

Towards Supporting QIR

Steps for Adopting the Quantum Intermediate Representation

Yannick Stade
yannick.stade@tum.de
Technical University of Munich
Germany

Lukas Burgholzer
lukas.burgholzer@tum.de
Technical University of Munich
Munich Quantum Software Company
Germany

Robert Wille
robert.wille@tum.de
Technical University of Munich
Munich Quantum Software Company
Germany

Abstract

Intermediate representations (IRs) play a crucial role in the software stack of a quantum computer to facilitate efficient optimizations for executing an application on hardware. One of those IRs is the *Quantum Intermediate Representation* (QIR), which builds on the classical LLVM compiler infrastructure. In this article, we outline different approaches to how QIR can be adopted. This exploration culminates in a demonstration of what it takes to turn an existing quantum circuit simulator into a QIR runtime and that such a transition is less daunting than it might seem at first. We further show that switching to QIR does not entail any performance deficits compared to the original simulator. On the contrary, the presented steps effortlessly allow adding support for arbitrary classical control flow to any classical simulator. We conclude with an outlook on future directions using QIR. The implemented QIR runtime is available under <https://github.com/munich-quantum-toolkit/core>.

CCS Concepts

• **Hardware** → **Quantum computation**.

Keywords

quantum computing, intermediate representation, compiler, runtime, software stack

1 Introduction

Since the famous results of Shor [29], the promise holds that quantum computers can offer an exponential speedup over classical computers for specific problems. Many applications [13, 15, 24, 31, 37] have since been developed to exploit this potential computational power [3, 5]. However, physical implementations of quantum computers still need to catch up and are far from being useful for the aforementioned applications, with only a few realizations of toy examples on real hardware devices reported, e. g., [4, 32]. Nevertheless, the development of real quantum computers has seen rapid progress over the past years. Breakthroughs in different technologies, such as superconducting qubits [12, 16], trapped ions [21], and neutral atoms [4, 20], have demonstrated significant advancements in the number of qubits and their reliability.

However, hardware is only one part of the story: Successfully realizing applications on quantum computers requires a sophisticated stack of software tools to transform a classical problem description into a quantum solution, to optimize the resulting (hybrid) quantum-classical program, and, finally, to execute it [6, 8].

Since real quantum computers are not capable of executing arbitrary instructions directly, a compiler with various transformation and optimization passes is needed [19, 27, 28, 30, 33, 36]. These passes are essential to minimize the overhead introduced by the compilation process and to maintain a high fidelity of the resulting quantum program. Many of the problems involved in this process are computationally hard to solve, making the development of such a compiler or a general compiler infrastructure challenging.

Intermediate representations are a key component of any compiler, enabling the seamless combination of various passes and transformations. Huge parts of the quantum computing ecosystem have come to rely on the OpenQASM format, proposed initially as version 2 [10] and refined in version 3 [11], by IBM. The format started as a low-level quantum assembly language and has since evolved into a more general-purpose quantum programming language, adding classical elements.

In contrast, the *Quantum Intermediate Representation* (QIR), initially proposed by Microsoft [25], takes a different approach. It aims to reuse as much as possible of the existing classical compiler infrastructure, namely the LLVM compiler framework [17], and augment it with quantum instructions. Despite its promises, the adoption of QIR in the quantum computing ecosystem remains low, especially outside the realm of industrial players like Microsoft, Nvidia, or Xanadu that have the resources and the expertise to develop and maintain such a compiler infrastructure.

In this article, we aim to shine a light on the challenges and opportunities that come with adopting QIR in the quantum computing software ecosystem. This should help readers to develop a better understanding of the various options to support QIR in their software tools. In particular, we detail the steps that are necessary to turn an existing quantum circuit simulator into a QIR runtime. In this regard, we show how to implement the LLVM IR interface defined by QIR for a simulator that is written in C++. Meanwhile, we elaborate on the different resource management strategies supported by QIR and how to deal with them in a classical runtime for quantum computing. The complete code is publicly available in open-source as part of the *Munich Quantum Toolkit* (MQT, [36]) under <https://github.com/munich-quantum-toolkit/core>.

Overall, the described steps show that it does not take much to turn an existing simulator into a QIR runtime. The evaluation compares the new approach with the current method of parsing QASM files at runtime. Benchmark results indicate that both approaches yield similar performance, with no clear advantage in terms of runtime. However, when using QIR, one can take advantage of the entire compiler infrastructure that exists around LLVM IR. As

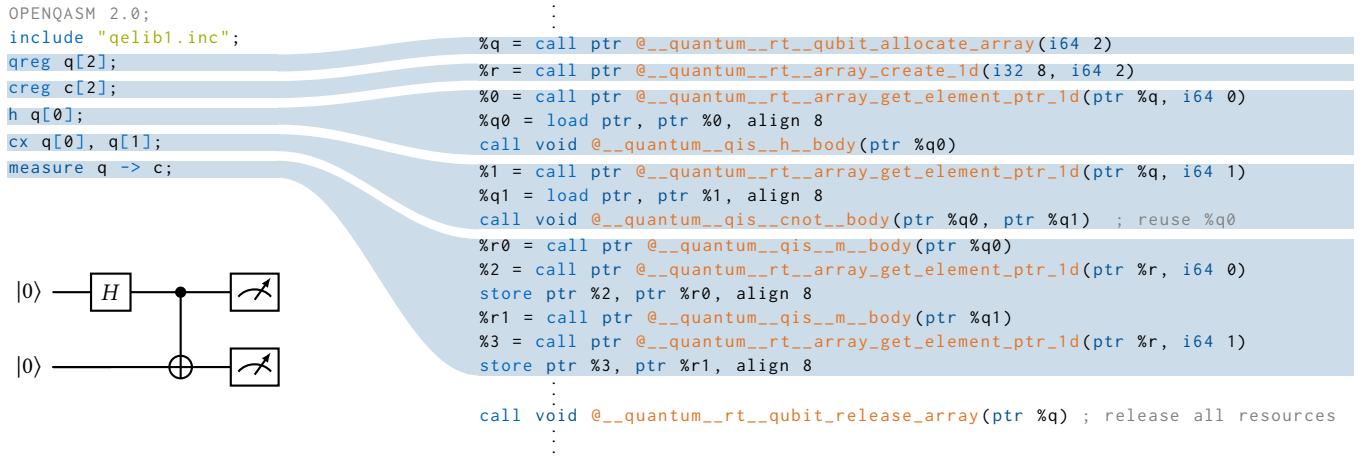


Figure 1: The quantum “Hello World” program, i. e., a circuit to create a Bell state (bottom left), expressed in OpenQASM 2.0 (top left) and QIR with dynamically allocated qubits (right). The corresponding lines of code in each fragment are linked. For more details, see also Ex. 1 and 2.

a result, the execution of hybrid programs, where classical and quantum operations are interleaved arbitrarily, comes for free.

Throughout the article, we assume the reader to be familiar with the basics of quantum computing and quantum circuits. We refer the reader to the literature [22] for an in-depth introduction.

2 An Abridged History of Quantum Intermediate Representations

First, this section briefly reviews the history of quantum intermediate representations and how quantum computations are typically represented. While certainly not exhaustive, these recollections provide a starting point for understanding the different approaches.

2.1 OpenQASM 2: Quantum Computation \approx Quantum Circuit

The *Open Quantum Assembly* (OpenQASM) language was initially proposed in version 2 by IBM in 2017 [10] to describe quantum experiments to be executed on their publicly available quantum computers. It is a low-level language that describes quantum computations in a straightforward manner, i. e., as an enumeration of quantum instructions. Additionally, it supports measurements, resets, (limited) feedback, and gate subroutines. Since its inception, OpenQASM has been widely adopted throughout the quantum computing community and is heavily used as a text-based exchange format for quantum circuits.

Example 1. The “Hello World” of quantum computing, the creation of a Bell state, is expressed in OpenQASM 2.0 on the top left in Fig. 1. After loading the quantum (standard) library, the code declares a quantum register with two qubits and a classical register with two bits. Then, a Hadamard-gate is applied to the first qubit, followed by a CNOT-gate controlled by the first qubit and targeting the second qubit. Finally, both qubits are measured.

2.2 OpenQASM 3: The Need for Hybrid Quantum Programs

While OpenQASM 2 provided a great starting point, it has become apparent over time that some degree of classical logic and control flow, including conditionals and loops that might depend on the results of quantum measurements, is desirable for writing more complex quantum programs. For near-term applications, this allows to describe variational quantum algorithms, where the quantum circuit is part of a larger classical optimization loop. For long-term applications, classical feedback is essential for quantum error correction [9, 23]. As a consequence, OpenQASM has been extended to version 3 [11], which integrates classical logic and control flow.

Fundamentally, OpenQASM started as a quantum assembly language and step-by-step evolved into a more general-purpose quantum programming language by adding classical elements on top. It builds on its strong adoption in the quantum computing community and the corresponding compiler infrastructure previously built around OpenQASM 2. At the same time, the extension to OpenQASM 3 requires implementing traditional compiler optimizations on top of the IR, such as loop unrolling, constant propagation, or constant folding to generate efficient executable code for the target hardware. Given how ubiquitous OpenQASM is in the quantum computing community, this extension is a natural and sensible step to enable more complex programs. However, it also requires the reimplementing of concepts that are already established and used for decades in classical compilers. This discrepancy has led to the development of QIR, which is discussed next.

2.3 QIR: Adding Quantum to the Classical

In contrast to OpenQASM, the *Quantum Intermediate Representation* (QIR, [25]) adopts a different strategy: It builds upon an established classical compilation infrastructure, augmenting it with quantum instructions. This approach allows QIR to utilize the existing classical compiler infrastructure, thereby inheriting its optimizations and transformations for classical code without additional

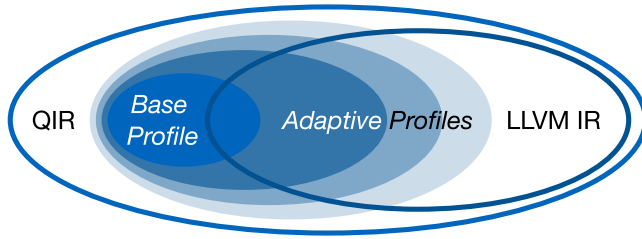


Figure 2: A Venn diagram showing the different profiles of QIR. The base profile is the most restrictive one and only uses very few instructions of the LLVM IR. QIR in its whole extent is a proper superset of LLVM IR.

effort. Specifically, QIR extends the classical IR used in the compiler framework *LLVM* [17]. It uses this IR to implement various compiler optimizations on this IR before, eventually, compiling it to target-specific assembly for execution. As such, LLVM IR can express arbitrary classical program logic, including functions, conditionals, and loops. However, it cannot express quantum computations by itself. To this end, QIR, as initially proposed by Microsoft, defines a set of additional functions that can be used to express quantum computations with the LLVM IR.

Example 2. *The relevant lines of a QIR program that represent the same circuit as in Ex. 1 are shown on the right in Fig. 1.¹ Initially, an array %q of two qubits and another array %r to fit two measurement results are allocated. Note here, that the result type is an opaque pointer according to the QIR specification. Hence, we specify the element size to be 8 bytes to fit a pointer to the results.*

For the application of the Hadamard-gate, the first qubit is extracted from the qubit array. Since, %1 is the pointer to the first element in the array and not the qubit reference itself, we dereference this variable and store the qubit reference in %q. Hereafter, the Hadamard-gate is applied to the qubit. In the next block of code, we also extract the second qubit %q1 and perform a CNOT-gate on both. Afterward, the state of both qubits is measured and stored in the initially allocated array %r. Finally, after the results got processed by classical code or recorded in the output of the program, all resources are released—something that cannot be done explicitly in OpenQASM.

In order to speed up the adoption of QIR, multiple restrictions to QIR, so-called *profiles* have been defined that limit the expressiveness of QIR.² In its most restrictive form, the *base profile* only allows a sequence of quantum instructions that ends with the measurement of all qubits, which effectively makes it very similar to OpenQASM 2. The more permissive *adaptive profiles* allow the successive transition to fully support all features contained in LLVM IR. In its full generality, QIR is a proper superset of LLVM IR that builds on top of all the existing tooling and optimizations available for LLVM IR. For an illustration of the relation between the different profiles, see Fig. 2.

¹Note, that we chose to use modern LLVM syntax with opaque pointers here, as opposed to the legacy syntax used in the initial QIR specification.

²https://github.com/qir-alliance/qir-spec/tree/main/specification/under_development/profiles (visited on 03/03/2025)

3 Adopting QIR: Quo Vadis?

The proposal of QIR is both logical and appealing: It makes sense to avoid reinventing the wheel and to build on decades of classical compilation expertise, especially, since quantum computations will inherently involve classical input and produce classical output. However, the adoption of QIR in the quantum computing software stack is a challenging task: What does it actually mean to adopt QIR in the stack? How do we transform these quantum-classical programs so that they can be executed? What about the actual execution? In the following, we discuss two fundamentally different directions of adopting QIR, where the first of both is subdivided into parsing and transforming QIR programs.

3.1 Parsing QIR Programs

In order for a quantum program to be executed, it must be transformed so that it complies with all the restrictions imposed by the hardware. Tools responsible for this transformation need to be able to accept the input quantum program and transform it accordingly [19, 27, 28, 30, 36]. When using QIR, tools may either receive the program as a text file containing QIR code or as an in-memory representation of the program, such as an *Abstract Syntax Tree* (AST). The former approach requires a parser that can turn the text-based representation into some in-memory representation. This in-memory representation can either be the QIR AST, in which case LLVM can be used, or another custom and tool-specific IR. For a custom IR, the parser's (and, in fact, also the exporter's) complexity depends on the profile of QIR to be supported.

Example 3. *For the straight-line quantum programs, it suffices to iterate over the lines to construct an in-memory representation of the resulting quantum circuit. For example, when parsing the QIR program from Ex. 2, the parser would need to track the assignment of variables (i. e., %q, %0, %1, ...) to their values to infer the respective qubit that is passed to a quantum instruction. The instructions themselves can be matched with a simple pattern and corresponding actions can be taken to gradually build up the quantum circuit.*

On the one hand, the advantage of a custom parser is that one can avoid the dependency on LLVM, which should be a carefully considered decision because of the substantial size and complexity of the module. LLVM as a dependency can significantly increase the build and maintenance overhead of a project. On the other hand, when not using LLVM, one has to reimplement all the optimizations and transformations that are already provided for LLVM IR “for free”—similar to the ongoing development of tools for OpenQASM 3. Additionally, when using a custom IR, one is limited to the capabilities of that existing IR. If that IR is not expressive enough to capture all the details of a quantum program, the QIR program cannot be fully translated and, hence, the tool cannot be used on these programs. As a consequence, the IR would have to add support for representing and handling all the concepts that QIR introduces, which might be a further significant effort in addition to reimplementing the optimizations and transformations.

3.2 Transforming QIR Programs

As a continuation of the first direction, after loading the quantum program, multiple passes in the software stack optimize and transform the program to meet all constraints imposed by the hardware. For the remainder of this section, we assume that the program is represented as a QIR AST in memory.

The support for QIR can, again, be realized in two different ways: Either the QIR is transformed directly, or the program is transpiled into another custom IR, transformed, and transpiled back into QIR. While the latter approach enables quick adoption of QIR, it carries the same deficits as parsing the text-based QIR file into a custom IR. For tools to support QIR directly, the AST of a QIR program must be handled and transformed. While this might be simple enough for the base profile, it becomes more complex for the adaptive profiles, which could have arbitrarily complex classical code in-between quantum instructions. This approach, however, has the advantage that the tool can directly take advantage of existing optimizations and transformations that are already implemented for QIR—the core motivation of an IR in a compiler.

Example 4. One use case of classical *FOR*-loops is the application of a gate to multiple qubits. The following QIR snippet shows a simple *FOR*-loop that performs a Hadamard gate on the qubits with hardware addresses 0, ..., 9.

```
%i = alloca i32, align 4
store i32 0, ptr %i, align 4          ; int i = 0
br label %for.header

for.header:
%1 = load i32, ptr %i, align 4
%cond = icmp slt i32 %1, 10           ; i < 10
br i1 %cond, label %body, label %exit

body:
%2 = load i32, ptr %i, align 4
call void @__quantum__qis__h__body(
    ptr inttoptr (i64 %2 to ptr)) ; h(qi)
%3 = load i32, ptr %i, align 4
%4 = add nsw i32 %3, 1                ; i++
store i32 %4, ptr %i, align 4
br label %for.header

exit: ...
```

Since QIR builds on the LLVM infrastructure, it is straightforward to unroll any loops with statically known bounds in the QIR program. Hence, a quantum optimization pass does not have to handle the *FOR*-loop, but only sees the ten individual Hadamard gates that are applied to the qubits.

While the previous sections discussed handling and transforming QIR programs themselves, the question remains how these transformed programs are eventually executed.

3.3 Executing QIR Programs

In contrast to the type of support for QIR discussed so far, the execution of QIR programs is fundamentally different. Another direction of adopting QIR enables the execution of QIR programs and can either be seen as a complement to the previously discussed options or as an orthogonal approach.

A file that contains LLVM IR bytecode can be executed directly with the `lli` tool provided by the LLVM project. This tool reads the bytecode and interprets it on the fly. Furthermore, the compiler `clang`, also part of the LLVM project, can compile the bytecode to machine code and emit an executable.

Despite `clang` and `lli` being already able to parse QIR programs, when compiling or interpreting those files individually without any additional definitions, the tools will raise an error for every quantum instruction they encounter. The raised error message will point out that a definition for the respective function cannot be found. This is logical, as the quantum instructions are not part of the LLVM IR standard and, hence, `clang` and `lli` do not know how to handle them. To overcome this issue, and be able to compile QIR programs to executables, we have to provide the missing definitions for the quantum instructions.

Example 5. One example that takes this approach is the Catalyst Quantum Runtime implemented by Xanadu that incorporates their classical quantum circuit simulator Lightning [1], which is part of PennyLane [2]. Every function, such as `@__quantum__qis__h__body`, is implemented so that it modifies the internal state of the simulator to reflect the application of the respective gate. Note that these functions need not be implemented in LLVM IR, as this would be quite cumbersome. Instead, the definition of the QIR functions can be provided as an implementation (written in C/C++, Rust or other kinds of languages) of a C interface, which is then compiled to LLVM just as any other regular classical program code.

This approach is orthogonal to the previous two options as it only concerns the implementation of quantum instructions, while the actual program structure is handled by the runtime. To this end, this approach is perfectly suited for integrating classical simulation techniques with QIR, especially ones developed in a language that compiles to LLVM, such as C/C++ or Rust. The resulting binaries can be maximally optimized by the LLVM compiler infrastructure, which is a significant advantage.

4 Realization of a Classical QIR Runtime

In the following, we demonstrate how an existing quantum circuit simulator can be turned into a QIR runtime—illustrating the steps necessary to support QIR. For this demonstration, we use the quantum circuit simulator based on decision diagrams [34, 35] that is bundled with MQT Core [7] as part of the *Munich Quantum Toolkit* (MQT [36]). The resulting implementation of the QIR runtime is also publicly available as open-source software as part of MQT Core under <https://github.com/munich-quantum-toolkit/core>.

4.1 Implementing the Quantum Instructions

The specification of QIR defines a couple of functions to deal with data structures used by the QIR runtime. For example, there are dedicated functions to create and manipulate arrays of qubits and measurement results (see also Sec. 4.3). Additionally, the runtime provides special handling for strings and big integers. However, the core part of the QIR runtime is the quantum instruction set (QIS). The name of those functions follows the following pattern: `@__quantum__qis__<gate>__body`, where `<gate>` is the name of the respective quantum gate. For example, the function that implements the rotation around the x-axis is declared as `declare void @__quantum__qis__rx__body(ptr, double)`, where the first parameter is the qubit reference and the second one is the rotation angle.

Any implementation of the specification must provide definitions for all declarations. The standard way to do that is to implement the

C-API corresponding to the LLVM function signatures defined in the specification.³ The following code fragment shows the definition of the QIR function corresponding to the RX gate.

```
extern "C" {
void __quantum__qis__rx__body(Qubit* qubit, double phi) {
    auto& backend = QIR_DD_Backend::getInstance();
    backend.apply(qc::RX, phi, qubit);
}
} // extern "C"
```

The backend is implemented as a singleton class that is accessed via the static method `QIR_DD_Backend::getInstance`. The first time this method is called, the singleton instance is created and initialized. Afterward, we can forward the application of any quantum gate to the backend and accordingly modify the quantum state represented in the runtime. Below, you see an excerpt of the implementation of the singleton class `QIR_DD_Backend` that serves as a mediator between the QIR runtime and the simulator.

```
class QIR_DD_Backend {
private:
    explicit QIR_DD_Backend(); // private constructor
public:
    // Return the unique instance of the backend.
    static QIR_DD_Backend& getInstance() {
        static QIR_DD_Backend instance;
        return instance;
    }
    // ... other attributes and methods
};
```

4.2 Handling Dynamic Qubit Allocation

Qubits in QIR are represented as opaque pointers. Thus, a crucial part of the runtime is to translate between those opaque pointers and the qubit identifiers used in the simulator. The simulator under consideration uses consecutively numbered unsigned integers starting from 0 to identify qubits in the underlying state. Hence, the opaque pointer values of qubits in the QIR program must be translated to the respective qubit identifiers in the simulator. QIR offers two qubit management strategies—either via dynamically allocating qubits or via static addressing. In the following, the management of dynamically allocated qubits is described. You can find more details on how to handle static addressing in Sec. 6.1.

First, a definition for the QIR function to allocate a qubit needs to be provided. Note the surrounding `extern "C"` declaration of the function. This is required because we must implement the C interface to finally be able to properly link the runtime with the QIR code. The `extern "C"` declaration ensures that this part of the code is compiled as C code even though it is contained in a `.cpp` file.

```
extern "C" {
Qubit* __quantum__rt__qubit_allocate() {
    auto& backend = mqt::QIR_DD_Backend::getInstance();
    return backend.qAlloc();
}
} // extern "C"
```

The above function simply delegates the allocation of a new qubit to the runtime, which internally maintains a `qRegister` hash map of unique addresses and qubit indices as follows. The dynamic

qubit addresses are given out in increasing order starting from `MIN_DYN_QUBIT_ADDRESS`.

```
class QIR_DD_Backend {
private:
    std::unordered_map<const Qubit*, size_t> qRegister;
    uintptr_t currentMaxQubitAddress =
        MIN_DYN_QUBIT_ADDRESS;
    size_t currentMaxQubitId = 0;
    ...
public:
    auto qAlloc() -> Qubit* {
        auto* qubit =
            reinterpret_cast<Qubit*>(currentMaxQubitAddress++);
        qRegister.emplace(qubit, currentMaxQubitId++);
        return qubit;
    }
    // ... other attributes and methods
};
```

The actual translation of an instruction's qubit addresses to the simulator's identifiers is delegated to the following function within the runtime.

```
template <size_t SIZE>
auto QIR_DD_Backend::translateAddresses(
    const std::array<Qubit*, SIZE>& qubits)
    -> std::array<size_t, SIZE> {
    // transform opaque qubit pointers to qubit ids
    std::array<size_t, SIZE> qubitIds{};
    for(size_t i = 0; i < SIZE; ++i) {
        qubitIds[i] = qRegister.at(qubits[i]);
    }
    return qubitIds;
}
```

In particular, qubit addresses are never dereferenced in this setting, but only used as keys in the hash map. For an alternative approach, where qubit addresses point to valid memory locations and the qubit identifiers are stored in the memory, see Sec. 7.

No matter how the address translation is implemented, the runtime must cope with qubits that might be allocated and deallocated in the middle of the program execution. This dynamic allocation might not be necessary for QIR files that follow a certain profile that explicitly disallows allocation in the middle of the program. E. g., the base profile prohibits dynamic qubit allocation entirely and only allows static addressing (see Sec. 6.1). However, the presented quantum runtime supports static and dynamic qubit addressing. As a consequence, the runtime must be able to dynamically adjust the size of the quantum state in the simulator to accommodate the qubits that are allocated during the program execution.

For the chosen quantum circuit simulator that is based on decision diagrams, this enlargement of the quantum state is straightforward. The following code fragment sketches this process for the implemented runtime: The input parameter is the maximum qubit identifier that is currently needed, e. g., in the application of a quantum gate. Based on that, the backend checks whether this identifier is already contained in the current state, and if not, accordingly extends the state with additional qubits.

```
auto QIR_DD_Backend::enlargeState(uint64_t maxQubit)
    -> void {
    if (maxQubit >= numQubitsInQState) {
        const auto d = maxQubit - numQubitsInQState + 1;
        numQubitsInQState += d;
        dd->resize(numQubitsInQState);
        [initialize new qubits in |0>-state]
    }
}
```

³The main reason for choosing C for the interface is that C has a well-defined and stable application binary interface (ABI), which allows the resulting object file to be properly linked with the LLVM IR code. This, in no way, precludes an implementation in higher-level languages such as C++ or Rust, as also demonstrated by this article.

4.3 Defining Additional Data-Structures

Besides the definition of qubits as pointers to an opaque type %Qubit, the QIR specification also defines a couple of data structures that are used by the QIR runtime. This subsection illustrates their realization in the implemented runtime using the example of the %Array type. To not leave the array type incomplete, the backend provides the following definition with attributes that are necessary to implement all QIR runtime functions related to arrays.

```
using Array = struct ArrayImpl {
    int32_t refcount;           // reference count
    int32_t aliasCount;        // count of aliases
    std::vector<int8_t> data;    // vector of bytes
    int64_t elementSize;       // number of bytes per elem.
};
```

The QIR's specification enforces the principle of reference counting on classical resources like arrays. For arrays specifically, alias counting is employed to keep track of read-only references to the array—the handling of those is omitted here for brevity.

The life-cycle of a classical resource in QIR, and in particular of an array includes its creation, access, and release. First, the code snippet below provides the implementation of a one-dimensional array creation for a given element size s in number of bytes and the total number of elements n .

```
Array* __quantum__rt__array_create_1d(
    int32_t s, int64_t n) { // s: element's size
    return new Array{1, 0, std::vector<int8_t>(s * n), s};
}
```

During program execution data should be written and read from the array. To this end, the function below returns a pointer to the i -th element of the array. Using this pointer, the data in the corresponding element can be accessed or modified.

```
int8_t* __quantum__rt__array_get_element_ptr_1d(
    Array* arr, int64_t i) {
    [perform bounds checking]
    return &arr->data[arr->elementSize * i];
}
```

At the end, when the array is no longer needed, the associated memory should be freed to make space for other resources. The deallocation of classical resources is part of the reference counting mechanism in QIR. Hence, the function to update the reference count, frees the associated memory when the reference count decreases to zero.

```
void __quantum__rt__array_update_reference_count(
    Array* arr, int32_t k) {
    if (arr == nullptr) return;
    arr->refcount += k; // k can also be negative (!)
    if (arr->refcount == 0) delete arr;
}
```

5 Demonstrating the Runtime's Functionality

The explanations and code snippets above have, hopefully, given a rough overview of what it takes to realize a QIR runtime based on an existing quantum circuit simulator. This section complements the description of the implementation by providing the results of running multiple benchmarks with the implemented runtime. This achievement demonstrates our ability to streamline the process of connecting quantum and classical systems without reinventing the wheel.

For the evaluation, we considered five different families of quantum circuits from the MQT Bench quantum circuit benchmark suite [26]. Those correspond to the first five families listed in Table 1 and can be realized using the QIR base profile. Additionally, we created three other sets of benchmark circuits from the MQT Core library [36]. Those last three circuit families in Table 1 all make use of QIR features defined as part of the adaptive profile. More precisely, they employ intermediate measurements, reuse already measured qubits, and, in case of the qft-iter and qpeexact-iter circuits, furthermore contain quantum operations that are conditioned on previous measurement results.

At the time of writing, neither MQT Bench nor MQT Core directly produce QIR code. Thus, all circuits were initially created as OpenQASM 3 programs and subsequently translated to equivalent QIR programs. During this process, we created two versions of the QIR files: one that uses dynamically allocated qubits as described in the implementation section, i. e., Sec. 4.2, and one that uses statically addressed qubits (see Sec. 6.1).

Before the QIR files can be executed, they must be compiled to an executable. For this, every QIR file is linked to the implemented QIR runtime separately and compiled to an executable. Those executables can then be invoked, and their execution time can be measured. For a comparison between the execution of the QIR files and the existing MQT simulator, we employed the default simulation capabilities of the MQT to parse and simulate OpenQASM 3 files.

During the evaluation we collected the following metrics:

(1) Compilation Times

- (a) of the application to parse and simulate OpenQASM 3 files
- (b) of the library containing the QIR runtime (without a specific QIR file)
- (c) of every QIR file using the already compiled library from Item 1b

(2) File Sizes

- (a) of the application from Item 1a
- (b) of the OpenQASM files parsed by the application
- (c) of the executables compiled from the QIR files




(3) Execution Times









- (a) of the application to parse and simulate OpenQASM 3 files
- (b) of the QIR executables using the QIR runtime




Note that the application and the library only need to be compiled once, which is why we record the times of Items 1a and 1b only once. The results of all experiments, which were executed on a machine with an Apple M3 chip, 16 GB of RAM, and clang version 16.0.0, are summarized in Table 1.

At the top of the table, we provide the file sizes and compilation times of the QASM simulator application together with the compilation time for the QIR runtime library. Note that the compilation of the simulator executable takes more time than the compilation of the QIR runtime library together with each QIR file. However, the application only needs to be compiled once and can then be used to parse and simulate multiple OpenQASM files without re-compilation, while a separate executable has to be compiled per QIR file.

Regarding the results of the individual circuits, we first take a look at the resulting file sizes. For the simulation of QASM files,

Table 1: Evaluation ResultsQASM simulator executable file size : 744 kB and compilation time : 10.496 sQIR runtime library compilation time : 3.028 s

Benchmark	#Qubits	Dynamically Parsing QASM		Executing Compiled QIR					
		 [kB]	 [s]	Static Addressing			Dynamic Addressing		
				 [s]	 [kB]	 [s]	 [s]	 [kB]	 [s]
ghz	256	13	0.008	3.828	324	0.013	3.846	346	0.019
	512	26	0.028	3.864	349	0.045	3.875	371	0.089
	1024	53	0.141	3.911	398	0.201	3.863	420	0.273
	2048	111	0.736	3.973	480	0.980	3.937	535	2.340
	4096	228	4.412	4.399	662	5.675	4.013	931	6.807
qft	30	12	0.003	3.822	317	0.002	3.849	339	0.002
	35	17	0.004	3.873	317	0.004	3.843	339	0.004
	40	22	0.005	3.838	317	0.004	3.848	340	0.005
	45	27	0.006	3.899	318	0.005	3.866	340	0.005
qpeexact	30	13	0.003	3.825	317	0.003	3.858	340	0.002
	35	18	0.007	3.934	318	0.003	3.838	340	0.004
	40	23	0.006	3.823	318	0.004	3.867	340	0.004
	45	29	0.208	3.919	334	0.005	4.108	340	0.006
random	14	11	0.193	4.175	339	0.218	3.941	344	0.216
	15	13	0.713	3.948	339	0.736	3.983	344	0.761
	16	14	2.088	3.968	339	2.110	3.917	361	2.151
	17	15	18.315	3.905	340	17.703	4.302	362	17.430
	18	18	95.679	3.913	340	89.789	4.011	362	90.568
wstate	256	33	0.042	3.863	341	0.049	3.850	363	0.057
	512	67	0.165	3.892	382	0.237	4.075	404	0.259
	1024	136	0.802	4.274	448	1.004	3.938	486	1.023
	2048	282	4.544	4.283	613	5.397	4.033	668	5.709
	4096	574	30.504	5.599	920	34.191	4.316	1295	36.063
bv-iter	256	15	0.002	3.830	346	0.002	3.962	347	0.002
	512	29	0.003	3.879	371	0.002	3.976	371	0.002
	1024	58	0.006	3.952	420	0.003	3.910	437	0.002
	2048	117	0.011	4.377	535	0.004	3.863	552	0.004
	4096	236	0.020	6.923	750	0.007	4.117	866	0.006
qft-iter	30	22	0.002	3.839	338	0.002	3.881	339	0.001
	35	30	0.002	4.265	338	0.002	3.827	339	0.001
	40	40	0.003	4.192	338	0.002	3.871	355	0.002
	45	51	0.003	3.828	355	0.001	3.865	356	0.002
qpeexact-iter	30	24	0.002	3.822	340	0.002	3.859	341	0.002
	35	32	0.003	3.822	357	0.002	3.839	358	0.001
	40	42	0.003	3.840	357	0.002	3.844	358	0.001
	45	53	0.004	3.883	357	0.002	3.860	358	0.002

: Compilation times in seconds (see Item 1): Files Sizes in kilobytes (see Item 2): Execution Times seconds (see Item 3)

the application together with the QASM file is required to run the respective circuit. Hence, we compare the sum of the file sizes of the application and the QASM file with the size of the compiled QIR executables. It stands out that for smaller circuits, the compiled QIR files are significantly smaller. However, for larger circuits, the file size of the QIR executables increases faster than the file size of the QASM files. Nevertheless, for all circuits in the evaluation, the size of the QIR executables remains below the size of the application and the QASM file together.

With a look at the execution times, all three different settings lead to similar results.⁴ This is hardly surprising, as all of the approaches are carrying out identical quantum computations, and the majority of the execution time is spent applying the respective operations to the current state. Hence, the overhead from parsing the file within

⁴One may notice that the qubit ranges and the execution time scales between the individual circuit families differ wildly. This is a testament to the properties of decision diagrams, which are used as the data structure for the simulation. Similar to tensor networks, decision diagrams allow representing certain problem classes much more efficiently and compactly than an array-based statevector representation by exploiting structure in the underlying representation. At the same time, they incur a significant overhead for problem classes that do not offer any benefit for the decision diagram structure as, e.g., seen in the results for the random circuit family [14].

the application to simulate QASM files is negligible. The execution times of the QIR files using dynamic and static qubit addresses also do not differ significantly.

In summary, those results manifest that with manageable effort, an existing quantum circuit simulator can be turned into a QIR runtime while maintaining the same performance. This opportunity opens up a way for fast adoption of the QIR standard with tools that were previously developed to work with QASM files or similar intermediate representations. On top of that, one unlocks the ability to execute arbitrary hybrid classical-quantum programs such as quantum error correction protocols with classical feedback loops or dynamic quantum circuits as demonstrated with the iterative versions of the Bernstein-Vazirani algorithm, the Quantum Fourier Transform, and the Quantum Phase Estimation algorithm in Table 1.

6 Challenges in Compiling QIR Programs

The integration of a quantum IR like QIR into a compiler infrastructure is challenging and comes with a number of open research questions. Here, we detail two of those challenges.

6.1 Static and Dynamic Qubit Addresses

QIR provides the opaque pointer type `%qubit*` for addressing individual qubits. In Ex. 2, the qubits were addressed dynamically by first allocating the respective qubits and retrieving dynamic qubit addresses that later can be used as arguments to quantum instructions. In contrast to dynamic addresses, there is also the possibility to address qubits statically.

Example 6. *When addressing the qubits statically instead of dynamically, the circuit from Fig. 1 can be rewritten in QIR in the following snippet. Especially, the lines for allocating the qubits disappear.*

```

;
call void @__quantum__qis__h__body(ptr null)
call void @__quantum__qis__cnot__body(ptr null,
    ptr inttoptr (i64 1 to ptr))
call void @__quantum__qis__mz__body(ptr null, ptr null)
call void @__quantum__qis__mz__body(ptr inttoptr
    (i64 1 to ptr), ptr inttoptr (i64 1 to ptr))
;

```

This type of addressing qubits can be useful when compiling a quantum program to an executable very close to hardware. In the end, the hardware only has a fixed number of qubits, and the compiler must ensure that the program does not exceed this number. So while a quantum program can be written with dynamic qubit addresses, the compiler must at some point assign the program's qubits to the hardware's qubits—a process very similar to register allocation in classical compilers. At the moment, this distinction between static and dynamic qubit addresses is not yet fully established in the QIR community, but it is an important aspect to consider when integrating QIR into a compiler infrastructure.

In the context of implementing a QIR runtime for a quantum circuit simulator—as outlined above—dynamic qubit addresses are the preferred way to address qubits. Most simulators support a variable number of qubits; the maximum feasible number might even depend on the input circuit itself, e. g., on the degree of entanglement that is created during its execution. By allocating the qubits on demand, one can efficiently scale the simulated state to the required size and save memory if no large state is needed.

To support static qubit addresses, the runtime would have to infer the number of qubits required for the simulation somehow. According to the examples in the QIR specification, every function in a QIR file has an attribute denoting the number of required qubits by this function. Those attributes can be accessed by compilation passes in the process of compiling the source code into an executable. However, in our setting, where we just provide the definitions of the QIR runtime functions as a C/C++ implementation, we cannot access those attributes from the QIR files. One can still allocate qubits on the fly when encountering a new qubit address that is not yet part of the simulated quantum state (similar to how this is handled in the runtime implementation evaluated in Sec. 5).

6.2 Hybrid Classical-Quantum Computing

As QIR is a superset of LLVM IR, arbitrarily complex classical computations can be expressed in LLVM IR. Those heavy classical computations are preferably executed on dedicated classical hardware rather than on the classical co-processor of a quantum computer—if that is even feasible: The classical co-processor of quantum computers must be very fast, and special-purpose hardware like FPGAs or ASICs are employed in many such cases. These are incapable of executing arbitrary classical code, which is also not their purpose. Consequently, the question of how to decide which part of the code should be executed on the classical hardware and which part on the quantum hardware naturally arises for a hybrid classical-quantum programs that contain quantum as well as classical instructions.

Especially in the realm of error correction, where conditional gate applications based on intermediate measurements must be performed on the quantum computer to ensure low latency, the distinction is more complicated than just offloading only the quantum instructions. At the same time, it must be ensured that the classical code offloaded to the quantum hardware can be executed in the required time frame to uphold the coherence of the qubits. Hence, as long as quantum computers cannot achieve arbitrary coherence of the qubits, there will always be programs that describe an infeasible execution and must be rejected.

7 Conclusions

QIR is one way to bridge the gap between purely classical and purely quantum computations. As an extension of LLVM IR, it builds on decades of compiler infrastructure and optimizations. As outlined in this article, the adoption of QIR can be achieved with less effort than it might seem at first glance. However, some open research questions remain to be addressed in the future, especially in the context of hybrid classical-quantum programs. One framework for exploring solutions to these questions is the *Multi-Level Intermediate Representation* (MLIR, [18]), which is a natural choice for the next step in the evolution of QIR.

Acknowledgements

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 the Munich Quantum Valley (MQV), which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus.

References

- [1] Ali Asadi et al. Hybrid quantum programming with PennyLane Lightning on HPC platforms. (2024). arXiv: 2403.02512.
- [2] Ville Bergholm et al. PennyLane: automatic differentiation of hybrid quantum-classical computations. (2022). arXiv: 1811.04968.
- [3] Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM Jour. of Comp.* DOI: 10.1137/S0097539796300921.
- [4] Dolev Bluvstein et al. 2023. Logical quantum processor based on reconfigurable atom arrays. *Nature*. DOI: 10.1038/s41586-023-06927-3.
- [5] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tappa. 1999. Tight Bounds on Quantum Searching. In *Quantum Computing*. (1st ed.). Wiley. DOI: 10.1002/3527603093.ch10.
- [6] Lukas Burgholzer, Jorge Echavarria, Martin Schulz, Laura Schulz, and Robert Wille. 2024. Building Efficient Software Stacks for Quantum Computers: Experiences from the Munich Quantum Software Stack. In *Int'l Conf. on Quantum Computing and Engineering*. IEEE.
- [7] Lukas Burgholzer, Yannick Stade, Tom Peham, and Robert Wille. 2025. MQT Core: The backbone of the Munich Quantum Toolkit (MQT). *Journal of Open Source Software*. DOI: 10.21105/joss.07478.
- [8] Lukas Burgholzer et al. 2025. The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC. (2025). arXiv: 2509.02674 [quant-ph].
- [9] Earl T. Campbell, Barbara M. Terhal, and Christophe Vuillot. 2017. Roads towards fault-tolerant universal quantum computation. *Nature*. DOI: 10.1038/nature23460.
- [10] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. (2017). arXiv: 1707.03429.
- [11] Andrew W. Cross et al. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*. DOI: 10.1145/3505636.
- [12] Google Quantum AI et al. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature*. DOI: 10.1038/s41586-022-05434-1.
- [13] Lov K. Grover. 1997. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 2. DOI: 10.1103/PhysRevLett.79.325.
- [14] Thomas Grurl, Jurgen Fus, Stefan Hillmich, Lukas Burgholzer, and Robert Wille. 2020. Arrays vs. Decision Diagrams: A Case Study on Quantum Circuit Simulators. In *Int'l Symp. on Multi-Valued Logic*. 2020 IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL). IEEE. DOI: 10.1109/ISMVL49045.2020.000-9.
- [15] Stuart Harwood, Claudio Gambella, Dimitar Trenev, Andrea Simonetto, David Bernal Neira, and Donny Greenberg. 2021. Formulating and Solving Routing Problems on Quantum Computers. *IEEE Transactions on Quantum Engineering*. DOI: 10.1109/TQE.2021.3049230.
- [16] Youngseok Kim et al. 2023. Evidence for the utility of quantum computing before fault tolerance. *Nature*. DOI: 10.1038/s41586-023-06096-3.
- [17] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Int'l Symp. on Code Generation and Optimization*. IEEE Computer Society. DOI: 10.1109/CGO.2004.1281665.
- [18] Chris Lattner et al. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Int'l Symp. on Code Generation and Optimization*. DOI: 10.1109/CGO51591.2021.9370308.
- [19] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM. DOI: 10.1145/3297858.3304023.
- [20] Hannah J. Manetsch, Gyohei Nomura, Elie Bataille, Kon H. Leung, Xudong Lv, and Manuel Endres. A tweezer array with 6100 highly coherent atomic qubits. (2024). arXiv: 2403.12021.
- [21] S. A. Moses et al. 2023. A race-track trapped-ion quantum processor. *Phys. Rev. X*, 4. DOI: 10.1103/PhysRevX.13.041052.
- [22] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information*. (10th anniversary ed ed.). Cambridge University Press.
- [23] Tom Peham, Ludwig Schmid, Lucas Berent, Markus Müller, and Robert Wille. 2024. Automated Synthesis of Fault-Tolerant State Preparation Circuits for Quantum Error Correction Codes. (2024). arXiv: 2408.11894.
- [24] Alberto Peruzzo et al. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*. DOI: 10.1038/ncomms5213.
- [25] QIR Alliance: <https://qir-alliance.org>. 2021. *QIR Specification*. Version 0.1. QIR Alliance: <https://qir-alliance.org>.
- [26] N. Quetschlich, L. Burgholzer, and R. Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*. DOI: 10.22331/q-2023-07-20-1062.
- [27] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2025. MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing. *ACM Transactions on Quantum Computing*. DOI: 10.1145/3673241.
- [28] Daniel Schoenberger, Stefan Hillmich, Matthias Brandl, and Robert Wille. 2024. Using Boolean Satisfiability for Exact Shuttling in Trapped-Ion Quantum Computers. In *Asia and South Pacific Design Automation Conf*. DOI: 10.1109/ASP-DAC58780.2024.10473902.
- [29] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Symp. on Foundations of Computer Science*. IEEE Comput. Soc. Press. DOI: 10.1109/SFCS.1994.365700.
- [30] Yannick Stade, Ludwig Schmid, Lukas Burgholzer, and Robert Wille. 2024. An Abstract Model and Efficient Routing for Logical Entangling Gates on Zoned Neutral Atom Architectures. In *Int'l Conf. on Quantum Computing and Engineering*. arXiv: 2405.08068.
- [31] Nikitas Stamatopoulos et al. 2020. Option pricing using quantum computers. *Quantum*. DOI: 10.22331/q-2020-07-06-291.
- [32] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. 2001. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*. DOI: 10.1038/414883a.
- [33] Robert Wille and Lukas Burgholzer. 2023. MQT QMAP: Efficient Quantum Circuit Mapping. In *Proc. 2023 Int. Symp. Phys. Des.* ACM. DOI: 10.1145/3569052.3578928.
- [34] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. 2023. Decision Diagrams for Quantum Computing. In *Design Automation of Quantum Computers*. Springer International Publishing. DOI: 10.1007/978-3-031-15699-1_1.
- [35] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. 2022. Tools for Quantum Computing Based on Decision Diagrams. *ACM Transactions on Quantum Computing*. DOI: 10.1145/3491246.
- [36] Robert Wille et al. 2024. The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing. In *Int'l Conf. on Quantum Software*. arXiv: 2405.17543. A live version of this document is available at <https://mqt.readthedocs.io>.
- [37] Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. 2019. Quantum Generative Adversarial Networks for learning and loading random distributions. *npj Quantum Information*. DOI: 10.1038/s41534-019-0223-2.