The MQT Compiler Collection

A Blueprint for a Future-Proof Quantum-Classical Compilation Framework

Lukas Burgholzer*†, Daniel Haag*, Yannick Stade*, Damian Rovara*, Patrick Hopf*†, Robert Wille*†
*Chair for Design Automation, Technical University of Munich, Germany
†Munich Quantum Software Company (MQSC), Garching bei München, Germany
{lukas.burgholzer, daniel.haag, yannick.stade, damian.rovara, patrick.hopf, robert.wille}@tum.de
cda.cit.tum.de/research/quantum

Abstract*—As the capabilities of quantum computing hardware continue to rise, algorithms that exploit them are becoming increasingly complex. These developments increase the need for sophisticated compilation frameworks that translate high-level algorithms into executable code. In the past, most solutions were built with a *quantum-first* approach and handled mostly pure quantum programs without classical elements such as structured control flow. However, developments in quantum algorithms, error correction, and optimization, as well as the integration into high-performance computing (HPC) environments, depend on such classical elements. As quantum-first approaches increasingly struggle to handle these concepts, *classical-first* approaches are becoming a promising alternative. In this work, we present the *MQT Compiler Collection*, a blueprint for a future-proof *quantum-classical* compilation framework built on the *Multi-Level Intermediate Representation* (MLIR). After years of experience with the quantum-first approach and its shortcomings, we propose a framework that embraces core MLIR concepts to support the *full compilation pipeline* from high-level algorithms to hardware-specific instructions. The proposed architecture is designed from the ground up to support complex optimizations beyond, e.g., simple gate cancellation. It is publicly available at github.com/munich-quantum-toolkit/core.

I. Introduction

With the rapid growth of quantum software tools and the increasing capabilities of quantum hardware, robust compilation has become a critical bottleneck in the quantum computing stack [1]. Compilation frameworks serve as the bridge between a diverse ecosystem of high-level programming languages and a heterogeneous zoo of hardware platforms—ranging from superconducting circuits, trapped ions, and neutral atoms to photonics and beyond. Historically, this bridge has been built using a *quantum-first* approach (see the left half of Fig. 1). Early *intermediate representations* (IRs), such as OpenQASM 2, established a standard for allocating qubits and defining gate sequences, effectively serving as the assembly language of early quantum computing.

However, quantum computing is inherently hybrid, relying on classical computing for control, error correction, and optimization, particularly as quantum computers are integrated into classical high-performance computing (HPC) environments [1]–[4]. Features taken for granted in classical computing, such as structured control flow, variable scoping, and general-purpose logic, are often entirely absent from quantum-first IRs or added only as ad-hoc extensions. Although newer standards such as OpenQASM 3 [5] aim to address these gaps, many established tools, including Qiskit [6], Cirq [7], Amazon Braket [8], BQSKit [9], and the Munich Quantum Toolkit (MQT; [10]), remain constrained by legacy architectures that treat classical computing as a second-class citizen or a mere addition.

In contrast, *classical-first* approaches (see the right half of Fig. 1) take a fundamentally different path. Instead of reinventing the wheel, they build on decades of classical compiler development and propose to extend established frameworks with quantum concepts. This approach views the quantum computer as an accelerator—similar to a GPU—integrated into a larger classical computing environment [11], [12]. The *Quantum Intermediate Representation* (QIR; [13]) marked a significant shift in this direction by extending the widely used LLVM [14] framework for quantum computing. However, LLVM usually operates on a relatively low level of abstraction. Its IR is designed for instruction-level optimization and code generation, which makes it challenging to represent and optimize high-level quantum algorithms that might, for example, contain structured control flow. This gap has led to the rising popularity of the *Multi-Level Intermediate Representation* (MLIR; [15]).

MLIR addresses these limitations by enabling the definition of custom dialects at various levels of abstraction. It allows for the seamless composition of compilation passes within a shared infrastructure, promoting modularity, reuse, and interoperability. However, adopting MLIR poses substantial challenges for tools rooted in the quantum-first paradigm, as it requires moving beyond Python-based rapid prototyping toward disciplined C++ software engineering with a strong focus on stability and maintainability.

In this work, we present the *MQT Compiler Collection* as a blueprint implementation of a future-proof *quantum-classical* compilation framework built on MLIR. Drawing on our domain expertise from the quantum-first era, we embrace and combine the best of both worlds. Building on the principles of MLIR, the proposed implementation supports the *full compilation pipeline* from high-level algorithms to hardware-specific instructions.

The remainder of this paper is organized as follows. Section II provides the necessary background on MLIR and discusses related work. In Section III, we motivate our design choices and outline the core principles of the proposed architecture. Section IV details the technical implementation of the *MQT Compiler Collection*. Section V concludes the paper.

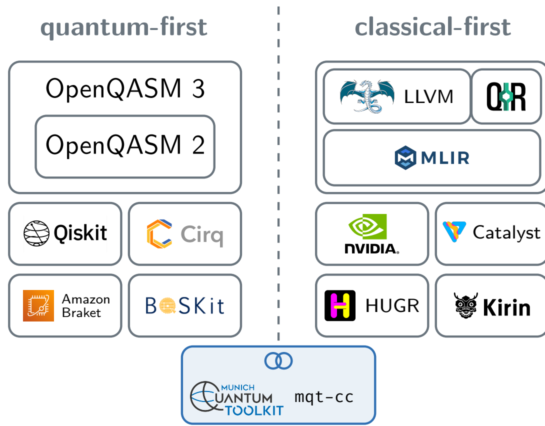


Figure 1. The *MQT Compiler Collection* (mqt-cc) combines quantum and classical concepts to build a future-proof compilation framework.

II. Background and Related Work

To ensure that this article is self-contained, we briefly review the relevant aspects of MLIR. Subsequently, we discuss related approaches and how they address the challenge of defining an infrastructure for quantum compilation.

A. Multi-Level Intermediate Representation

MLIR addresses the fragmentation of (classical) compilation infrastructure by providing a unified, extensible framework that supports multiple levels of abstraction. At its core is the concept of *dialects*. Much like natural languages, each dialect defines its own vocabulary in the form of operations, types, and attributes. Crucially, these dialects can be mixed within a single program, allowing different concepts to be expressed side by side. For instance, the `scf` dialect provides concepts for representing structured control flow (like loops and conditionals), while the `arith` dialect enables the description of arithmetic operations. When implementing a new dialect for a specific domain—such as quantum computing—developers can leverage these existing dialects without having to reimplement common functionality.

To make these concepts more graspable, let us have a look at one of the toy examples from the MLIR documentation.

Example 1. *The snippet below defines a tensor, transposes it, computes its product with itself, and then prints the result.* `toy.constant`*,* `toy.transpose`*,* `toy.mul`*, and* `toy.print` *are* operations*, and* `%0`*,* `%1`*, and* `%2` *are the* values *returned by the operations. The keyword* `dense` *signifies the built-in* attribute *for multi-dimensional arrays, whereas* `tensor` *is a built-in* type.

Toy.mlir

```
%0 = toy.constant dense<[
        [1.000000e+00, 2.000000e+00, 3.000000e+00],
        [4.000000e+00, 5.000000e+00, 6.000000e+00]
     ]> : tensor<2x3xf64>
%1 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
%2 = toy.mul %1, %1 : tensor<3x2xf64>
toy.print %2 : tensor<3x2xf64>
```

The *multi-level* nature of the MLIR framework allows for a progressive compilation strategy. A program can start at a high level of abstraction, closely matching the source language or algorithm, and be successively "lowered" to more concrete representations. This process is facilitated by the *conversion* framework, which handles the transformation of operations from one dialect to another. During lowering, operations from high-level dialects can be gradually replaced by operations from lower-level dialects. In the quantum context, this means one can, e.g., describe a high-level quantum algorithm and progressively lower it to a program that can actually be executed on some specific quantum(-classical) hardware—all within the same infrastructure.

At each stage of the lowering process, MLIR provides powerful optimization tools to simplify the underlying program representation. One can define *canonicalization* patterns for operations—rules that perform local simplifications (e.g., removing self-inverse gates). More complex or global optimizations can be implemented as *transformation passes*. The framework provides standardized interfaces for these passes, e.g., for traversing and manipulating the IR—making it easier to write robust compiler analyses and transformations.

Furthermore, MLIR includes a flexible *translation* framework for importing and exporting external IRs to and from MLIR, respectively. This allows importing established representations, such as OpenQASM, into MLIR for optimization, and exporting the results back to other formats or directly to executable code.

B. Related Work

The shift towards classical-first frameworks has been a major trend in recent years, driven largely by the need for scalable and integrated quantum-classical compilation. A prominent example is Microsoft's Azure Quantum [16], which heavily leverages QIR to build a robust ecosystem for quantum development. Similarly, many other industrial players have begun building their tools with a strong connection to classical compilation infrastructure.

NVIDIA's CUDA-Q [17] focuses on streamlining hybrid application development by providing a unified programming model for CPUs, GPUs, and QPUs. Central to its architecture is the Quake dialect, which brings quantum concepts into the MLIR ecosystem. Xanadu's Catalyst [18], inspired by the design of QIRO [19], also leverages MLIR to enable just-in-time compilation of hybrid quantum programs. It captures Python-based programs and translates them into a representation that can be optimized and executed efficiently.

Other approaches draw inspiration from MLIR's design philosophy without directly building upon it. Quantinuum's HUGR [20] introduces a hierarchical, graph-based IR for mixed quantum-classical programs. It is implemented in Rust and emphasizes a unified representation for various stages of compilation. QuEra's Kirin [21] follows a similar path, aiming to provide a flexible compilation infrastructure for embedded domain-specific languages. These efforts collectively highlight the growing consensus that modular, multi-level IRs are essential for the future of quantum compilation.

III. Motivation

Building upon insights from both quantum-first and classical-first compilation efforts, we present the *MQT Compiler Collection*—a comprehensive quantum-classical compilation framework built on MLIR that supports the full pipeline from high-level algorithms to hardware-specific instructions. Rather than viewing these paradigms as competing approaches, we seek to combine their respective strengths: the domain expertise and optimization techniques cultivated in quantum-first tools with the robust infrastructure and proven compiler methodology of classical-first frameworks. By embracing and integrating concepts from both worlds, the MQT Compiler Collection serves as a blueprint for future-proof compilation infrastructure. We argue that the core principles of MLIR—modularity, specialized dialects, and progressive lowering—provide an ideal foundation for achieving this synthesis.

A. The MLIR Opportunity

The need for a compilation infrastructure that handles both quantum and classical aspects is evident. However, adopting MLIR is not just about using a specific software framework; it is about adopting a *mindset* of progressive lowering and modular abstraction. Rather than building a monolithic compiler that compiles high-level algorithms into hardware pulses in a single step, the MLIR philosophy encourages breaking the problem down into smaller, more manageable steps.

While one could implement such a layered architecture from scratch, MLIR offers a proven infrastructure for defining dialects, conversions, transformation passes, and more. This is particularly valuable for the quantum community, as it avoids the need to build and maintain custom implementations for common compiler tasks. Since quantum computers are increasingly viewed not as standalone devices but as specialized accelerators working in tandem with CPUs and GPUs (e.g., in HPC environments), a framework inherently designed for such heterogeneity is a compelling choice. By treating quantum operations as just another dialect within the broader compilation landscape, we can reuse existing analyses and optimizations for the classical parts of the program, such as loop optimizations and constant folding. This is why we have opted to implement the proposed quantum compilation framework in MLIR, fully embracing the classical-first mindset by adopting established MLIR patterns and techniques.

B. Design Philosophy

After years of experience with the quantum-first approach, we aim to apply our expertise to build a future-proof quantum-classical compilation framework. The proposed framework retains the domain-specific capabilities of our previous efforts (see, e.g., Refs. [10] and [22]) while leveraging MLIR's robust infrastructure to overcome previous limitations. Our goal is to provide a blueprint implementation that supports the full compilation pipeline from high-level algorithms to hardware-specific instructions. The proposed architecture is built to be extended and adapted as the field evolves. To achieve this, our design philosophy is guided by several key principles:

- **Full-Stack Support**: We explicitly target the full compilation stack and support the full compilation pipeline from high-level algorithms to hardware-specific instructions. Modifiers and custom gate definitions ensure extensibility and flexibility throughout the lowering process.
- **Next-Generation Programs**: We look beyond static circuits to support structured quantum programs with control flow, dynamic circuits, and hybrid quantum-classical workflows.
- **Comprehensive Optimization**: The proposed architecture is designed to support a fully-fledged optimization framework. We aim beyond simple gate cancellation and peephole optimizations to enable complex passes such as high-level synthesis, placement and routing, and non-local optimizations.
- **Rigorous MLIR Adoption**: We do not just use MLIR as a mean to an end; we embrace its methodology. This means not reinventing the wheel, but leveraging existing dialects (e.g., `arith` for arithmetic operations) as well as built-in features and data structures.
- **Modern Standards**: We adopt modern standards to ensure longevity and compatibility. We build against the latest stable releases of LLVM/MLIR, embrace QIR 2.0 (with opaque pointer support), and support OpenQASM 3.x.

To this end, the architecture leverages two distinct dialects, each designed around a fundamental programming paradigm:

- **QC (Quantum Circuit)**: The QC dialect serves as a natural interface for input/output and hardware mapping. It utilizes *imperative* concepts and *reference* semantics to mirror the physical reality of quantum devices.
- **QCO (Quantum Circuit Optimization)**: The QCO dialect adopts a (first-order) *functional* approach with *value semantics*, providing an explicit representation ideally suited for complex optimizations and transformations.

By supporting both representations within the same framework, we can leverage the strengths of each model without compromise. As illustrated in Fig. 2, this enables a seamless flow from high-level language descriptions to hardware-executable code, while providing opportunities for optimization at different levels of abstraction.

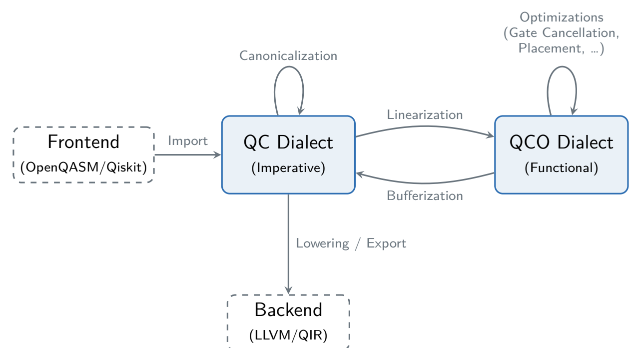


Figure 2. The MQT Compiler Collection pipeline.

The compilation flow starts with a circuit description in a language or framework such as OpenQASM or Qiskit. This input is imported into the QC dialect, where the program structure is preserved in an imperative form. From there, the representation is converted to the QCO dialect by converting qubit references into quantum state values (also called *linearization*). In this functional representation, various optimization passes are applied to the program. Canonicalization and gate cancellation prune redundant operations, while backend-specific passes—such as a mapping algorithm for superconducting qubit architectures—adapt the circuit to device constraints. These passes are merely examples of the transformations enabled by the framework; the infrastructure makes it easy to add further optimizations as needed. Once optimized, the circuit is converted back to the QC dialect by reconstructing qubit references from the data flow. Finally, the result is lowered to low-level representations, such as LLVM IR and QIR, enabling execution on simulators or physical quantum hardware [23]. The infrastructure is easily extensible, as demonstrated by an integration of PennyLane via an MLIR plugin [24].

IV. The MQT Compiler Collection

In this section, we present the technical details of the *MQT Compiler Collection*, an open-source implementation of the architecture proposed in Section III-B. We detail the design of the two core dialects—QC and QCO—and highlight how their imperative and functional natures complement each other to facilitate both interoperability and optimization. Due to space limitations, we focus on key concepts and representative examples to illustrate the core ideas. The full implementation is available at github.com/munich-quantum-toolkit/core.

A. The QC Dialect: An Imperative Interface

The QC dialect is designed to act as the primary interface between external quantum programming languages and the compiler. Most established languages, such as OpenQASM, and frameworks, such as Qiskit, follow an *imperative* programming model, where qubits are treated as mutable resources that are modified in place by sequential operations via side effects. The QC dialect mirrors these *reference semantics* in its `!qc.qubit` type. It defines a comprehensive set of operations:

- **Resource Management**: `qc.alloc` and `qc.dealloc` for allocating and deallocating qubits as well as support for qubit registers.
- **Standard Gate Library**: A comprehensive library of well-known gates (e.g., `qc.h`, `qc.x`, `qc.rz`).
- **Non-Unitary Operations**: `qc.measure` for extracting information and `qc.reset` for re-initializing qubits.
- **Modifiers**: First-class support for controlled (`qc.ctrl`), inverse (`qc.inv`), and power (`qc.pow`) modifiers.
- **Structure**: Operations for grouping instructions and defining custom gates.

All unitary operations, including standard as well as custom gates and gates wrapped in modifiers, implement a common `UnitaryOpInterface` that provides a unified API for introspection and manipulation (see Fig. 3).

QCInterfaces.td

```
def UnitaryOpInterface : OpInterface<"UnitaryOpInterface"> {
  let methods = [
    InterfaceMethod<" ... ", "size_t",    "getNumQubits",   (ins)>,
    InterfaceMethod<" ... ", "size_t",    "getNumTargets",  (ins)>,
    InterfaceMethod<" ... ", "size_t",    "getNumControls", (ins)>,
    InterfaceMethod<" ... ", "Value",     "getQubit",       (ins "size_t":$i)>,
    InterfaceMethod<" ... ", "Value",     "getTarget",      (ins "size_t":$i)>,
    InterfaceMethod<" ... ", "Value",     "getControl",     (ins "size_t":$i)>,
    InterfaceMethod<" ... ", "size_t",    "getNumParams",   (ins)>,
    InterfaceMethod<" ... ", "Value",     "getParameter",   (ins "size_t":$i)>,
    InterfaceMethod<" ... ", "bool",      "isControlled",   (ins)>,
    InterfaceMethod<" ... ", "bool",      "isSingleQubit",  (ins)>,
    InterfaceMethod<" ... ", "bool",      "isTwoQubit",     (ins)>,
    InterfaceMethod<" ... ", "StringRef", "getBaseSymbol",  (ins)>,
  ];}
```

Figure 3. Abbreviated TableGen definition of the `UnitaryOpInterface`.

Example 2. *Consider the creation of a Bell pair. In the QC dialect, the reference semantics are evident: we allocate two qubits and apply operations to them sequentially.* `%q0` *and* `%q1` *represent the qubit references themselves, which are reused throughout the program. The data flow is* implicit*—defined only by the program order.*

QC_Bell_Pair.mlir

```
%q0 = qc.alloc : !qc.qubit
%q1 = qc.alloc : !qc.qubit
qc.h %q0 : !qc.qubit
qc.ctrl(%q0) {
  qc.x %q1 : !qc.qubit
} : !qc.qubit
qc.dealloc %q0 : !qc.qubit
qc.dealloc %q1 : !qc.qubit
```

This representation is ideal for interacting with other tools and for the final mapping to hardware, but the implicit data flow can obscure dependencies, making advanced optimizations challenging. That said, simple cleanup tasks can already be implemented in the QC dialect to ensure that the input is not unnecessarily bloated.

Example 3. *A common cleanup task is removing pairs of allocation and deallocation operations that have no intervening computations. In the QC dialect, this requires verifying that the allocated qubit has no uses other than its deallocation. The optimization pattern matches a* `DeallocOp`*, looks up the defining* `AllocOp`*, and checks if the qubit is used only once.*

RemoveAllocDealloc.cpp

```
struct RemoveAllocDeallocPair final :
    OpRewritePattern<DeallocOp> {
  using OpRewritePattern::OpRewritePattern;
  LogicalResult matchAndRewrite(
      DeallocOp op,
      PatternRewriter& rewriter) const override {
    // Return if qubit is not defined by AllocOp
    auto allocOp =
        op.getQubit().getDefiningOp<AllocOp>();
    if (!allocOp) return failure();
    // Return if qubit has more than one use
    if (!op.getQubit().hasOneUse()) return failure();
    // Erase alloc-dealloc pair
    rewriter.eraseOp(op);
    rewriter.eraseOp(allocOp);
    return success();
  }
};
```

Applying this pattern simplifies the IR as shown below:

QC_Alloc_Dealloc.mlir
```
%q0 = qc.alloc : !qc.qubit
%q1 = qc.alloc : !qc.qubit
qc.h %q0 : !qc.qubit
qc.dealloc %q0 : !qc.qubit
qc.dealloc %q1 : !qc.qubit
```

Optimized.mlir
```
%q0 = qc.alloc : !qc.qubit
qc.h %q0 : !qc.qubit
qc.dealloc %q0 : !qc.qubit
```

B. The QCO Dialect: A Functional Optimization Graph

To enable more complex circuit transformations, we have designed the QCO dialect. In contrast to QC, QCO adopts *value semantics* that are typical for *functional* programming languages. Here, operations do not modify qubits in place. Instead, they consume input quantum states and produce new, updated output states—effectively modeling qubits as linear types. This transforms the circuit into a dataflow graph where dependencies are *explicit*.

This structure is conceptually identical to the directed acyclic graphs (DAGs) that frameworks like Qiskit build internally to analyze and optimize circuits. However, while those frameworks have generally implemented custom DAG data structures and algorithms from scratch, the QCO dialect directly uses the use-def chains of MLIR values to represent this graph. In other words, the QCO dialect natively embeds the DAG representation within the IR itself. This design choice allows us to leverage MLIR's built-in analysis and transformation capabilities directly on the dataflow graph.

Example 4. *Revisiting the Bell pair circuit in the QCO dialect illustrates the explicit data flow. The* `qco.h` *operation consumes the initial state of the first qubit (*`%q0_0`*) and produces a new state (*`%q0_1`*). The subsequent controlled operation explicitly depends on* `%q0_1` *as its control input. Because every operation produces new values, the dependency graph is encoded directly in the program structure.*

QCO_Bell_Pair.mlir
```
%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 (%targ_in = %q1_0) {
  %targ_out = qco.x %targ_in : !qco.qubit → !qco.qubit
  qco.yield %targ_out
} : ({!qco.qubit}, {!qco.qubit}) →
    ({!qco.qubit}, {!qco.qubit})
qco.dealloc %q0_2 : !qco.qubit
qco.dealloc %q1_1 : !qco.qubit
```

C. Optimizations Made Easy

The explicit data flow of the QCO dialect significantly simplifies the implementation of optimization passes. Many circuit optimizations can be expressed as simple pattern-matching rules (canonicalization patterns) on the dataflow graph. While we have already shown an example of such a pattern in the QC dialect, it was only reasonable to implement it there since it was enough to check the *number* of uses. Slightly more complex optimizations, such as gate cancellation, immediately profit from the value semantics of QCO. Because dependencies are explicit, detecting if two operations act on the same qubit in sequence is trivial—one simply checks if the input of the second operation is the output of the first.

Example 5. *Hermitian gates like the Hadamard gate satisfy* $H^\dagger H = I$*, i.e., they are self-inverse. Thus, any two adjacent Hadamard gates cancel each other out. Implementing this check in QCO is concise because the adjacency is guaranteed by the linear data flow. The following shows a generalized pattern for cancelling any two types of consecutive inverse operations:*

InverseCancellation.cpp
```
template <typename InverseOpType, typename OpType>
inline mlir::LogicalResult
removeInversePairOneTargetZeroParameter(
    OpType op,
    mlir::PatternRewriter& rewriter) {
  // Return if previous operation is not InverseOpType
  auto prevOp = op.getInputQubit(0)
      .template getDefiningOp<InverseOpType>();
  if (!prevOp) return failure();
  // Replace both operations with their input
  // This effectively erases them from the IR
  rewriter.replaceOp(prevOp,
                     prevOp.getInputQubit(0));
  rewriter.replaceOp(op, op.getInputQubit(0));
  return success();
}
```

Applying this pattern simplifies the IR as shown below:

QCO_Hadamard.mlir
```
%q0_0 = qco.alloc : !qco.qubit
%q0_1 = qco.x %q0_0 : !qco.qubit → !qco.qubit
%q0_2 = qco.h %q0_1 : !qco.qubit → !qco.qubit
%q0_3 = qco.h %q0_2 : !qco.qubit → !qco.qubit
qco.dealloc %q0_3 : !qco.qubit
```

Optimized.mlir
```
%q0_0 = qco.alloc : !qco.qubit
%q0_1 = qco.x %q0_0 : !qco.qubit → !qco.qubit
qco.dealloc %q0_1 : !qco.qubit
```

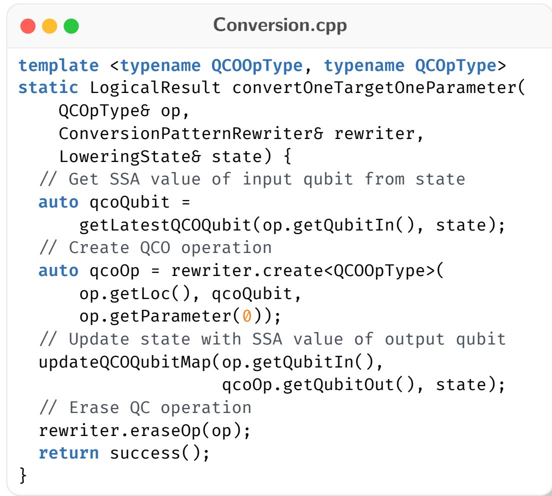
Beyond local canonicalization patterns, the proposed framework supports sophisticated transformations. We have ported the QMAP mapping algorithm [25] to MLIR as a representative example of a complex, backend-aware pass. This pass maps quantum circuits to architectures with limited connectivity. Building on the existing MLIR infrastructure enabled us to support routing for circuits with structured control flow—a feature that was not possible in the previous open-source implementation without significant engineering effort [26].

D. Bridging the Paradigms

To connect the imperative world of QC with the functional world of QCO, we provide robust bidirectional conversion

passes. These passes automate the translation between reference and value semantics. Converting from QC to QCO involves “linearization” or building the dataflow graph via tracking the latest value of each qubit and introducing new values for each operation. Conversely, converting back to QC involves “bufferization” or reconstructing qubit references from the data flow by ensuring that each qubit value is replaced with the appropriate reference.

Example 6. *The code below shows the core logic for converting a single-qubit gate from QC to QCO. The compiler tracks the current value for each qubit using* `getLatestQCOQubit`*. It creates the corresponding QCO operation using the latest value and updates the state with the new output value using* `updateQCOQubitMap`*.*



Conversion.cpp

```
template <typename QCOOpType, typename QCOpType>
static LogicalResult convertOneTargetOneParameter(
    QCOpType& op,
    ConversionPatternRewriter& rewriter,
    LoweringState& state) {
  // Get SSA value of input qubit from state
  auto qcoQubit =
      getLatestQCOQubit(op.getQubitIn(), state);
  // Create QCO operation
  auto qcoOp = rewriter.create<QCOOpType>(
      op.getLoc(), qcoQubit,
      op.getParameter(0));
  // Update state with SSA value of output qubit
  updateQCOQubitMap(op.getQubitIn(),
                    qcoOp.getQubitOut(), state);
  // Erase QC operation
  rewriter.eraseOp(op);
  return success();
}
```

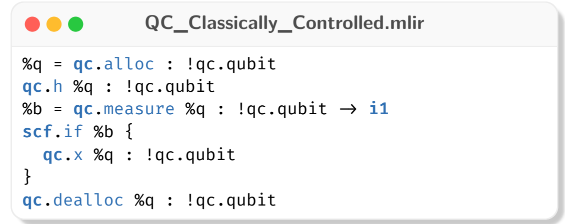
Conversions for more complex constructs, such as the control modifier, require delicate threading of the qubit values to ensure the linearity constraints are upheld. However, the modular nature of MLIR’s conversion framework makes it straightforward to implement and maintain these transformations.

E. Structured Control Flow

As stressed in Section III-B, any future-proof quantum-classical compilation framework must support structured quantum programs. That is, it must support programs that do not only consist of a static sequence of quantum operations but also include classical control flow that may, additionally, depend on measurement outcomes.

MLIR’s built-in `scf` dialect provides operations for defining loops (such as `scf.for` and `scf.while`) and conditionals (such as `scf.if`), along with predefined canonicalization patterns. These operations can be seamlessly integrated into an IR utilizing the QC dialect. Let us illustrate this with a simple example.

Example 7. *The following program demonstrates a native reset operation using structured control flow. A qubit is prepared in a superposition state via a Hadamard gate. The qubit is subsequently measured, yielding a classical bit $c \in \{0,1\}$. If the measurement outcome is $c = 1$, the qubit is flipped using an X gate; otherwise, no operation is performed. As a result, the qubit is always reset to the $|0\rangle$ state.*



QC_Classically_Controlled.mlir

```
%q = qc.alloc : !qc.qubit
qc.h %q : !qc.qubit
%b = qc.measure %q : !qc.qubit → i1
scf.if %b {
  qc.x %q : !qc.qubit
}
qc.dealloc %q : !qc.qubit
```

This goes to show how natural it is to express dynamic quantum circuits using structured control flow in the proposed framework. Moreover, the integration with MLIR’s `scf` dialect allows us to directly leverage existing optimizations for these constructs. For instance, loop unrolling and conditional simplifications can be applied without any additional effort.

V. Conclusion

In this work, we have presented the *MQT Compiler Collection*—a blueprint implementation that demonstrates how to build future-proof quantum-classical compilation infrastructure on MLIR. By combining insights from both quantum-first and classical-first paradigms, we have shown that it is possible to support the full compilation pipeline from high-level algorithms to hardware-specific instructions within a unified, modular framework. At its core, the proposed dual-dialect architecture exemplifies the power of MLIR’s design philosophy: the imperative QC dialect provides a natural interface for external tools and hardware, while the functional QCO dialect enables sophisticated optimizations through value semantics.

Beyond the technical contributions, this work serves as a community resource. Developed entirely as open-source software, the MQT Compiler Collection offers a concrete starting point for researchers and practitioners looking to adopt MLIR for quantum compilation or to extend the framework with new capabilities. We hope that by sharing both our implementation and the lessons learned from bridging quantum-first expertise with classical-first infrastructure, we can accelerate the development of robust, interoperable compilation frameworks across the quantum computing ecosystem. The full implementation is publicly available as part of MQT Core [22] at github.com/munich-quantum-toolkit/core.

Acknowledgments

The authors acknowledge funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program grant agreement No. 101001318 and No. 101114305 (“MILLENION-SGA1” EU Project), and the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus. Furthermore, this work was supported by the BMFTR under grant numbers 13N17298 (SYNQ) and 01MQ25001I (FullStaQD), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant numbers 563402549 and 563436708, and the Austrian Research Promotion Agency (FFG) together with the states of Upper Austria and Tyrol within the COMET module Quantum Algorithm Engineering (FFG) under grant number 923923.

REFERENCES

[1] L. Burgholzer, J. Echavarria, P. Hopf, *et al.*, “The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC,” in *Int’l Conf. on High Performance Computing in Asia Pacific Region*, 2026. DOI: 10.1145/3773656.3773669.

[2] E. Mansfield, S. Seegerer, P. Vesanen, *et al.*, “First practical experiences integrating quantum computers with HPC resources: A case study with a 20-qubit superconducting quantum computer,” in *Workshops of the Int’l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2025. DOI: 10.1145/3731599.3767551.

[3] A. Shehata, P. Groszkowski, T. Naughton, M. Gopalakrishnan Meena, E. Wong, D. Claudino, *et al.*, “Bridging paradigms: Designing for HPC-Quantum convergence,” *Future Generation Computer Systems*, 2026. DOI: 10.1016/j.future.2025.107980.

[4] T. S. Humble, A. McCaskey, D. I. Lyakh, M. Gowrishankar, A. Frisch, and T. Monz, “Quantum Computers for High-Performance Computing,” *IEEE Micro*, 2021. DOI: 10.1109/MM.2021.3099140.

[5] A. W. Cross, A. Javadi-Abhari, T. Alexander, *et al.*, “OpenQASM 3: A broader and deeper quantum assembly language,” *ACM Transactions on Quantum Computing*, 2022. DOI: 10.1145/3505636. arXiv: 2104.14722.

[6] A. Javadi-Abhari, M. Treinish, K. Krsulich, *et al.*, *Quantum computing with Qiskit*, 2024. arXiv: 2405.08810.

[7] Cirq Developers, *Cirq*, 2025, https://github.com/quantumlib/Cirq.

[8] Braket Developers, *Amazon Braket: a fully managed quantum computing service provided by AWS*, 2019, https://aws.amazon.com/braket/.

[9] E. Younis, C. C. Iancu, W. Lavrijsen, M. Davis, and E. Smith, *Berkeley Quantum Synthesis Toolkit (BQSKit) v1*, 2021. DOI: 10.11578/dc.20210603.2.

[10] R. Wille, L. Berent, T. Forster, *et al.*, “The MQT handbook: A summary of design automation tools and software for quantum computing,” in *Int’l Conf. on Quantum Software*, 2024. DOI: 10.1109/QSW62656.2024.00013. arXiv: 2405.17543, A live version of this document is available at https://mqt.readthedocs.io.

[11] E. Kaya, J. Echavarria, M. N. Farooqi, *et al.*, “A software platform to support disaggregated quantum accelerators,” in *Workshops of the Int’l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2024. DOI: 10.1109/SCW63240.2024.00205.

[12] I. Sitdikov, M. E. Sahin, U. Bacher, *et al.*, *Quantum resources in resource management systems*, 2025. arXiv: 2506.10052.

[13] T. Lubinski, C. Granade, A. Anderson, *et al.*, “Advancing hybrid quantum–classical computation with real-time execution,” *Frontiers in Physics*, 2022. DOI: 10.3389/fphy.2022.940293.

[14] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Int’l Symp. on Code Generation and Optimization (CGO)*, 2004. DOI: 10.1109/CGO.2004.1281665.

[15] C. Lattner, M. Amini, U. Bondhugula, *et al.*, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *Int’l Symp. on Code Generation and Optimization (CGO)*, 2021. DOI: 10.1109/CGO51591.2021.9370308.

[16] Microsoft Corporation, *Azure Quantum*, https://azure.microsoft.com/solutions/quantum-computing.

[17] The CUDA-Q development team, *CUDA-Q*, https://github.com/NVIDIA/cuda-quantum.

[18] D. Ittah, A. Asadi, E. Ochoa Lopez, *et al.*, “Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs,” *Journal of Open Source Software*, 2024. DOI: 10.21105/joss.06720.

[19] 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.

[20] M. Koch, A. Borgna, S. Sivarajah, *et al.*, *HUGR: A Quantum-Classical Intermediate Representation*, 2025. arXiv: 2510.11420.

[21] Kirin contributors, *Kirin*, 2024, https://github.com/QuEraComputing/kirin.

[22] L. Burgholzer, Y. Stade, T. Peham, and R. Wille, “MQT Core: The backbone of the Munich Quantum Toolkit (MQT),” *Journal of Open Source Software*, 2025. DOI: 10.21105/joss.07478.

[23] Y. Stade, L. Burgholzer, and R. Wille, “Towards Supporting QIR: Steps for Adopting the Quantum Intermediate Representation,” in *Workshops of the Int’l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2025. DOI: 10.1145/3731599.3767546.

[24] P. Hopf, E. Ochoa Lopez, Y. Stade, *et al.*, “Integrating Quantum Software Tools with(in) MLIR,” in *Int’l Conf. on High Performance Computing in Asia Pacific Region*, Association for Computing Machinery, 2026. DOI: 10.1145/3773656.3773658.

[25] A. Zulehner, A. Paler, and R. Wille, “An efficient methodology for mapping quantum circuits to the IBM QX architectures,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019. DOI: 10.1109/TCAD.2018.2846658.

[26] R. Wille and L. Burgholzer, “MQT QMAP: Efficient quantum circuit mapping,” in *Int’l Symp. on Physical Design*, 2023. DOI: 10.1145/3569052.3578928.