4782	2793
MCX	349	148	1494	5973	3138	399	200	1194	4773	2786	399	198	1194	4773	2786
Multiplier	x	x	x	x	x	79	64	4688	16768	5764	79	64	5168	17728	5886
PiecewiseLinearR	71	30	22372	87224	45711	81	40	21050	81592	44966	81	40	21050	84852	46752
PolynomialPauliR	17	6	26962	105805	55664	19	8	26572	104245	50936	19	8	26572	104245	60488
WeightedAdder	x	x	x	x	x	38	80	1857	7042	3259	38	80	1989	7658	3458