MLIR for Quantum Beyond Gate Cancellation: Quantum Circuit Mapping Reimagined

Matthias Reumann*, Yannick Stade*, Robert Wille*†, and Lukas Burgholzer*†
*Chair for Design Automation, Technical University of Munich, Munich, Germany
†MQSC, Garching near Munich, Germany
{matthias.reumann, yannick.stade, robert.wille, lukas.burgholzer}@tum.de
www.cda.cit.tum.de/research/quantum

***Abstract*—The Multi-Level Intermediate Representation (MLIR) framework has become a cornerstone for building extensible, domain-specific compilers, with the quantum computing community already leveraging it to model quantum programs and implement basic optimizations. However, computationally intensive tasks in the quantum compilation pipeline, such as quantum circuit mapping, remain underexplored within the MLIR ecosystem. This paper proposes an MLIR-native blueprint for these non-local, quantum-specific optimization routines by reimplementing a well-established, state-of-the-art mapping A* search algorithm for qubit routing and SWAP insertion. Our evaluation demonstrates that this approach not only integrates seamlessly into an MLIR-based quantum compiler collection but also surpasses previous non-MLIR solutions in both solution quality and runtime. The implementation is open-source and publicly available at github.com/munich-quantum-toolkit/core.**

***Index Terms*—quantum computing, quantum circuit mapping, quantum compilation**

I. Introduction

As quantum computing hardware scales beyond the noisy intermediate-scale quantum (NISQ) era [1, 2], the complexity of quantum programs that can be executed increases significantly. However, the inherent noise and limited coherence times of current and near-term devices necessitate highly optimized compilation pipelines to maximize circuit fidelity. While simple, local optimizations such as gate cancellation or fusion are standard in most compilers, they are increasingly insufficient to bridge the gap between algorithmic requirements and hardware capabilities. Consequently, there is a growing need for advanced, global optimization routines capable of significantly reducing circuit depth and error rates.

Developing such sophisticated compilers from scratch is a formidable software engineering challenge. Instead of reinventing the wheel, the classical compiler community has converged on Multi-Level Intermediate Representation (MLIR) [3] as a modular and extensible framework for building domain-specific compilers. Following its immense success in the machine learning and high-performance computing domains [4–7], MLIR is gaining traction in the quantum computing community. Several works have explored modeling quantum programs within MLIR and implementing basic optimizations [8–12]. However, these efforts largely focus on feasibility studies involving relatively simple, local transformations. A crucial question remains unanswered: Is MLIR suitable for implementing complex, quantum-specific optimization routines that are non-local and computationally intensive?

In this work, we address this question by investigating quantum circuit mapping —often synonymously referred to as qubit routing or layout synthesis in the literature—within the MLIR ecosystem. Quantum circuit mapping is a mandatory and critical step in the compilation pipeline for architectures with restricted qubit connectivity, such as superconducting quantum processors. The problem involves transforming an abstract quantum circuit into one that respects the hardware's coupling graph by inserting SWAP gates, aiming to minimize the added overhead. Since the problem is NP-complete and the search space grows exponentially with the number of qubits, it represents a significantly more challenging test case for MLIR than typical peephole optimizations.

With this work, we do not aim to propose yet another mapping algorithm, as the literature is already rich with solutions. Instead, we implement a well-established, state-of-the-art mapping approach based on A* search [13] using MLIR-native abstractions. This allows us to critically evaluate whether the framework's infrastructure facilitates the development of such complex algorithms and how the resulting performance compares to bespoke C++ implementations.

Our evaluation demonstrates that:

- The investigated mapping algorithm integrates seamlessly into an MLIR-native compiler collection, confirming that MLIR provides the necessary tools and abstractions for complex quantum optimizations.
- Leveraging existing MLIR infrastructure and utilities significantly reduces development time, allowing developers to focus on algorithms rather than boilerplate code.
- The MLIR-native implementation outperforms the original state-of-the-art C++ implementation in terms of both solution quality (fewer SWAP gates) and runtime.

Ultimately, this work provides evidence that MLIR is not just a viable modeling tool but a powerful and efficient framework for the next generation of high-performance quantum compilers.

This paper is structured as follows: To ensure that the paper is self-contained, we review the necessary background in Section II. Section III contextualizes and motivates this work within the existing literature. In Section IV, we outline the proposed solution in detail and subsequently evaluate it in Section V. Finally, Section VI concludes the paper.

II. Background

This section outlines the fundamental concepts required to understand the remainder of this paper.

A. Multi-Level Intermediate Representation (MLIR)

Building an end-to-end quantum compilation pipeline entails substantial engineering effort. The underlying infrastructure must efficiently model, parse, and transform quantum programs while addressing both current and emerging requirements.

Whereas previous efforts often involved building quantum compilers from scratch [14, 15], a recent trend has shifted toward a "classical-first" approach [16]. This strategy leverages established classical compiler infrastructures, such as the Multi-Level Intermediate Representation (MLIR) [3], to support quantum computing. This section provides an overview of modeling and transforming quantum programs within the MLIR framework.

The MQT Compiler Collection [16] models quantum programs as directed acyclic "data-flow" graphs. Each quantum operation is represented as a vertex that consumes and produces qubit values (represented as ingoing and outgoing edges, respectively). Hence, each edge captures the evolution of a qubit's state throughout the circuit. This representation is particularly advantageous for optimization and complex transformations, as it makes dependencies between operations explicit.

Moreover, each qubit state is described as a Static Single Assignment (SSA) value. By definition, an SSA value is assigned exactly once, ensuring that the data flow remains immutable and explicitly traceable. Within the MLIR framework, these values are denoted by the `%` prefix.

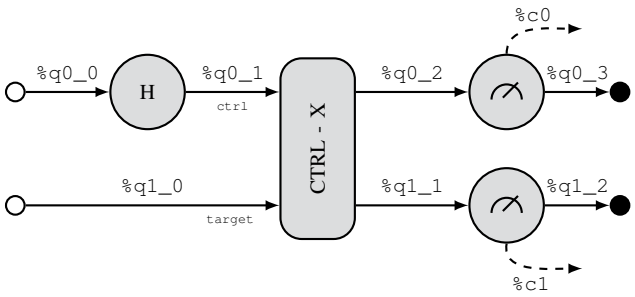

Figure 1. The data-flow graph of a two qubit circuit.

Example 1. *Figure 1 illustrates the data-flow graph of a two-qubit circuit. As shown, allocations (∘) initialize the circuit by producing the initial SSA values `%q0_0` and `%q1_0`. A subsequent Hadamard transforms the first qubit, consuming `%q0_0` and producing `%q0_1`. Similarly, a controlled-X consumes `%q0_1` and `%q1_0` to produce `%q0_2` and `%q1_1`. Measurements follow a similar logic, with the addition that they also produce classical SSA values, `%c0` and `%c1`. Finally, sinks (•) consume the final qubit values, thereby defining the end of their respective lifetimes.*

To implement this domain-specific language, the MQT Compiler Collection leverages MLIR's dialect mechanism. A dialect groups a set of operations, types, and attributes under a common namespace. While MLIR provides various built-in dialects, its architecture is fundamentally extensible, offering a framework for developing custom dialects tailored to specialized domains. Furthermore, MLIR natively supports the parsing and transformation of IR modules that integrate primitives from multiple coexisting dialects. The Quantum Circuit Optimization (QCO) dialect, introduced within the MQT Compiler Collection, defines the primitive types (e.g., `!qco.qubit`) and operations (e.g., `qco.h`) necessary for modeling quantum computations.

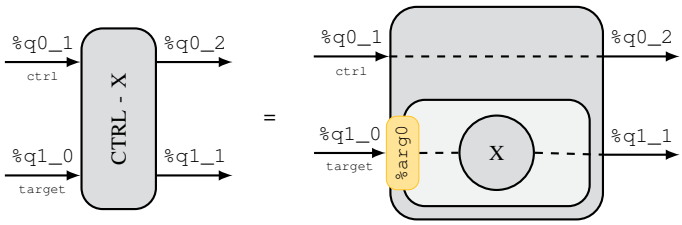

Figure 2. An illustration of the control-modifier in the QCO dialect.

Example 2. *The following snippet presents the textual intermediate representation (IR) corresponding to the data-flow graph in Figure 1. The controlled-X gate is realized via the `qco.ctrl` modifier, a generic construct that enables the definition of arbitrary controlled operations. The `qco.yield` operation serves as the terminator for the nested region (delimited by curly brackets), passing the transformed qubit values back to the outer scope. Figure 2 provides a visual schematic of the data-flow within this control-modifier construct.*

```
%q0_0 = qco.alloc : !qco.qubit
%q1_0 = qco.alloc : !qco.qubit

%q0_1 = qco.h %q0_0 : !qco.qubit -> !qco.qubit
%q0_2, %q1_1 = qco.ctrl(%q0_1)
                    targets (%arg0 = %q1_0)
{
    %q1_1 = qco.x %arg0 : !qco.qubit -> !qco.qubit
    qco.yield %q1_1
} : ({!qco.qubit}, {!qco.qubit}) ->
    ({!qco.qubit}, {!qco.qubit})

%q0_3, %c0 = qco.measure %q0_2 : !qco.qubit
%q1_2, %c1 = qco.measure %q1_1 : !qco.qubit

qco.sink %q0_3 : !qco.qubit
qco.sink %q1_2 : !qco.qubit
```

MLIR provides two primary methods for traversing IRs:

- **Structural perspective**: This involves walking the IR operations in hierarchical order (top-down or bottom-up) and recursively visiting nested regions, such as those within `qco.ctrl`. MLIR provides utilities to perform these traversals efficiently.

- **Data-flow perspective**: This involves traversing the data-flow graph (as illustrated in Figure 1) by following MLIR's def-use chains. These chains link an SSA value to the operation that defined it and the subsequent operations that consume it (the "users"). While MLIR supports standard SSA data-flow, it lacks built-in support for traversing these chains according to specific quantum semantics.

Example 3. *For instance, in the circuit shown in Figure 1, the defining operation for the SSA value* `%q0_1` *is the Hadamard gate, while its consumer (or user) is the controlled-X operation.*

The MLIR framework provides comprehensive utilities for manipulating IRs by inserting, updating, or deleting operations. These transformations are orchestrated by its pass infrastructure, which provides a generic mechanism for analysis and rewriting. In this context, a pass traverses the IR to perform analysis or transformation. Furthermore, individual passes can be composed into a pass pipeline, ensuring they are executed in a deterministic sequence. Much like its dialect system, MLIR's pass infrastructure is highly extensible, allowing for the development of custom transformations tailored to specific quantum compilation requirements.

Example 4. *The canonicalization pass of the QCO dialect implements a custom rewrite strategy that removes unused qubits. To find these, MLIR's def-use chain is utilized: If the defining operation of the input of the* `qco.sink` *operation is a* `qco.alloc` *operation, both operations can be removed.*

In summary, the MLIR framework and the MQT Compiler Collection's QCO dialect provide the necessary foundation to implement an efficient quantum circuit mapping pass.

B. Quantum Circuit Mapping Using A Search*

Each quantum computing architecture imposes a unique set of hardware-specific constraints. Consequently, the compiler must transform high-level quantum programs into a format that adheres to these constraints while maintaining logical equivalence.

In superconducting architectures, for instance, two-qubit connectivity is typically restricted: gates can only be executed between physically adjacent qubits. Consequently, a necessary precondition for mapping is the decomposition in one and two-qubit gates (excluding barriers). These limitations are formally represented by a coupling graph $G = (V, E)$, where vertices V denote hardware qubits and edges E represent the available two-qubit interactions.

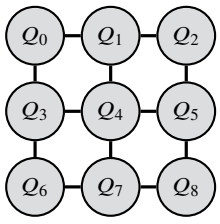

Figure 3. A 3×3 grid architecture.

Example 5. *Figure 3 depicts the coupling graph for a superconducting processor, where physical qubits are arranged in a* 3×3 *grid. Two-qubit operations are restricted to adjacent qubits that are connected with an edge. For example, while a two-qubit gate can be directly applied to hardware qubits* Q_0 *and* Q_3*, a direct interaction between* Q_0 *and* Q_4 *is prohibited by the architecture.*

Due to these topological constraints, many quantum programs cannot be executed natively on a given target architecture. This occurs whenever no perfect initial program-to-hardware qubit mapping exists, for which the target qubits of all two-qubit gates are physically adjacent on the hardware coupling graph.

A naive solution to this problem involves inserting SWAP gates along the shortest path between non-adjacent target qubits. This process progressively moves the state of one target qubit until it is physically adjacent to the other, enabling the execution of the required gate.

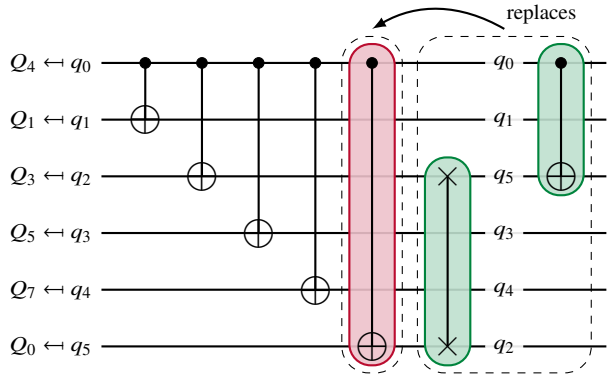

Figure 4. By exchanging the program qubits q_5 and q_2 with a SWAP gate, the final controlled-X gate acts on the adjacent hardware qubits Q_4 and Q_3.

Example 6. *Figure 4 illustrates the mapping procedure for a 6-qubit circuit onto the architecture from Figure 3. Initially, the program qubits* $(q_0, q_1, q_2, q_3, q_4, q_5)$ *are assigned to the physical qubits* $(Q_4, Q_1, Q_3, Q_5, Q_7, Q_0)$*, respectively. Because program qubit* q_0 *has an output degree of* 5 *and the maximum output degree of the architecture is* 4 *(*Q_4*), no static mapping can satisfy all connectivity constraints. Consequently, certain two-qubit gates remain non-executable. For example, the controlled-X gate (highlighted in red) initially acts on a non-adjacent pair of hardware qubits. To resolve this, a SWAP operation is performed between the hardware qubits* Q_0 *and* Q_3*, exchanging the locations of the program qubits* q_2 *and* q_5*. Subsequently, the final gate is mapped to the hardware qubits* Q_3 *and* Q_4*. Because these qubits are adjacent in the topology, the gate can now be executed natively.*

While straightforward to implement, this local routing strategy often incurs significant gate overhead [13]. Unfortunately, finding a mapping that globally minimizes the total number of additional SWAP gates is known to be NP-hard [17, 18]. To mitigate this, a variety of heuristic search methods have been developed [13, 19–22]. The present work focuses on the

A*-based mapping algorithm introduced by Zulehner et al. [13]. The following paragraphs outline the algorithm's details.

In this approach, the circuit is typically partitioned into layers of disjoint two-qubit gates. The A* algorithm is then used to find a sequence of SWAP gates that satisfies the connectivity constraints for the current layer while considering the impact on subsequent gates.

Example 7. *Figure 5 illustrates a quantum circuit partitioned into three layers based on gate dependency constraints. The first layer comprises two-qubit gates g1 and g2, which can be executed concurrently because they act on disjoint sets of qubits. Because g3 requires the outputs of both g1 and g2, it is assigned to the second layer. Similarly, g4 is placed in the third layer because it depends on g3. Because for fidelity-unaware routing single qubit gates are irrelevant, they can be safely ignored when layering.*

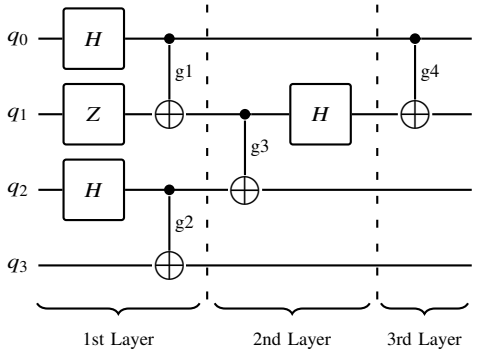

Figure 5. A quantum circuit divided into three layers.

As the number of qubits and consequently the concurrent two-qubit gates increases, finding an optimal SWAP sequence within the exponentially large search space becomes increasingly intractable. To maintain feasible runtimes, the mapping algorithm may partition a single layer into multiple sub-layers.

Example 8. *The "individual gate" strategy decomposes a layer into its two-qubit gates. Consequently, each resulting layer contains only a single gate. For instance, the first "individual gate" layer in Figure 5 is either* $\{(q_0, q_1)\}$ *or* $\{(q_2, q_3)\}$

Each node s in the A* search space represents a specific program-to-hardware mapping π_s:

$$\pi_s : q_0, q_1, \ldots, q_{n-1} \rightarrow Q_0, Q_1, \ldots, Q_{m-1} \tag{1}$$

where n and m denote the number of program and hardware qubits, respectively. We distinguish two types of special nodes:

- **Root Node**: Represents the initial mapping at the start of the current layer's optimization. It serves as the origin for the search.
- **Goal Node**: A node is defined as a goal node if its associated mapping π_s satisfies the connectivity constraints for all gates in the current layer.

The search algorithm terminates upon the discovery of a goal node, yielding a mapping that allows the native execution of all two-qubit operations in the layer.

The total cost $f(s)$ is determined by the standard A* evaluation function:

$$f(s) = g(s) + h_i(s) \tag{2}$$

Here, $g(s)$ represents the path cost from the root to the current node s, defined as the number of SWAP gates applied to reach the current mapping π_s:

$$g(s) = \text{dist}(\text{root}, s) \tag{3}$$

The heuristic function $h_i(s)$ estimates the remaining cost to satisfy the connectivity requirements of the current layer L_i. It is calculated as the sum of the distances between the hardware qubits assigned to each program qubit pair in the layer:

$$h_i(s) = \sum_{(q_a, q_b) \in L_i} (D(\pi_s(q_a), \pi_s(q_b)) - 1) \tag{4}$$

where the distance function $D(Q_j, Q_k)$ denotes the shortest path length between hardware qubits Q_j and Q_k in the coupling graph G.

Starting from the root, the algorithm iteratively expands the node with the lowest total cost $f(s)$ until a goal node is reached. To expand a node s, the algorithm identifies all two-qubit gates within the current layer L_i and their mapped hardware qubits. For each such gate, the algorithm generates child nodes by applying all possible SWAP operations that involve at least one of these hardware qubits. Consequently, each child corresponds to a unique mapping π_s resulting from a single SWAP. These new states are then inserted into a minimum priority queue — the frontier — for fast retrieval.

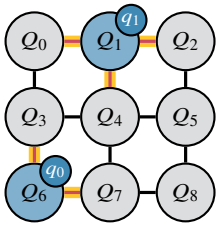

Figure 6. The considered SWAP gates on the architecture shown in Figure 3 for a layer consisting solely of the gate $(q_0, q_1) \mapsto (Q_6, Q_1)$.

Example 9. *Figure 7 illustrates the first two expansion steps of the A* search for a layer consisting of a single gate:* $(q_0, q_1) \mapsto (Q_6, Q_1)$. *In the initial expansion, the algorithm generates successor nodes and evaluates the cost function* $f(s)$ *for the root node for all feasible SWAP operation. Figure 6 illustrates the process of finding feasible SWAP operations for the gate* $(q_0, q_1) \mapsto (Q_6, Q_1)$. *The algorithm considers all SWAP operations incident to either* Q_1 *or* Q_6. *For hardware qubit* Q_1, *the available operations are* (Q_0, Q_1), (Q_1, Q_4), *and* (Q_1, Q_2),. *Similarly, for* Q_6, *the feasible SWAP gates are* (Q_6, Q_7) *and* (Q_6, Q_3). *After the initial expansion, the algorithm selects the node with the lowest total cost for the next expansion. If multiple successors have identical costs, we choose one at random. In the second expansion step, the search proceeds by evaluating a new set of potential SWAPs based on*

the updated program-to-hardware mapping, and continues this iterative process until a goal state is visited. In this example, we generate two potential goal nodes in the second expansion step: (Q_0, Q_3) *and* (Q_6, Q_3)*. Consequently, the algorithm terminates, if it were to visit one of these nodes.*

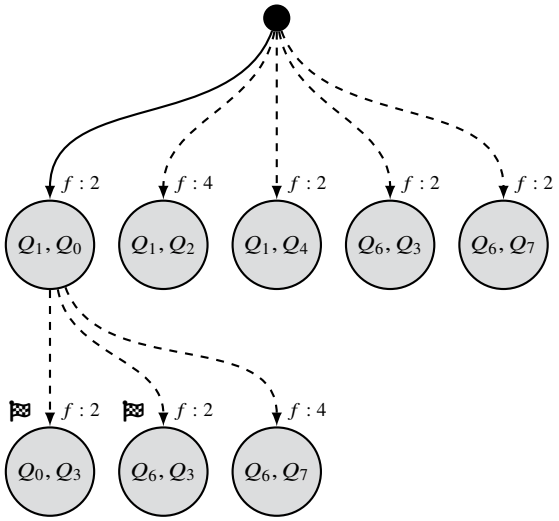

Figure 7. The first two expansion steps of the A* search algorithm for a layer consisting solely of the gate $(q_0, q_1) \mapsto (Q_6, Q_1)$.

Because the SWAP sequence and the resulting layout thereof directly influence subsequent layers, the heuristic may also consider future layers. The *lookahead-aware* heuristic cost function is defined as:

$$\hat{h_i}(s) = \sum_{j \in W(i, N_L)} \lambda^{j-i} \cdot h_j(s), \tag{5}$$

where $W(j, N_L) = (k)_{j \le k < j+N_L}$ and the weight $\lambda \in (0, 1)$ decays exponentially to decrease the contribution of the lookahead layers.

Example 10. *The initial "front" of the circuit shown in Figure 5 is the layer* $\{(q_0, q_1), (q_2, q_3)\}$*. The goal of the A* search is to find a SWAP sequence such that both of these gates are executable on the target architecture. Given the window size* $N_L = 3$ *and the weight* $\lambda = 0.5$*, the lookahead layers for this front are* $\{(q_1, q_2)\}$ *and* $\{(q_0, q_1)\}$ *with decay factors* 0.5 *and* 0.25*, respectively. The two-qubit gates inside the lookahead layers contribute to the heuristic cost function* $\hat{h_0}(v)$ *but don't have to be executable, yet.*

III. Motivation

While MLIR has been leveraged for various tasks in the quantum compilation pipeline, a significant research gap remains in its use for complex, non-local transformations.

Existing literature features several specialized dialects: the *Quantum* dialect focuses on lowering MLIR to low-level IRs [10], while *QSSA* introduces static analysis passes to enforce the no-cloning theorem [9]. Optimization efforts have largely remained localized: For instance, *QIRO* utilizes data flow analysis for local circuit improvements [8], and Xanadu's *Catalyst* framework targets just-in-time (JIT) compilation [11]. Furthermore, IBM's *qe-compiler* employs MLIR to generate executables for quantum control systems [12], and the *ASDF* compiler provides a foundation for basis-oriented quantum languages [23]. While QLLVM [24] fundamentally relies on MLIR for its compilation flow, it implements the SABRE algorithm at the lower LLVM abstraction level, bypassing the structural advantages of MLIR.

Collectively, these projects prioritize local optimization routines and translation between abstraction layers. However, a robust, end-to-end quantum compiler requires efficient handling of global optimizations, such as quantum circuit mapping. To date, a purely MLIR-native quantum compilation stack has not been presented in the literature. As a fundamental building block of superconducting qubit compilation, quantum circuit mapping is an ideal candidate for demonstrating that MLIR is well-suited to addressing complex, end-to-end challenges.

IV. Proposed Solution

While the literature is rich with quantum circuit mapping algorithms and their respective implementations, MLIR-native approaches remain largely unexplored. This section details the modular components of the proposed mapping pass for superconducting architectures. Each component demonstrates how the strengths of MLIR can be leveraged to build reusable and maintainable solutions without compromising on performance.

Although these refinements are primarily architectural rather than algorithmic, as we will demonstrate in the evaluation, these improvements — facilitated by the MLIR ecosystem — exert a significant impact on the solution's overall scalability.

The source code is publicly available at github.com/munich-quantum-toolkit/core.

A. Program Traversal

Efficient program traversal is essential for optimization, transpilation, and general compilation routines. We combine the linear qubit type of the QCO dialect with MLIR's efficient def-use chain representation to streamline the traversal of quantum programs. Specifically, we implement a *bidirectional (qubit) wire iterator* (referred to as "wire iterator" in the following) that traverses all operations associated with a single qubit while preserving the underlying data-flow dependencies. By composing multiple such iterators for the individual qubits in a quantum program, we enable the implementation of complex, multi-qubit traversal patterns and global program analysis.

Example 11. *Figure 8 shows the forward iteration process of the proposed wire iterator. The iterator* `it` *first points to the allocation operation of the first qubit (indicated by a blue highlight around the white circle). After one increment (*`++it`*), it points to the controlled operation. Incrementing again, it follows the output qubit SSA value of the controlled operation and, afterwards, points to the Z operation.*

We utilize the wire iterator to divide the program into layers of disjoint two-qubit operations. The traversal is initialized with each iterator positioned at the qubit allocation operations.

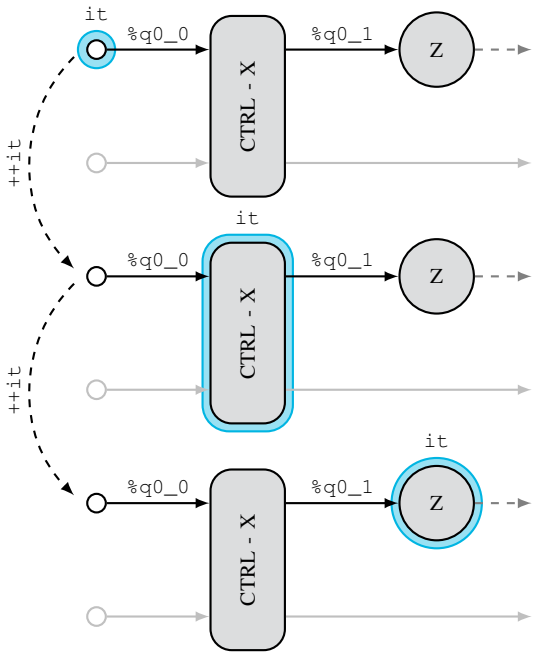

Figure 8. Iteration of a wire iterator. The blue highlight indicates the current position of the iterator.

During each iteration, the iterators advance along their respective wires until a two-qubit operation is encountered. A two-qubit operation is marked as "ready" and inserted into the current layer only when it has been reached twice—once by each of the two involved qubits' iterators. This ensures that all dependencies for the two-qubit operation have been satisfied, and it can be executed in parallel with other operations in the same layer.

Example 12. *Figure 9 illustrates the first iteration of the layer-finding algorithm for five qubits. In this example, the layer consists of two controlled operations, which we highlight in green. The algorithm advances the wire iterator of each qubit until a two-qubit operation is encountered. If two wire iterators reach the same two-qubit operation in the same iteration, the two-qubit operation is considered "ready" and inserted into the current layer. "Unready" operations, such as the controlled-X acting on the upper two qubits, belong to one of the subsequent layers. Lastly, the wire iterators of the "ready" two-qubit operations are incremented and the next iteration starts.*

By initializing the wire iterators at the qubit deallocation operations and decrementing them, the same logic can be applied to traverse the circuit backwards. This symmetry allows for an analogous backward traversal to compute reverse-order layers, which can be beneficial for certain optimization strategies that consider both forward and backward passes through the program, such as the SABRE algorithm [19].

To further minimize the number of layers, the algorithm aggregates operations into two-qubit blocks. We define such a block as the sequence of all single- and two-qubit operations acting on the same pair of qubits as a "defining" two-qubit

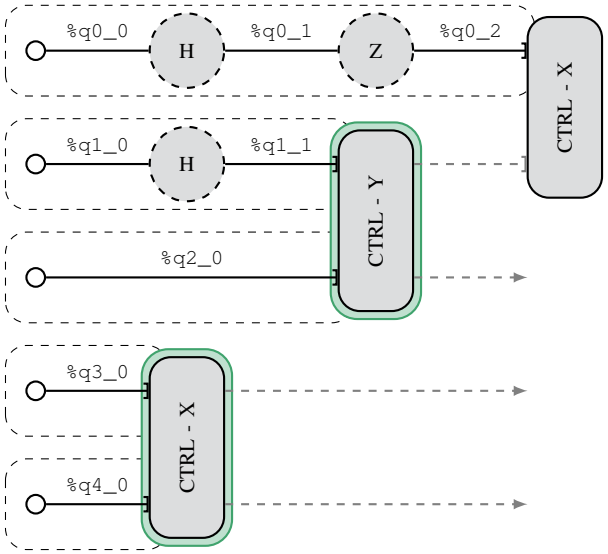

Figure 9. Finding the two-qubit gates (highlighted in green) of a layer.

operation. If this defining two-qubit operation satisfies the target architecture's connectivity constraints, all operations within the block are guaranteed to be executable as well. Leveraging MLIR's def-use chain, this strategy is efficiently implemented by advancing a pair of wire iterators in tandem until they encounter an operation involving a third qubit.

Example 13. *Figure 10 shows the composition of a two-qubit block. The two-qubit block starts with the controlled-X operation on the upper two qubits and contains all single- and two-qubit operations acting on the same qubits. In this example, these operations are the two Hadamards, the Z operation, and finally, the controlled-Z operation. The controlled-Y operation does not belong to the two-qubit block because it acts on a third qubit, namely the lowest one.*

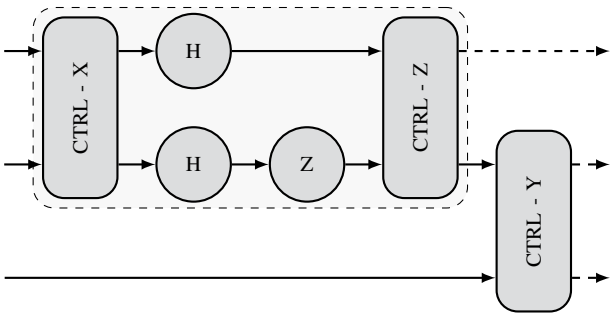

Figure 10. A two-qubit block aggregates all one and two-qubit operations acting on the same wires.

Finally, we enable program traversal across arbitrary gate sets by leveraging the `UnitaryOpInterface` within the QCO dialect. While existing approaches [25] are often constrained by hard-coded gate sets, the proposed solution remains entirely gate-agnostic. This design future-proofs the mapping pass against evolving architectures and varying native gate-sets.

B. Arena-Based A* Search

As the most computationally intensive component of the mapping pass, the performance of the A* search is integral to the overall performance. By utilizing MLIR's built-in data structures and memory management primitives, we introduce several architectural refinements over prior implementations.

The implementation of the A* mapping algorithm in [25] stores the complete sequence of SWAP operations from the root to the current node for each node of the A* search graph. This design imposes a significant memory overhead, as each child node redundantly replicates its parent's path history. To mitigate this, we propose an arena-based A* search. Instead of storing the full path, each node maintains only a pointer to its parent, allowing the optimal SWAP sequence to be reconstructed once a goal node is found. To manage these nodes, we leverage MLIR's `SpecificBumpPtrAllocator`, which allocates nodes in contiguous memory blocks by simply advancing a pointer within a pre-allocated (possibly growing) memory region – the "arena". Usually, the frontier manages and stores the nodes of the search graph. Because in the proposed solution the arena contains the nodes, the frontier consists of node pointers instead. This approach avoids memory fragmentation with near-zero allocation overhead and allows the entire search graph to be deallocated in a single operation upon completion.

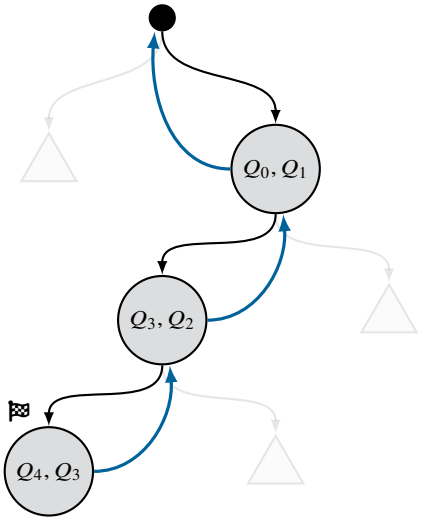

Figure 11. Reconstructing the SWAP sequence in the search-graph via parent pointers.

Example 14. *Figure 11 illustrates the SWAP sequence reconstruction mechanism. The reconstruction originates at the goal node (indicated by the flag symbol) and traverses the solution path via parent pointers until the root node is reached. In the example shown, the sequence* (Q_4, Q_3)*,* (Q_3, Q_2)*, and finally* (Q_0, Q_1) *is extracted. This sequence is subsequently reversed to yield the final path from the root to the goal node.*

We minimize reliance on the C++ Standard Library (`std::`) in favor of specialized data structures from the MLIR ecosystem. For instance, we utilize `SmallVector` to reduce heap allocations by leveraging its inline storage for small-to-moderate element counts. Similarly, replacing standard maps with LLVM-native associative containers, such as `DenseMap`, further optimizes cache locality. Moreover, `llvm::PriorityQueue` serves as a drop-in replacement for `std::priority_queue` while providing convenience functions to re-order elements efficiently.

Because distinct SWAP sequences can yield the same program-to-hardware qubit mapping, expanding nodes with identical states introduces significant search redundancy. To prune suboptimal paths, we implement a closed-set mechanism by storing these mappings in an MLIR `DenseMap`, which maintains the minimum SWAP sequence length for each mapping. A node is only expanded if its number of SWAPs is strictly less than the previously seen one.

Example 15. *Figure 12 illustrates a scenario where distinct paths within the search space yield the same program-to-hardware qubit mapping. The search starts at the root node with an identity mapping, where each program qubit* q_i *is assigned to its corresponding hardware qubit* Q_i*. As shown, the SWAP sequences* $((0,1))$ *and* $((0,2),(0,1),(1,2))$ *result in the same qubit permutation:* $(0,1,2) \mapsto (1,0,2)$*. Expanding these equivalent nodes (highlighted in yellow) yields identical sub-trees, as the expanding strategy depends solely on the current mapping rather than the path taken to reach it.*

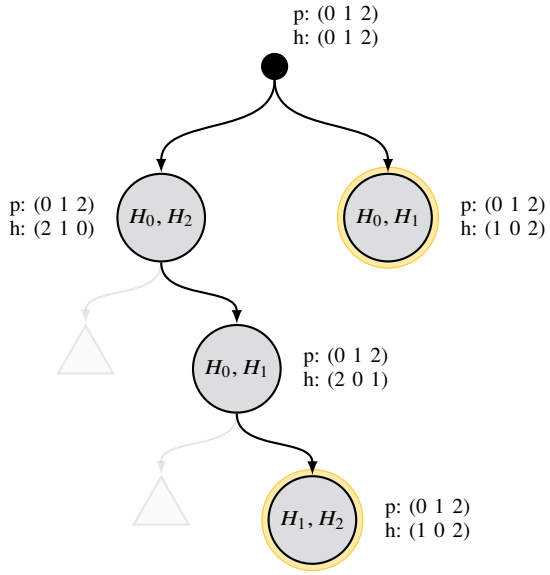

Figure 12. Two paths in the search graph lead to the same program-to-hardware qubit mapping.

C. Parallel Initial Layout Refinement

The initial program-to-hardware qubit mapping substantially influences the total SWAP overhead of the final circuit. To address this, prior work [19] introduced an iterative bidirectional strategy that refines the initial mapping through successive forward and backward routing passes. The authors in [20]

augment this idea by executing multiple independent "trials" in parallel to explore a broader range of initial configurations. Finally, the best candidate is chosen based on the number of inserted SWAPs. We adapt these strategies for the proposed A*-based mapping algorithm to explore different solution paths in the search-graph.

Unlike original SABRE-based implementations [19, 20], which generate a reconstruction plan during the final pass to rebuild the program, the proposed approach inserts SWAP operations directly into the IR. As a consequence, counting the number of inserted SWAPs during the final forward pass requires deep copying the IR for each thread. To maintain memory efficiency during parallel execution, we instead collect the number of inserted SWAPs during the final backward pass, therefore choosing the best initial layout for the final forward pass heuristically.

Whereas high-performance parallelism often requires third-party dependencies such as OpenMP, MLIR mitigates this overhead by providing native multithreading support and exposing primitives for parallel computation. We leverage these utilities to implement the parallel trial strategy. The `parallelForEach` function implements task-based parallelism by executing a callback function on a collection of initial configurations. The following snippet shows the implementation of this concurrent execution logic within the mapping pass, where `ntrials` is the number of random trials and `nqubits` is the number of qubits the architecture supports.

```
// Generate `ntrials` many random layouts:
SmallVector<Trial> trials;
trials.reserve(ntrials);
for (std::size_t i = 0; i < trials.size(); ++i) {
    trials.emplace_back(
        Layout::random(nqubits, seed);
    );
}

// Run the trials in parallel:
mlir::parallelForEach(&getContext(), trials,
    [&, this](Trial& trial) {
        refineLayout(func, arch, trial);
    });
```

D. SWAP Insertion

After selecting the most promising refined initial layout, the mapping pass performs a final “hot” routing phase by directly inserting SWAP operations in the IR. This strategy avoids the computational overhead of reconstructing the IR from scratch and the maintenance of auxiliary data structures typically required for a full rebuild.

To insert a SWAP operation into the IR, we utilize the MLIR `IRRewriter`. This process involves setting the insertion point through the rewriter’s API and instantiating the operation via `SWAPOp::create`. Because a SWAP operation exchanges the states of its two input qubits, the pass must rewire the output SSA values to maintain the intended program semantics.

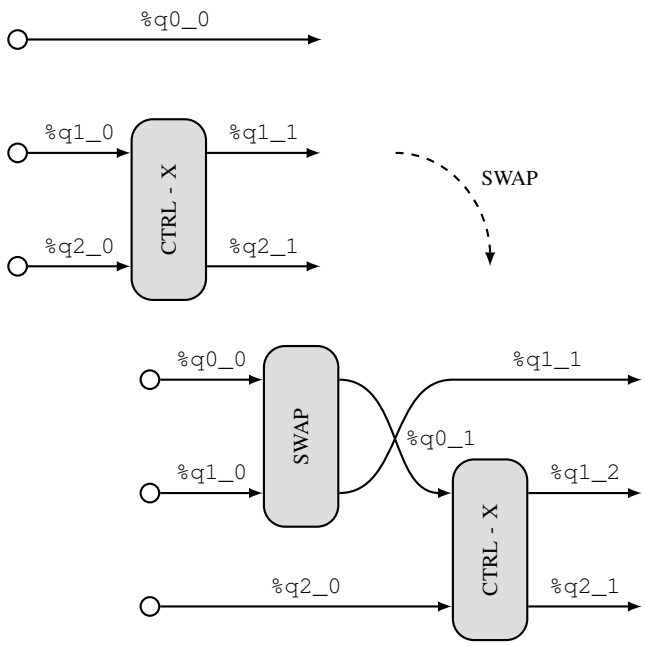

Figure 13. The rewiring process after SWAP insertion maintains the intended program semantics.

Example 16. *Figure 13 illustrates the rewiring process. The inserted SWAP operation exchanges the states of the input qubits `%q0_0` and `%q1_0`. Consequently, to maintain semantic equivalence, the operands of the controlled-X operation are updated such that it now consumes `%q0_1` as its input.*

However, inserting SWAP operations solely based on data-flow dependencies may result in a structurally invalid program, as the operations that produce the input SSA values are not guaranteed to precede the SWAP in the IR’s linear order. If such an SSA dominance violation occurs, the pass fails, and subsequent passes in the pass pipeline are aborted. To resolve this, we perform a final topological sort of the program using the `sortTopologically` utility provided by MLIR.

Example 17. *The following snippet illustrates an invalid IR (omitting allocations, sinks, and types for clarity). The highlighted SWAP operation consumes SSA values as operands that are defined later in the instruction sequence. This out-of-order definition violates SSA dominance, which requires that a value be defined before it is used.*

```
%q2_1, %q1_1 = qco.ctrl(%q2_0)
                    targets (%arg0 = %q1_0) { ... }
%q5_2, %q3_1 = qco.swap %q5_1, %q3_0
%q1_2, %q0_1 = qco.ctrl(%q1_1)
                    targets (%arg0 = %q0_0) { ... }
%q3_2, %q2_2 = qco.ctrl(%q3_1)
                    targets (%arg0 = %q2_1) { ... }
%q5_1, %q4_1 = qco.ctrl(%q5_0)
                    targets (%arg0 = %q4_0) { ... }
```

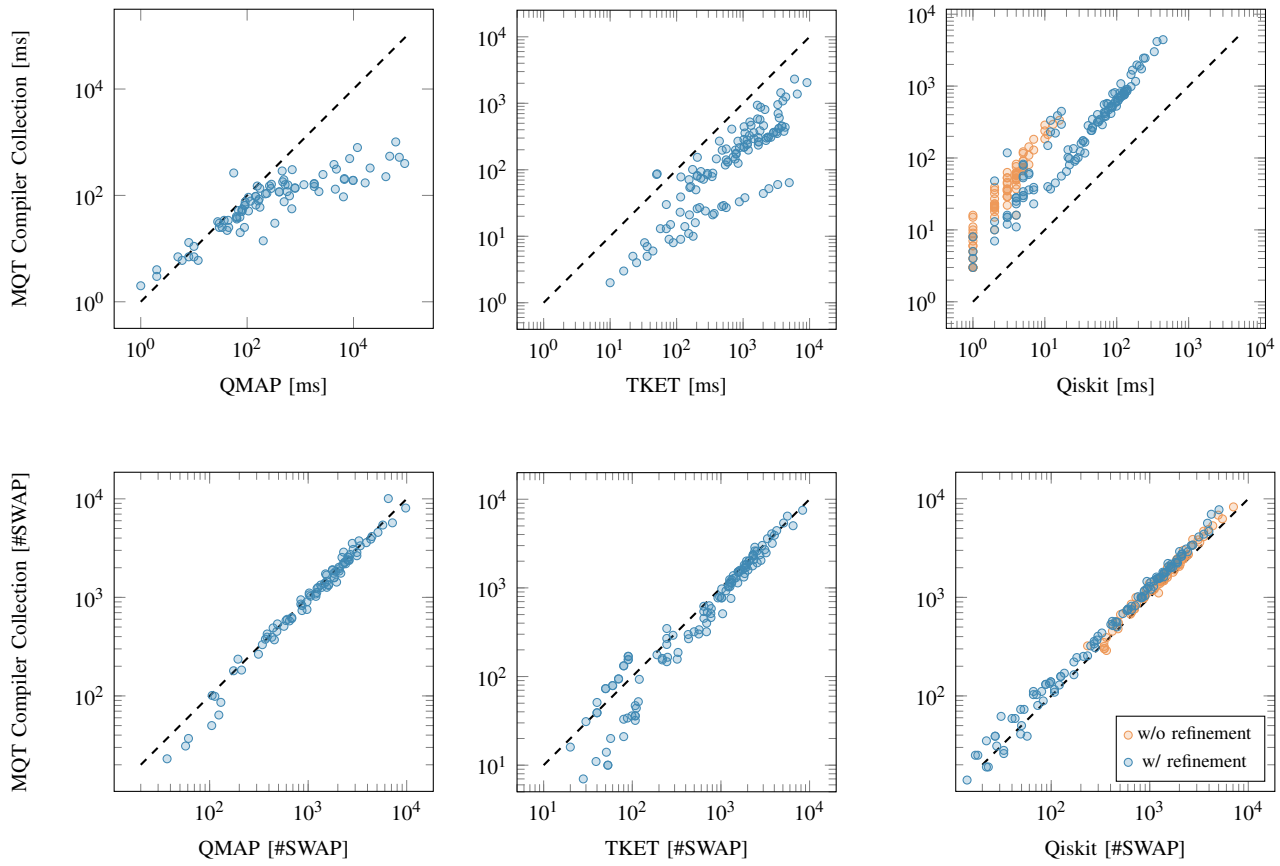

Figure 14. A comparison between QMAP, TKET, Qiskit, and the proposed solution.

V. Evaluation

As one of the most computationally complex tasks in an end-to-end quantum compilation pipeline, the runtime and solution quality plays an integral part of the overall performance of the compiler. Thus, this section evaluates the proposed solution and compares it to state-of-the-art implementations. The proposed solution is part of the Munich Quantum Toolkit (MQT) [26] and publicly available at github.com/munich-quantum-toolkit/core.

A. Experimental Setup

To ensure the reproducibility of our results, this section details the environment used for all benchmarks. We conducted our experiments on an Apple M5 Pro (18-core CPU) equipped with 24 GB of RAM running macOS Tahoe 26.3.

Library	Version
QMAP	2.6.0
TKET	2.1.77
Qiskit	2.4.0
MLIR (LLVM)	22.1.3

By implemented the benchmarks as a standalone C++20 executable, we eliminate the overhead of Python bindings and ensure peak performance. The project was compiled using Clang 22.1.3. Detailed version information for all external libraries can be found in the table above. To further optimize the MLIR framework's performance, we performed a local build with assertions disabled.

We target a 10×12 square lattice architecture for the mapping pass across all benchmarks, modeling IBM's state-of-the-art 120-qubit Nighthawk processor [27]. To evaluate the proposed solution under production-grade conditions, we benchmark almost hundred quantum programs with sizes ranging from 30 to 120 qubits. We specifically focus on the upper end of this range—the "utility-scale" spectrum to demonstrate scalability. These benchmarks are generated using MQT Bench [28] and the results are shown in Figure 14.

B. MQT Compiler Collection vs. QMAP

Both the QMAP library and the proposed solution implement the same A* mapping algorithm. However, while QMAP utilizes bespoke data structures, our solution is built upon the MLIR framework. The following results evaluate how leveraging MLIR as a compilation backbone impacts both runtime performance and solution quality.

For benchmarking, both implementations use an individual-gate strategy with 15 lookahead layers, and the proposed solution also incorporates two-qubit block aggregation. Regarding initial placement, QMAP employs a single bidirectional iteration (a forward and backward pass) on its dynamic layout, whereas our approach evaluates 18 random initial layouts to

more broadly explore the search space. Both algorithms apply a decay factor λ of 0.5.

On average, the proposed solution reduces SWAP insertions by 5 % and runtime by 46 %. The SWAP reduction is primarily due to the parallel random layout trials, which offer a more thorough search than QMAP's dynamic strategy. Notably, the 46 %. runtime improvement underscores the efficiency of the MLIR ecosystem, demonstrating that the A* algorithm can be significantly accelerated by leveraging MLIR primitives and optimized data structures.

C. MQT Compiler Collection vs. TKET

The TKET library offers state-of-the-art quantum compilation primitives including a specialized quantum circuit mapping algorithm [14]. This section demonstrates that by leveraging the MLIR framework, the proposed solution outperforms TKET's `LexiRoute` [21] method in terms of runtime and SWAP overhead.

In our experimental setup, TKET was configured to utilize its default placement strategy alongside the `LexiRouteRoutingMethod` for circuit routing. Driven by the significant runtime improvements observed in the previous benchmarks, we increased the lookahead depth of the proposed solution to 32 to further optimize the mapping quality. To prevent the decay factor from vanishing over this extended horizon, the decay parameter λ was adjusted to 0.7.

The proposed solution achieves a 13 % reduction in the number of inserted SWAP gates. While the search algorithm is a primary driver of this improvement, the dynamic initial layout strategy is similar to TKET's default, and therefore, this reduction may be partially attributed to the random trial mechanism. Furthermore, the implementation requires, on average, 74 % less runtime to compute and insert these SWAPs, surpassing the runtime improvements observed in the previous QMAP benchmark. These results highlight the efficiency of the A* search mapping algorithm and underscore the performance advantages of its MLIR-native implementation.

D. MQT Compiler Collection vs. Qiskit

The final evaluation compares the proposed solution to Qiskit's SABRE, a leading algorithm widely regarded as a state-of-the-art benchmark for circuit mapping.

The configuration of our solution remains identical to that in the previous section, except for the iteration count, which was increased to 4 to match SABRE's default setting. Both implementations were evaluated using 18 parallel trials.

To isolate the impact of layout refinement strategies, we conducted an additional benchmark comparing runtime and solution quality across a single forward pass using an identical, randomly selected initial layout for both tools.

The results indicate that SABRE outperforms the proposed solution in both metrics. Specifically, SABRE inserts 8 % fewer SWAPs and requires 28 % less runtime in the bidirectional benchmark. While our solution identifies better mappings (fewer SWAPs) in over one-third of the test cases, the results in the non-refined benchmark ultimately confirm SABRE's algorithmic advantage.

In summary, these results suggest that while the MLIR framework can significantly enhance the performance of existing mapping implementations, it is not a silver bullet. It cannot magically bridge the gap when compared to fundamentally more efficient algorithmic designs.

VI. Conclusions

This paper presents a significant step toward an MLIR-native, end-to-end quantum compilation pipeline by evaluating the framework's capacity for global optimization. We have demonstrated that by leveraging the MLIR ecosystem and its built-in primitives, it is possible to achieve substantial improvements in both runtime performance and solution quality compared to tailored implementations such as QMAP and TKET. These results underscore the performance advantages of integrating quantum primitives within a robust, general-purpose compiler framework.

However, as our comparison with Qiskit's SABRE illustrates, a high-performance foundation like MLIR complements rather than substitutes for sophisticated algorithmic design and theoretical complexity analysis: The underlying mapping logic remains the ultimate arbiter of success.

Future research could extend these concepts by incorporating hardware calibration data into the search algorithm to enable noise-aware quantum circuit mapping via standardized interfaces such as QDMI [29] . Furthermore, to support the growing demand for hybrid quantum-classical computing, subsequent iterations should incorporate classical control flow. Finally, leveraging MLIR's graph regions to model these hybrid programs offers a promising avenue to eliminate the computational overhead of topological sorting after SWAP insertion.

The MQT Compiler Collection is an open-source project and publicly available at github.com/munich-quantum-toolkit/core.

Acknowledgments

This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 101001318), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 563436708, and the European Union's Horizon Europe research and innovation programme under grant agreement No 101114305 ("MILLENION-SGA1" EU Project). Furthermore, this work was part of the Munich Quantum Valley, which the Bavarian state government supports with funds from the Hightech Agenda Bayern Plus, and has been supported by the BMFTR under grant number 01MQ25001I (FullStaQD).

While preparing the manuscript, Anthropic's Claude, OpenAI's GPT, and Google's Gemini were used to improve readability, spelling, grammar, and clarity throughout the manuscript. Each LLM output was reviewed by the authors and edited manually as needed. The authors take full responsibility for the final content.

References

[1] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, 2018. DOI: 10.22331/q-2018-08-06-79.

[2] J. Preskill, "Beyond NISQ: The Megaquop Machine," *ACM Transactions on Quantum Computing*, 2025. DOI: 10.1145/3723153.

[3] C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *International Symposium on Code Generation and Optimization (CGO)*, 2021. DOI: 10.1109/CGO51591.2021.9370308.

[4] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *International Workshop on Machine Learning and Programming Languages*, 2019. DOI: 10.1145/3315508.3329973.

[5] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Conference on Operating Systems Design and Implementation*, 2016.

[6] A. Paszke *et al.*, "PyTorch: an imperative style, high-performance deep learning library," in *Int'l Conf. on Neural Information Processing Systems*. 2019.

[7] T. Jin *et al.*, *Compiling ONNX Neural Network Models Using MLIR*, 2020. arXiv: 2008.08272.

[8] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler, "QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization," *ACM Transactions on Quantum Computing*, 2022. DOI: 10.1145/3491247.

[9] A. Peduri and S. Bhat, "QSSA: An SSA-based IR for Quantum Computing," in *International Conference on Compiler Construction*, 2022. DOI: 10.1145/3497776.3517772.

[10] A. McCaskey and T. Nguyen, *A MLIR Dialect for Quantum Assembly Languages*, 2021. DOI: 10.48550/arXiv.2101.11365.

[11] D. Ittah *et al.*, "Catalyst: a Python JIT compiler for auto-differentiablehybrid quantum programs," *Journal of Open Source Software*, 2024. DOI: 10.21105/joss.06720.

[12] M. B. Healy *et al.*, *Design and architecture of the IBM Quantum Engine Compiler*, 2024. DOI: 10.48550/arXiv.2408.06469.

[13] A. Zulehner, A. Paler, and R. Wille, "An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019. DOI: 10.1109/TCAD.2018.2846658.

[14] S. Sivarajah *et al.*, "TKET: A Retargetable Compiler for NISQ devices," *Quantum Science and Technology*, 2020. DOI: 10.1088/2058-9565/ab8e92.

[15] A. Javadi-Abhari *et al.*, *Quantum computing with Qiskit*, 2024. DOI: 10.48550/arXiv.2405.08810.

[16] L. Burgholzer *et al.*, "The MQT Compiler Collection: A Blueprint for a Future-Proof Quantum-Classical Compilation Framework," in *Design, Automation Test in Europe Conference*, 2026. DOI: 10.23919/DATE69613.2026.11539504.

[17] M. Y. Siraichi, V. F. D. Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation as a combination of subgraph isomorphism and token swapping," *Proceedings of the ACM on Programming Languages*, 2019. DOI: 10.1145/3360546.

[18] A. Botea, A. Kishimoto, and R. Marinescu, "On the Complexity of Quantum Circuit Compilation," *Proceedings of the International Symposium on Combinatorial Search*, 2021. DOI: 10.1609/socs.v9i1.18463.

[19] G. Li, Y. Ding, and Y. Xie, "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019. DOI: 10.1145/3297858.3304023.

[20] H. Zou *et al.*, *LightSABRE: A Lightweight and Enhanced SABRE Algorithm*, 2024. DOI: 10.48550/arXiv.2409.08368.

[21] A. Cowtan *et al.*, "On the Qubit Routing Problem," Schloss Dagstuhl, 2019. DOI: 10.4230/LIPICS.TQC.2019.5.

[22] W. Tang *et al.*, "AlphaRouter: Quantum Circuit Routing with Reinforcement Learning and Tree Search," in *International Conference on Quantum Computing and Engineering*, 2024. DOI: 10.1109/QCE60285.2024.00112.

[23] A. J. Adams *et al.*, "Asdf: A compiler for qwerty, a basis-oriented quantum programming language," in *International Symposium on Code Generation and Optimization*, Association for Computing Machinery, 2025. DOI: 10.1145/3696443.3708966.

[24] Y. Zhu *et al.*, *QLLVM: A Scalable Quantum-Classical Co-Compilation Framework based on LLVM*, 2026. arXiv: 2604.15094.

[25] R. Wille and L. Burgholzer, "MQT QMAP: Efficient quantum circuit mapping," in *International Symp. on Physical Design*, 2023. DOI: 10.1145/3569052.3578928.

[26] R. Wille *et al.*, "The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing," in *IEEE International Conference on Quantum Software (QSW)*, 2024. DOI: 10.1109/QSW62656.2024.00013, A live version of this document is available at https://mqt.readthedocs.io.

[27] IBM Quantum, *IBM Quantum Computing*, https://www.ibm.com/quantum/hardware, 22.4.2026, 2026.

[28] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing," *Quantum*, 2023, MQT Bench is available at https://mqt-bench.app. DOI: 10.22331/q-2023-07-20-1062.

[29] R. Wille *et al.*, "QDMI - Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack," in *International Conference on Quantum Computing and Engineering (QCE)*, 2024. DOI: 10.1109/QCE60285.2024.10411.