Back-end-aware Fault-tolerant Quantum Oracle Synthesis

Mingfei Yu
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland

Alessandro Tempia Calvino
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland

Mathias Soeken
Microsoft Quantum
Zurich, Switzerland

Giovanni De Micheli
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland

Abstract

Quantum oracle synthesis involves compiling arbitrary Boolean functions into quantum circuits using specific quantum gates supported by the target quantum computer. The *Clifford+T* gate library is particularly common in fault-tolerant quantum computing systems. Utilizing *XOR-AND-inverter graphs* (XAGs) as the logic representation for the target Boolean functions has received extensive attention due to the observed direct correlation between the number of AND nodes in an XAG and the *T count* and the *helper qubit count* of the quantum oracle optimally compiled from it. However, to be deployed onto fault-tolerant quantum hardware, quantum gates must be further re-expressed by logical *quantum error correction* (QEC) code operations, a process known as *back-end compilation*. This paper enhances the current XAG-based oracle synthesis techniques by establishing a link between the properties of XAGs and quality measures of back-end-compiled quantum oracles. This link unlocks more optimization opportunities—experimental results demonstrate average reductions of 4.49% in T count, 7.00% in *logical time steps*, and 14.89% in helper qubit count, respectively, on benchmarks optimized by the proposed back-end-aware XAG optimization approaches.

ACM Reference Format:
Mingfei Yu, Alessandro Tempia Calvino, Mathias Soeken, and Giovanni De Micheli. 2025. Back-end-aware Fault-tolerant Quantum Oracle Synthesis. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3658617.3697776

1 Introduction

The capabilities of quantum computers are now widely recognized. In contrast to classical physics, where a system is typically in a single well-defined state, in quantum mechanics, a system can exist in multiple states simultaneously, i.e., a *superposition*, until it is measured. In terms of a single qubit φ, its quantum state $|\varphi\rangle$ is a superposition of the two *computational basis states*, the classical states of 0 and 1 ($|0\rangle$ and $|1\rangle$), described as $|\varphi\rangle = a_0 |0\rangle + a_1 |1\rangle$. Here, a_0 and a_1 are complex amplitudes, and $|a_0|^2$ and $|a_1|^2$ indicate the probabilities that φ is in the classical 0 and 1 states, respectively.

To run a quantum algorithm on a quantum computer, the program has to be compiled into a quantum circuit through *quantum compilation*. Specifically, a quantum circuit is also called a *quantum oracle* when it implements a Boolean function. Given a Boolean function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m) : \mathbb{B}^n \to \mathbb{B}^m$, with $x = x_1 \ldots x_n$, $y = y_1 \ldots y_m$, and the bitwise XOR operation denoted as '$\oplus$', the corresponding quantum oracle can be described as $O_f : |x\rangle |y\rangle |0\rangle^k \mapsto |x\rangle |y \oplus f(x)\rangle |0\rangle^k$. Besides n qubits and m qubits to respectively store the inputs and outputs, oracle O_f requires k ancillary qubits to hold intermediate results, which shall be uncomputed to the $|0\rangle$ state at the end of the computation, so as to guarantee an accurate computation, as temporary data on the helper qubits compromise the measurement results. Quantum oracles play an essential role in various quantum algorithms.

Quantum computers are inherently error-prone, which is a fundamental challenge to realizing practical quantum computation. The *Clifford+T* gate library provides a solution to achieve scalable and reliable quantum computation. It consists of two types of gates: (i) Clifford gates, including *controlled-NOT* (*CNOT*), *Hadamard* (*H*), *π/2-phase rotation* (*S*) and its inverse ($S^\dagger$); (ii) non-Clifford gates, including *π/4-phase rotation* (*T*) and its inverse ($T^\dagger$). Compared to Clifford gates, the physical implementation of high-fidelity non-Clifford gates is much more resource-intensive, but non-Clifford gates are necessary to enable the gate set to be universal [8]. Thus, when synthesizing quantum oracles over the Clifford+T gate library, implementations that minimize the T-count are preferable.

Significant efforts have been dedicated to exploring quantum oracle synthesis techniques based on various logic representations, such as *look-up table* (LUT)-based methods [21] and *XOR-AND-inverter graph* (XAG)-based methods [13]. Then, existing research on low-resource-cost quantum oracle synthesis typically formulates the compilation task as implementing the target logic function into a *reversible circuit*, which consists of CNOT, NOT, and *Toffoli* gates. Using reversible circuits as an intermediate representation is advantageous because, when re-expressed using the Clifford+T gate library, the Toffoli gate is the only gate type that requires T gates for its implementation. Specifically, the computation and uncomputation of a Toffoli gate necessitate at least 4 T gates [6]. Since a Toffoli gate can realize a logical AND operation, there is a direct correlation between the number of AND nodes[1] in an *XOR-AND-inverter graph* (XAG) and the T count of its resulting quantum oracle [12]. Additionally, the qubit count is also determined by the number of AND nodes, which must be computed reversibly, while

[1] Unless otherwise specified, an AND node refers to a 2-input AND node in this paper.

ASPDAC '25, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01
https://doi.org/10.1145/3658617.3697776

XOR operations can be computed in-place using CNOTs, requiring no additional qubits. Thus, XAG-based synthesis techniques are particularly effective for low-resource-cost oracle synthesis by minimizing the number of AND nodes in the XAG implementation of the target Boolean function [19].

However, constraining the oracle synthesis problem to the reversible circuit level has its limitations. For successful quantum application execution, *quantum error correction* (QEC) is indispensable for protecting logical qubits from errors. While critical for scalable quantum computing, QEC increases the resource overhead significantly, such as constructing logical qubits from raw physical qubits, mapping logical qubits to 2D geometries, *etc.* These complexities are not considered at the reversible circuit level. In contrast, the *quantum instruction set architecture* (QISA) acts as an intermediary layer between reversible circuits and physical implementations, providing an option for reliably estimating resources needed for fault-tolerant quantum computing. QISA abstracts the QEC implementation at the physical level, retaining only fault-tolerant logical operations within its instruction set. We categorize the compilation of Boolean functions into reversible circuits and subsequently into QISA executables into two distinct stages: *front-end* and *back-end* compilation. Whereas existing works are limited to the front-end compilation stage, this work expands the oracle synthesis problem to include the back-end stage.

In this work, we employ the *planar QISA* based on the *surface code* [2, 4], the most established QEC code. Utilizing the *parallel synthesis sequential Pauli computation* (PSSPC) back-end compilation scheme [3], we reveal that the *logical time steps* required by the execution of a QISA executable of an oracle, a pivotal quality measure in the QISA level, are determined by the multi-qubit measurement operations necessary for remote Toffoli gate execution, as detailed in Section 2.2. The number of logical time steps directly corresponds to the execution runtime of the algorithm. Thus, for the first time, a direct correlation is established between the AND count of XAGs and the number of logical time steps involved in QISA executables, and consequently, the runtime it requires to run an algorithm on fault-tolerant quantum hardware.

Furthermore, inspired by recent advancements in Clifford+T construction of 3-control Toffoli gates [7], we demonstrate that utilizing this construction can reduce logical time steps and helper qubits without increasing the T count. Therefore, our compilation technique reduces the number of physical qubits and execution runtime required to run quantum oracles on fault-tolerant quantum computers compared to the state-of-the-art. We elaborate on the link between an XAG and its QISA executable quality measures in Section 3. Not only the AND count but also the connectivity of AND nodes influence the number of logical time steps and the usage of helper qubits in the compiled QISA executable: replacing two cascaded 2-input AND (AND2) nodes with a single 3-input (AND3) node reduces logical time steps from 8 to 7 and helper qubit count from 2 to 1.

We incorporated this cost metric into the XAG synthesis and optimization process, empowering *back-end-aware XAG-based quantum oracle synthesis* by developing logic synthesis and optimization algorithms, as detailed in Sections 4 through 6. Experimental evaluations have confirmed the effectiveness of our approach, with synthesized oracles achieving an average of 4.49% reduction in T count, 7.00% in logical time steps, and 14.89% in helper qubit count.

2 Background

2.1 XAG Optimization

2.1.1 XOR-AND-inverter Graphs. Boolean functions are typically abstracted as logic networks for compact representation. A logic network is a *directed acyclic graph* (DAG), where each node represents a logic operation. A network is an XAG if all nodes have two fan-ins and correspond to either an AND or XOR operation, with complementation regarded as a property of edges.

2.1.2 Cuts. A *cut* highlights a cone within a logic network. It is defined by the *root* (*primary output*, PO) and *leaves* (*primary inputs*, PIs) of the partial network described by the cut. A set of leaves is valid for a given root if: (a) each path from any PI to the root passes through at least one leaf; and (b) for each leaf, there is at least one path that passes exclusively through it.

2.1.3 Logic rewriting. Logic optimization techniques use cuts to enhance scalability. Among these, *logic rewriting* optimizes a logic network by replacing each part with its optimized implementation. It is commonly employed to reduce the size [15, 22], depth [18], or other metrics of logic networks. Existing research has used logic rewriting to reduce the AND count of XAGs [23].

2.2 Back-end Compilation

In fault-tolerant quantum computing, QEC is crucial for forming reliable logical qubits from noisy physical qubits. QISA serves as *logical QEC code*, a logical abstraction of lower-level physical QEC code. Previous research on quantum oracle synthesis has focused on compiling Boolean functions, typically represented as XAGs, into reversible circuits, with little consideration for the subsequent step of compiling into QISA executables. We distinguish these two stages as *front-end* and *back-end* compilation.

Implementing QEC is resource-intensive, necessitating the enhancement of oracle synthesis techniques to be back-end-aware. Establishing a link between XAG properties and QISA executable quality measures can unlock more optimization opportunities, as it would allow XAGs to be synthesized in a way that results in lower-resource-cost QISA executables.

We chose QISA executables, rather than physical QEC codes, as the objective of back-end compilation. The physical resource to implement a QISA executable depends on the required error rate of each QISA operation to achieve the desired execution accuracy of the target quantum algorithm, which varies significantly across applications. For instance, quantum factoring algorithms can tolerate lower execution accuracy since verifying the correctness of a solution is straightforward. Thus, targeting QISA executables for back-end compilation offers a generic yet sufficiently precise resource estimation for synthesized oracles.

2.2.1 Planar quantum instruction set architecture. QISA, serving as the logical QEC code, closely aligns with the physical system architecture, including geometric constraints. We adopted the planar QISA [3], which is based on the logic operations of the surface code, the most well-established physical QEC code. Thus, the planar

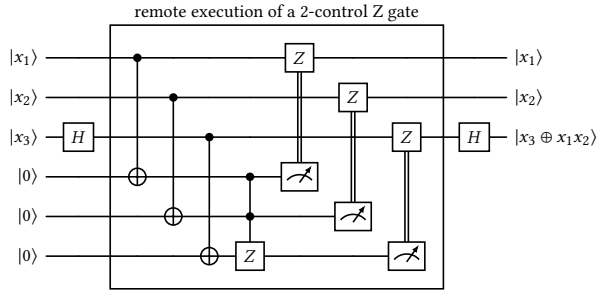

Figure 1: Remote execution of a Toffoli: The upper three qubits are algorithm ones and the lowers are synthesis ones.

QISA includes an instruction set consisting of logical surface code operations.

It supports the following Clifford operations: *single-qubit initialization*, *single-qubit measurement*, and *multi-qubit measurement*, as well as non-Clifford operations, whose realization requires T gates: Toffoli gates and arbitrary angle rotations around the X, Y, or Z axis. Since quantum oracle does not involve arbitrary angle rotations, Toffoli gate is the only non-Clifford operation in our context.

The T gates required for non-Clifford operations are supplied by *T factories*, which are prepared through a process known as *magic-state distillation* [16]. The produced T states are injected into the computation as T gates via magic-state teleportation in combination with a multi-qubit measurement [9]. This explains why non-Clifford operations are more resource-intensive, both in terms of runtime and physical resources.

2.2.2 Parallel synthesis sequential Pauli computation. We adopt the emerging PSSPC back-end compilation scheme proposed in [3], which enhances the execution of quantum algorithms by leveraging parallelism and optimizing the use of quantum resources. This scheme has two key features.

First, the execution of non-Clifford operations is delegated from the original qubits, called *algorithm qubits*, to remote qubits, called *synthesis qubits*. *Remote execution* allows for the parallel execution of non-Clifford operations, thereby improving runtime. It also ensures easy access to the T gates required by non-Clifford operations, as T factories are typically situated at layout boundaries [2].

Since Toffoli gates are the only non-Clifford operations for quantum oracles at the QISA level, we illustrate the remote execution of a Toffoli gate in Fig. 1. This process involves first entangling the algorithm qubits with the synthesis qubits, followed by measuring the synthesis qubits on the X basis. Each entanglement operation, depicted in Fig. 1 as a CNOT for clarity, is actually achieved by performing a 2-qubit Z measurement (refer to Fig. 14(a) in [10] for a more detailed illustration). After remotely executing the 2-control Z operation on the synthesis qubits, Z rotations are applied to the algorithm qubits for necessary phase corrections based on the X measurement results.

Second, after delegating the execution of non-Clifford operations to synthesis qubits, the remaining operations on the algorithm qubits are all Clifford gates. These are further eliminated by commuting them towards the output side of the circuit, a process known as *Clifford elimination*, which significantly reduces layout complexity. As a side effect, measurements and phase corrections, become sequential multi-qubit operations. Notably, measurements are significantly slower and thus dominate the runtime [3].

2.2.3 Resource cost of oracles from a back-end perspective. At the QISA level, the resource cost extends beyond the T count and helper qubit count considered at the reversible circuit level, to include the number of logical time steps required to execute the QISA executable of the oracle. It is a crucial quality measure, as more steps to execute mean an increased failure probability of quantum algorithms, requiring more physical qubits to lower the logical error rate for successful computation.

Due to Clifford elimination, the 2-qubit measurements to entangle the algorithm and synthesis qubits become multi-qubit measurements. Consequently, executing each Toffoli gate requires three logical time steps for the sequential multi-qubit measurements necessary for entanglement. Additionally, one extra logical time step is required for the T-free measurement-based uncomputation [6]. These logical time steps for the remote execution of Toffolis are the primary contributors to the execution time of the QISA executable of a quantum oracle [3]. This complements our resource cost model for quantum oracles, highlighting the following three key measures of interest: (a) T count, which impacts the number of physical qubits needed, as it dictates the number of T factories required to produce a sufficient number of T states; (b) logical time steps, which directly affect the runtime; and (c) helper qubit count, which primarily influences the total number of physical qubits.

Considering each AND node in an XAG translates into a Toffoli gate, a quantitative relationship between the AND count of an XAG and the resource cost measures of the resulting QISA executable is established. Specifically, given an XAG implementing the target Boolean function with $\#\wedge_2$ AND nodes, our observation, along with existing research [12], suggests that the compiled QISA executable would require $4{\cdot}\#\wedge_2$ T gates, $\#\wedge_2$ helper qubits, and $4{\cdot}\#\wedge_2$ logical time steps for execution.

3 AND3 Operation via 3-control Toffoli Gates

In [7], a 6-T construction of 3-control Z gates was proposed (Fig. 2). This suggests a 6-T construction of 3-control Toffoli gates, which

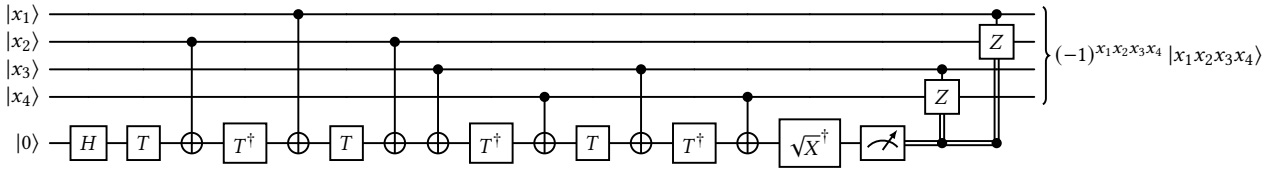

Figure 2: A 6-T 3-control Z gate construction [7].

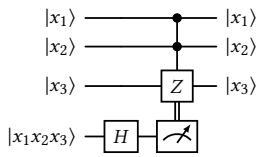

Figure 3: Measurement-based uncomputation of AND3.

essentially realizes an AND3 operation. In this section, we analyze this advanced construction and highlight its potential for producing lower-resource-cost quantum oracles.

3.1 A Six-T Construction of 3-control Toffolis

The 6-T construction of 3-control Z gates in Fig. 2 is based on the observation that a 3-control Z gate maps quantum state $|x_1x_2x_3x_4\rangle$ to $(-1)^{x_1x_2x_3x_4}|x_1x_2x_3x_4\rangle$. The coefficient equals $i^{x_1x_2\oplus x_3x_4-x_1x_2-x_3x_4}$, with i representing the imaginary unit. The main idea is using two overlapping Toffoli gates to implement $i^{x_1x_2\oplus x_3x_4}$; While building each Toffoli requires at least 4 T gates, the overlapping cancels two of them and cuts down the T count to 6. Notice that $\sqrt{X}^\dagger$ is a Clifford operation, as $\sqrt{X}^\dagger = (H\sqrt{Z}H)^\dagger = HS^\dagger H$.

Since $X = HZH$, a 6-T construction for 3-control X gates, i.e., 3-control Tofolli gates, can be realized on top of Fig. 2 by adding two more H gates to the target line to encase the 3-control Z gate.

3.2 Resource Requirement

A 3-control Toffoli operation realizes an AND3 operation. Its uncomputation is illustrated in Fig. 3. When the fourth qubit is measured as 1 (with a probability of 50%), a 2-control Z gate must be applied to the first three qubits for phase correction.

The costs of an AND3 operation using the 6-T 3-control Toffoli gate construction are: (a) T count: $6 + 4 \times 0.5 = 8$; (b) Logical time steps: $4 + 1 + (3 + 1) \times 0.5 = 7$; (c) One helper qubit. All logical time steps are consumed by measurements. Four steps are needed for entanglement during computation, one for uncomputation (Fig. 3), and an additional four steps in half the cases due to the extra 2-control Z gate required for phase correction.

Since an AND3 operation is equivalent to two concatenated AND2 operations, using the proposed 6-T 3-control Toffoli gate-based AND3 operation reduces resources compared to using two 4-T Toffoli gate-based AND2 operations: (a) T count remains the same (4×2 versus 8); (b) Logical time steps are reduced from 8 (4×2) to 7; (c) Helper qubits are reduced from 2 (1×2) to 1.

Incorporating this realization of AND3 operations, the resource costs of an oracle, regarding the QISA level, are: Denoting the numbers of AND2 and AND3 operations as $\#\wedge_2$ and $\#\wedge_3$, respectively,

- T count: $4{\cdot}\#\wedge_2 + 8{\cdot}\#\wedge_3$
- Logical time steps: $4{\cdot}\#\wedge_2 + 7{\cdot}\#\wedge_3$
- Helper qubit count: $\#\wedge_2 + \#\wedge_3$

Conclusively, during the XAG synthesis stage, maximizing the use of concatenated AND nodes, i.e., AND3 nodes, can result in lower-cost oracles. This observation introduces an additional aspect to the XAG-based quantum oracle synthesis problem. Not only does the number of AND nodes in an XAG matter, but also their connectivity, i.e., the topology of an XAG with respect to AND nodes, is crucial to its quality measure.

4 Group and Split

Our method for maximizing the use of AND3s in XAG synthesis involves two steps: (i) Maximally *group* concatenated AND nodes; (ii) *Split* each group of ANDs into pairs and merging each pair into an AND3 node. We refer to this strategy as *group-split*.

A similar approach can be found in solutions to the *AND-inverter graph* (AIG) depth optimization problem, where researchers propose *covering* AND nodes into groups to balance the logic network [14]. For the *grouping* step of our strategy, we adapted the AND-covering algorithm from [14]. All AND nodes in the XAG are traversed in reverse topological order, i.e., from the PO side to the PI side. An AND group is identified when one of the following conditions is met: (a) an AND node has multiple fanouts; or (b) an AND node's fanout is complemented.

Our strategy differs from existing methods in how the AND groups collected in the first step are utilized. Specifically, in the *split* step, while [14] focuses on splitting the AND groups for depth optimization, our goal is to maximize the count of AND3 nodes.

Group-split provides a runtime-efficient solution to exploit AND3 nodes, as it maintains the network structure with linear time complexity. Nonetheless, further exploration is warranted to restructure XAGs to enhance the opportunities for utilizing AND3s.

5 Database of cost-minimum XAGs

In this section, we aim to find the cost-minimum XAG implementation for small-scale Boolean functions, which falls under the category of *exact synthesis*. The optimum implementations are collected in a database, which enables efficient logic rewriting subsequently. In [17], a formulation is proposed for the exact synthesis of AND-count-minimum XAGs. We extend this formulation to consider both the AND count and the AND connectivity as the cost measures in our context.

Our formulation leverages the concept of *AND fence* [24]. By maximally grouping AND nodes in an XAG, as introduced in Section 4, the AND fence is numerically defined as the size of each group in topological order. Denoting the AND fence of an XAG as $\mathcal{F} = \{c_1, \cdots, c_\alpha\}$, where each c_i indicates there are c_i AND2s in the i-th AND group, we can calculate the numbers of AND2 and AND3 nodes in the XAG after splitting the AND groups as $\#\wedge_2 = \sum_{i=1}^{\alpha}(c_i \bmod 2)$ and $\#\wedge_3 = \sum_{i=1}^{\alpha} \lfloor c_i/2 \rfloor$. These values jointly determine the cost of the XAG, as discussed in Section 3.2.

We formulate the problem of finding the cost-minimum XAG implementation for a given function as an incremental *Boolean satisfiability* (SAT) problem. The inputs to each SAT instance are (a) the target function and (b) the target AND fence. If an instance is *satisfiable*, an XAG exists that simultaneously implements the function and meets the given AND fence.

In our SAT formulation: (a) Given that c_i concatenated AND2 nodes equal functionally a $(c_i + 1)$-input AND node, each step of the Boolean chain is a multi-fanin AND, with a fanin size corresponding to the target AND fence. (b) The fanins of each step are parity functions, whose operands can be either PIs or previous steps. (c) The PO is also a parity function, with operands selected from PIs and all steps in the Boolean chain. Thus, the task of the solver is to find a valid configuration of the operands of all involved parity functions. Fig. 4 showcases a certain trial of exploring if there exists

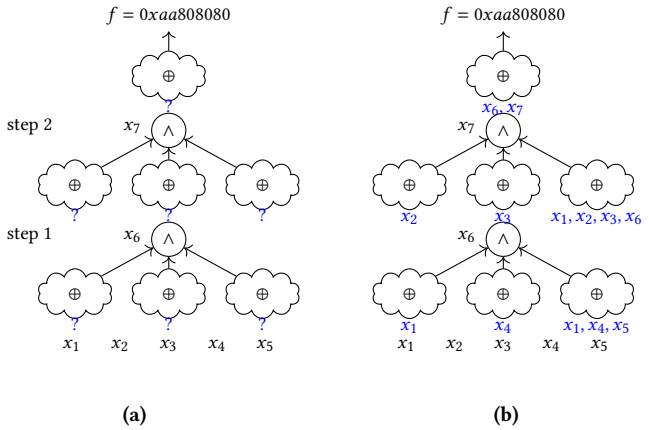

Figure 4: Synthesizing the cost-minimum XAG implementation of a Boolean function: (a) Visualizing a certain SAT instance; (b) Solving the SAT instance to synthesize an XAG.

an XAG implementation for the function $f = 0xaa808080$ [2] with the AND fence $\mathcal{F} = \{2, 2\}$.

Using this formulation, we can find the cost-minimum XAG implementation of a function by enumerating AND fences in a cost-increasing order. Specifically, we set the T count and the number of logical time steps as the primary and secondary criteria, respectively. Indeed, the XAG implementation synthesized in Fig. 4 is the cost-minimum one for the illustrated function.

Exploiting the proposed exact synthesis methodology, we aim to build a database that collects the cost-minimum XAG implementations for all 5-variable Boolean functions. While there exist 2^{32} functions, applying Boolean classification techniques can significantly reduce the number of unique cases, enabling the generation of a functionally complete database. *Affine equivalence classification* [5] is recognized to be a perfect fit, since affine operations are AND fence-invariant and, therefore, cost-invariant in our context. If two functions are affine-equivalent, their cost-minimum XAG implementations share the same AND fence. All 5-variable Boolean functions are classified into 48 affine-equivalent classes; For each class, a representative is selected in accordance with the classification algorithm, for which the cost-minimum XAG is synthesized.

6 XAG Optimization via Rewriting

With a database of optimum implementations for small-scale functions, a common way to conduct scalable logic optimization is the logic rewriting technique introduced in Section 2.1.3.

However, how to adapt logic rewriting to our cost optimization problem is not obvious. Logic rewriting generally refers to a procedure of replacing small regions of the logic network, identified by cuts, with better implementations. In contrast to existing applications of logic rewriting, in our context, the cost of each AND node is dynamically determined by its environment. This is because AND2 nodes at the boundaries of the cuts may be merged into AND3 nodes. This makes accurately evaluating the cost of a replacement an intractable problem, as one move can change the connectivity

[2]We represent truth tables in hexadecimal as a bit-string, with the most significant bit on the left-hand side.

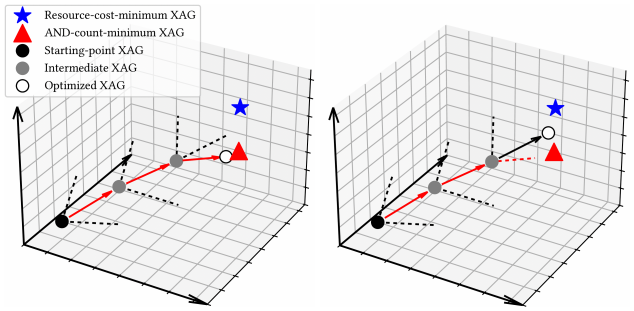

Figure 5: Role of an ideal cut filter in the rewriting procedure.

of those ANDs outside the logic cone highlighted by the cut. In this section, we aim to devise effective and efficient heuristics to address this problem.

6.1 Using AND Count as Cost Function

Since we adopt the T count as the primary criterion in resource measure, the 48 cost-minimum XAGs in our database are guaranteed to be T-count-minimum. Our exploration starts by simplifying the optimization problem as an AND count minimization problem, like existing XAG-based quantum oracle synthesis techniques [12].

Fig. 5a schematically depicts how a logic rewriting flow explores the design space. Each move indicates an operation of rewriting a cut. In each step, a particular node in the network is regarded as the root, and the cuts rooted on which are the candidates to be rewritten. For clarity, in the figure, we fix the number of candidates in each step to three. Among the three options, the AND-count-minimal one (marked in red) can be distinguished (denoted using dotted lines) and a rewriting is then committed.

Since rewriting a cut only ensures local optimality, a strategy that greedily minimizes AND count may lead to an XAG design approaching the AND-count-minimum one. Given the direct correlation between the T count in an oracle and the AND count in its corresponding XAG, such a design intuitively approaches the resource-cost-minimum one within the design space. However, as discussed in Section 3.2, the quality measures of a compiled quantum oracle, such as logical time steps and helper qubit count, also depend on the connectivity of AND nodes within its XAG. Therefore, developing a method to guide the design process closer to the resource-cost-minimum design is of significant interest.

6.2 Devising Cut Filters

As Fig. 5b illustrates, we suppose an elaborated cut filter to meet the following requirements: (a) When in a design space far from the global optimums, the filter should not prevent the flow from proceeding towards the direction that approaches the optimal solution most efficiently; (b) When getting close enough to the optimums, the filter should be able to effectively rule out those cuts that, once rewritten, would likely direct the flow to sub-optimality w.r.t. resource cost.

6.2.1 Rigid cut filter. The difficulty of developing a low-resource-cost-oriented logic rewriting method lies in efficiently taking into

Algorithm 1: Rigid cut filter for cost minimization.

```
   Input: Cuts rooted on node n, cuts
   Output: A set of cuts that pass the filter, cuts_valid
1  foreach cut c ∈ cuts do
2      if same_AND_group_as_fanout(n) then
3          return ∅
4      is_valid ← true
5      foreach leaf l ∈ leaves of cut c do
6          if same_AND_group_as_fanout(l) then
7              is_valid ← false
8              break
9      if is_valid then add c to cuts_valid
10 return cuts_valid
```

consideration the influence of rewriting a cut on the existing AND groups in the network. A potential cut filter design can be: let the filter check whether the root and its fanout node, or any leaf and its fanout node, belong to the same AND group. If a cut passes such a filter, it means rewriting this cut would not break any existing AND group and is therefore a conservative decision.

In Algo. 1, to judge if a node and its fanout node belong to the same AND group, the same criterion in the group-split strategy in Section 4 is implemented as the "same_AND_group_as_fanout" function (lines 2 and 6).

However, the inflexibility of the rigid cut filter is a concern. Using the flow in Fig. 5b for illustration, the filter may make unwise decisions and rule out the AND-count-minimal moves (the red arrows) in early steps, leading the flow to sub-optimality. Hence, a cut filter with more comprehensive decision logic is required to achieve a steady performance.

6.2.2 Voter-driven cut filter. The voter-driven cut filter described by Algo. 2 is with an improved decision-making mechanism: (a) Instead of stopping at checking if the root and leaves of the current cut belong to existing AND groups, it further analyzes the optimal implementation that would replace this cut (lines 6-14); This enables precise estimation of the number of AND groups that would be broken if the current cut is rewritten (recorded as *skip_score*). (b) Even rewriting a cut would cause the breaking of existing AND groups, it would not be directly filtered out; Instead, the decision depends on whether the number of AND groups to be broken exceeds a pre-defined *threshold* (lines 15 and 16).

Observing that most of the cost reduction is achieved by rewriting 5-input cuts, we statistically configure the threshold to one-fifth. This means a cut with five leaves would not be filtered out, as long as rewriting this cut would not break more than one existing AND group. In contrast, a cut with fewer than five leaves would be ruled out if rewriting it would result in a violation of any existing AND group, as the loss in cost reduction would likely outweigh the gain of optimally implementing this cut. This setting facilitates a more comprehensive evaluation of each rewriting choice.

7 Experimental Results

To evaluate the effectiveness of the proposed approach, experiments are designed and conducted on the EPFL combinational benchmark

Algorithm 2: Voter-driven cut filter for cost minimization.

```
   Input: Cuts rooted on node n, cuts, and the database, db
   Output: A set of cuts that pass the filter, cuts_valid
1  foreach cut c ∈ cuts do
2      f ← Boolean function of c
3      {f_repr, operations}← affine_canonicalize(f)
4      new_cut_impl ← apply operations to db[f_repr]
5      skip_score ← 0
6      if same_AND_group_as_fanout(n) then
7          po ← PO node of new_cut_impl
8          if same_AND_group_as_fanout(po) then
9              skip_score ← skip_score + 1
10     foreach leaf l ∈ leaves of cut c do
11         if same_AND_group_as_fanout(l) then
12             pi ← PI of new_cut_impl akin to l
13             if same_AND_group_as_fanout(pi) then
14                 skip_score ← skip_score + 1
15     if skip_score / (#leaves(c)+1) < threshold then
16         add c to cuts_valid
17 return cuts_valid
```

suite [1]. Since previous work on XAG-based oracle synthesis focuses exclusively on reducing the AND count of logic networks, the XAGs optimized using the state-of-the-art AND count reduction technique [11] are adopted as the baseline for comparison.

All four proposed techniques are evaluated: (i) *Group-split*: the group-split strategy proposed in Section 4 that maximally converts concatenated AND nodes into AND3 nodes, without restructuring the logic network; (ii) *Rewrite*: the AND-count-oriented logic rewriting flow introduced in Section 6.1, which exploits the database of cost-minimum XAG implementations of 5-variable functions proposed in Section 5; (iii) *Filter-rigid*: on top of *Rewrite*, the rigid cut filter (Algo. 1) is applied to guide the rewriting flow; (iv) *Filter-vote*: the voter-driven cut filter (Algo. 2) is applied to guide the rewriting flow. Notice that (i) is applied to the XAGs optimized by either (ii), (iii), or (iv) as a post-process to support the use of AND3 nodes.

All the proposals are implemented as part of the C++ logic synthesis library `mockturtle` [20][3]. The results reported below are obtained on an Apple M1 Max chip with 32GB memory.

Given that the T count is adopted as the primary criterion in resource measure, the latter three techniques that involve logic rewriting are applied repetitively for each benchmark, until no improvement in T count is observed. The runtime spent in total to reach the saturation is denoted as "time" in Table 1. Exploiting the numerical relations revealed in Section 3.2, using the AND2 and AND3 counts of an XAG, the resource cost of the resulting oracle, in terms of T count ("T"), number of logical time steps ("steps"), and helper qubit count, can be estimated. Due to space limitations, only the first two are reported in Table 1.

Without any effort on logic restructuring, applying the group-split strategy already achieves a 2.81% logical time steps reduction and a 7.47% helper qubit count reduction, with no increase in T count. This observation demonstrates the advantage of exploiting

[3]Available at: https://github.com/lsils/mockturtle

Table 1: Evaluation of back-end-aware XAG-based quantum oracle synthesis techniques.

Benchmark	Baseline		Group-split			Rewrite			Filter-rigid			Filter-vote		
	T	Steps	T	Steps	Time[s]	T	Steps	Time[s]	T	Steps	Time[s]	T	Steps	Time[s]
adder	512	512	512	512	<0.01	512	512	<0.01	512	512	<0.01	512	512	0.05
barrel shifter	3328	3328	3328	3328	<0.01	3328	3328	0.11	3328	3328	0.11	3328	3328	2.08
divider	20528	20528	20528	20338	<0.01	18884	18779	22.00	18988	18871	16.30	18992	18878	108.02
log2	35092	35092	35092	34812	<0.01	33136	32849	99.62	33284	32923	99.32	33088	32792	440.14
max	3488	3488	3488	3447	<0.01	3252	3240	1.16	3480	3438	0.16	3244	3230	9.73
multiplier	30340	30340	30340	30267	<0.01	30080	29970	43.93	30060	29939	40.31	30080	29968	352.44
sine	7836	7836	7836	7739	<0.01	7440	7272	48.93	7476	7279	35.35	7444	7271	179.20
square-root	20868	20868	20868	20739	<0.01	19300	19160	53.21	19408	19236	56.47	19272	19135	271.94
square	18384	18384	18384	18314	<0.01	18092	17992	19.01	18064	17940	19.29	18084	17985	119.02
round-robin arbiter	4696	4696	4696	4652	<0.01	4592	4255	12.62	4624	4238	9.11	4592	4255	87.58
coding-cavlc	1576	1576	1576	1510	<0.01	1472	1387	10.70	1520	1426	5.59	1492	1405	16.07
ALU control unit	180	180	180	176	<0.01	176	171	0.82	180	177	0.39	176	170	1.94
decoder	1312	1312	1312	1312	<0.01	1312	1312	0.29	1312	1312	0.29	1312	1312	5.15
i2c controller	2228	2228	2228	2173	<0.01	2092	2026	7.91	2084	2019	5.50	2092	2026	26.10
int to float converter	340	340	340	327	<0.01	316	303	1.98	324	309	1.11	316	303	4.34
memory controller	18780	18780	18780	18235	<0.01	17692	17052	39.53	17984	17275	29.86	17700	17052	392.59
priority encoder	1292	1292	1292	1265	<0.01	1268	1228	2.64	1064	1027	2.50	1156	1109	18.98
look-ahead XY router	372	372	372	342	<0.01	372	341	1.38	372	337	0.13	372	341	3.42
voter	17028	17028	17028	16695	<0.01	15440	14912	63.08	15652	15056	41.18	15372	14842	313.50
geometric mean (norm.)	**1**	**1**	**1**	**0.982**		**0.959**	**0.935**		**0.960**	**0.934**		**0.955**	**0.930**	

the 3-control Toffoli construction introduced in Section 3 to realize logical AND3 operation.

Based on a comparison among the estimated resource cost obtained without and with logic restructuring, the effectiveness of the rewriting flow and the generated database of the cost-minimum XAG implementations for 5-variable Boolean functions, is proved: the largest gap in cost minimization is achieved on *priority encoder*, where the rewriting flow, under the guidance of the rigid cut filter, achieved a 17.65% reduction in T count, a 20.51% reduction in logical time steps, and a 29.01% reduction in helper qubit count over the SOTA. Without the endeavor in XAG optimization, the gain obtained by applying the group-split strategy is trivial: a 2.09% reduction in logical time steps and an 8.33% reduction in helper qubit count, with an unchanged T count. This result sustains our argument of bringing the connectivity of AND nodes into the scope of the low-resource-cost oracle synthesis problem, i.e., enhancing XAG-based oracle synthesis techniques to be back-end-aware.

Lastly, we compare the performance of the three implemented logic rewriting flows. Applying the rigid cut filter outperforms others for 4 benchmarks. But as we suspected, it also happens for 13 benchmarks that the resulting T count is rather higher than without applying any filter, indicating that the inflexible decision-making mechanism can lead the rewriting flow to sub-optimality. By contrast, the voter-driven cut filter is qualified to guide the flow to explore a design space closer to the cost-minimum XAGs. Excluding the 4 tied benchmarks, the voter-driven cut filter guides the rewriting flow to find a better XAG implementation in 12 out of the 16 cases; Also, in the remaining 4 cases, the voter-driven cut filter-enhanced rewriting flow hardly ends up with a design of the highest T count, except for *barrel shifter*. These observations evidence that, at the sacrifice of a reasonably higher runtime, the voter-driven cut filter serves as a reliable cut filter.

8 Conclusion

XAGs have been identified as an ideal logic representation for quantum oracle synthesis due to the direct correlation between the AND count of an XAG and the T count and qubit count of its resulting quantum oracle. To more precisely estimate the resources required for implementing fault-tolerant quantum circuits, it is essential to consider the re-expression of quantum circuits as logical QEC codes, i.e., the back-end compilation process. We have established a link between the properties of XAGs and the quality measures of back-end-compiled quantum circuits. This connection indicates that both the AND count and the connectivity of AND nodes in XAGs should be considered. This insight unlocks additional optimization opportunities to reduce the resources required by quantum oracles and enables more accurate resource estimation.

To support this observation, we have devised back-end-aware XAG synthesis and optimization algorithms for low-resource-cost quantum oracle synthesis. Experimental results have demonstrated that our approach achieves average reductions of 4.49% in T count, 7.00% in logical time steps, and 14.89% in helper qubit count.

The feature that concatenated ANDs are preferable to separate ones distinguishes the problem itself as a unique and self-contained logic synthesis problem, which is technically interesting and merits additional study. Our approach is not limited by the technical fact that there is not yet T-count-efficient construction for Toffoli gates with more than three control lines. Therefore, when more advanced implementations of multiple-controlled Toffoli gates are discovered in the future, the proposed algorithms can be easily adapted.

Acknowledgments

This project is supported in part by Synopsys Inc. We are grateful to the anonymous reviewers for their insightful comments.

References

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL Combinational Benchmark Suite. In *International Workshop on Logic and Synthesis*.

[2] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. 2022. Surface Code Compilation via Edge-Disjoint Paths. *PRX Quantum* 3 (2022), 020342. Issue 2.

[3] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. 2022. Assessing Requirements to Scale to Practical Quantum Advantage. arXiv:2211.07629

[4] Sergey B. Bravyi and Alexei Yu. Kitaev. 1998. Quantum Codes on a Lattice with Boundary. arXiv:9811052

[5] Colin R. Edwards. 1975. The Application of the Rademacher–Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis. *IEEE Trans. Comput.* C-24, 1 (1975), 48–62.

[6] Craig Gidney. 2018. Halving the Cost of Quantum Addition. *Quantum* 2 (2018), 74.

[7] Craig Gidney and N. Cody Jones. 2021. A CCCZ Gate Performed with 6 T Gates. arXiv:2106.11513

[8] Daniel Gottesman. 1998. The Heisenberg Representation of Quantum Computers. In *International Colloquium on Group Theoretical Methods in Physics*. 32–43.

[9] Daniel Litinski. 2019. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum* 3 (2019), 128.

[10] Daniel Litinski and Naomi Nickerson. 2022. Active volume: An architecture for efficient fault-tolerant quantum computers with limited non-local connections. arXiv:2211.15465

[11] Hsiao-Lun Liu, Yi-Ting Li, Yung-Chih Chen, and Chun-Yao Wang. 2022. A Don't-care-based Approach to Reducing the Multiplicative Complexity in Logic Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4821–4825.

[12] Giulia Meuli, Mathias Soeken, Earl Campbell, Martin Roetteler, and Giovanni De Micheli. 2019. The Role of Multiplicative Complexity in Compiling Low T-count Oracle Circuits. In *International Conference on Computer-Aided Design*. 1–8.

[13] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. 2022. Xor-And-Inverter Graphs for Quantum Compilation. In *npj Quantum Information*, Vol. 8.

[14] Alan Mishchenko, Robert Brayton, Stephen Jang, and Victor Kravets. 2011. Delay Optimization using SOP Balancing. In *International Conference on Computer-Aided Design*. 375–382.

[15] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis. In *Design Automation Conference*. 532–535.

[16] Joe O'Gorman and Earl T. Campbell. 2017. Quantum Computation with Realistic Magic-state Factories. *Physical Review A* 95 (2017), 032338. Issue 3.

[17] Mathias Soeken. 2020. Determining the Multiplicative Complexity of Boolean Functions using SAT. arXiv:2005.01778

[18] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. 2017. Busy man's synthesis: Combinational delay optimization with SAT. In *Design, Automation & Test in Europe Conference & Exhibition*. 830–835.

[19] Mathias Soeken and Mariia Mykhailova. 2022. Automatic Oracle Generation in Microsoft's Quantum Development Kit using QIR and LLVM passes. In *Design Automation Conference*. 1363–1366.

[20] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, and Giovanni De Micheli. 2022. The EPFL Logic Synthesis Libraries. arXiv:1805.05121v3

[21] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. 2019. LUT-Based Hierarchical Reversible Logic Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 9 (2019), 1675–1688.

[22] Alessandro Tempia Calvino and Giovanni De Micheli. 2024. Scalable Logic Rewriting using Don't Cares. In *Design, Automation & Test in Europe Conference & Exhibition*. 1–6.

[23] Eleonora Testa, Mathias Soeken, Luca Amarú, and Giovanni De Micheli. 2019. Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications. In *Design Automation Conference*. 1–6.

[24] Mingfei Yu and Giovanni De Micheli. 2023. Striving for Both Quality and Speed: Logic Synthesis for Practical Garbled Circuits. In *International Conference on Computer-Aided Design*. 1–9.