Review of Intermediate Representations for Quantum Computing

F. Javier Cardama[1*†], Jorge Vázquez-Pérez[1†], César Piñeiro[1,2†], Juan C. Pichel[1,2†], Tomás F. Pena[1,2†], Andrés Gómez[3†]

[1]Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, 15782, Galicia, Spain.
[2]Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, Santiago de Compostela, 15705, Galicia, Spain.
[3]Galicia Supercomputing Center (CESGA), Santiago de Compostela, 15705, Galicia, Spain.

*Corresponding author(s). E-mail(s): javier.cardama@usc.es;
Contributing authors: jorgevazquez.perez@usc.es; cesaralfredo.pineiro@usc.es; juancarlos.pichel@usc.es; tf.pena@usc.es; andres.gomez.tato@cesga.es;
†These authors contributed equally to this work.

Abstract

Intermediate representations (IRs) are fundamental to classical and quantum computing, bridging high-level quantum programming languages and the hardware-specific instructions required for execution. This paper reviews the development of quantum IRs, focusing on their evolution and the need for abstraction layers that facilitate portability and optimization. Monolithic quantum IRs, such as QIR [1], QSSA [2], or Q-MLIR [3], their effectiveness in handling abstractions and their hybrid support between quantum-classical operations are evaluated. However, a key limitation is their inability to address qubit locality, an essential feature for distributed quantum computing (DQC).

To overcome this, InQuIR [4] was introduced as an IR specifically designed for distributed systems, providing explicit control over qubit locality and inter-node communication. While effective in managing qubit distribution, InQuIR's dependence on manual manipulation of communication protocols increases complexity for developers. NetQIR [5], an extension of QIR for DQC, emerges as a solution to achieve the abstraction of quantum communications protocols. This review emphasizes the need for further advancements in IRs for distributed quantum systems, which will play a crucial role in the scalability and usability of future quantum networks.

Keywords: intermediate representation, quantum compiling, distributed quantum computing, compilers

1 Introduction

The evolution of computing has consistently aimed to abstract hardware complexities to facilitate architecture-agnostic software development. A key milestone in classical computing was the introduction of compilers, which played a central role in translating high-level programming languages into hardware-specific instructions. Without them, every piece of software would have to be tailored to a specific hardware platform, significantly slowing the development and adoption of new technologies. As computing systems became more sophisticated, so did the need for more flexible and efficient compilation processes [6–9].

However, compilers are some of the most complex programs ever developed, often rivalling operating systems in their intricacy. Their development and maintenance require significant resources. To address this, the field of classical computing introduced the concept of Intermediate Representations (IR) or Intermediate Languages. These serve as a middle layer between the high-level languages (such as C++ or Python) and the low-level instructions specific to the hardware (such as x86 or ARM instruction sets). The IR allows for a modular compilation process, simplifying the creation of new compilers and facilitating the introduction of new languages and hardware architectures [10–12].

On the other hand, in recent decades, quantum computing has emerged as a groundbreaking field with the potential to revolutionize many areas of science and technology. The development of Shor's algorithm, which promises an exponential speedup in factoring large numbers, shows the immense computational power that quantum systems could bring [13]. This was a direct challenge to classical cryptography, which relies on the difficulty of prime factorization. Since then, the race to develop quantum computers has intensified, with the promise of solving problems currently intractable for classical computers [14, 15].

As the field of quantum computing evolves, we are seeing the emergence of new languages, libraries and frameworks designed to facilitate the programming of quantum hardware and simulators. Like classical computing, quantum computing has to follow a path of abstraction and simplification, building on decades of knowledge of classical compiler design. A key area is the proper development of quantum compilers and the integration of intermediate representations for quantum languages, easing the path towards more accessible quantum programming [16–19].

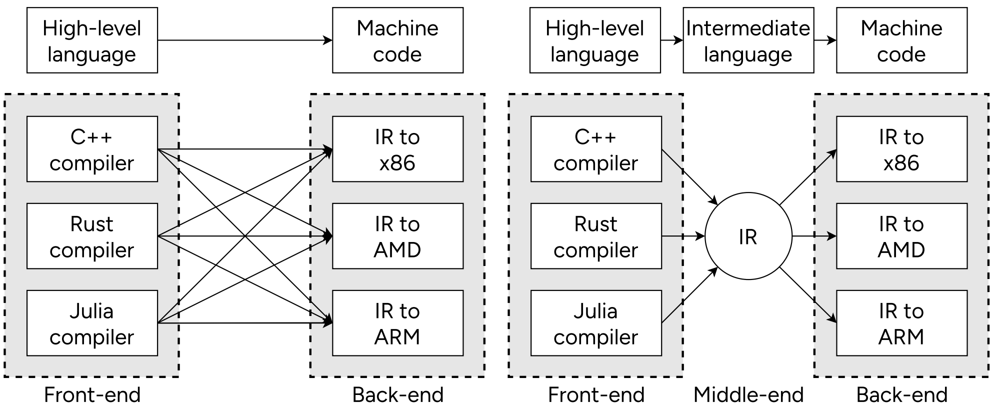
The need for quantum intermediate representations arises from what has been learned in classical computing, where abstraction has played a critical role in reducing the complexity of compiler development and, by extension, has enabled the proliferation of new languages. Intermediate quantum representations also serve as a bridge between the front-end (high-level quantum languages) and the back-end (quantum hardware implementations) and include quantum-specific instructions such as quantum gates, qubit registers, and entanglement operations. These abstractions are essential for the future of quantum computing, as they will enable the development of more efficient compilers, foster new quantum programming languages and ultimately make quantum computing more accessible to end users [1, 20].

2 Background

In the early days of computing, programs were written in assembly language, which is highly specific to the underlying hardware architecture. This close coupling between software and hardware severely limited code reusability across different machines. Assembly language, although highly efficient for specific hardware, hindered software portability and presented a problem for developers, who had to manage hardware-specific instructions manually. As a result, the need for higher levels of abstraction in programming languages became evident. Higher-level languages, such as Fortran, COBOL, and later C, emerged to abstract the complexities of hardware-specific details, making programming more efficient and portable [21].

One of the critical innovations that facilitated this shift was the development of compilers. A compiler translates high-level code into machine code, effectively decoupling the programming language from the hardware. This abstraction made it possible to write software once and run it on multiple architectures without modification, fostering code reusability. However, developing a compiler for every programming language and hardware architecture combination was a monumental task. This led to introducing intermediate representations (IR), which bridge the high-level source code and the low-level machine instructions. An IR allows a single high-level language to be translated into an intermediate form, which can then be compiled into machine code for different architectures. This reduced the complexity of compiler design, as developers only needed to target the IR rather than every possible hardware platform directly. The IR allows for a modular compilation process. Fig. 1 shows this advantage incorporating an IR into a simple compilation scheme, where the number of compilers required scales from a potentially exponential number of $n \cdot m$ (Fig. 1a) to a more manageable $n + m$ compilers (Fig. 1b), where n is the number of high-level languages and m the number of hardware platforms.

One of the most influential developments in intermediate representations was creating the Low-Level Virtual Machine (LLVM) framework [22], initially designed for C and C++ programs. LLVM introduced a flexible, retargetable IR that could be used across multiple hardware platforms, making it a foundational tool in modern compiler design. LLVM abstracts code into a platform-independent form that can later be optimized and translated into architecture-specific machine code. Its modular design has allowed LLVM to support a wide range of languages beyond C++, including Swift, Rust, and Julia, making it a cornerstone in the development of modern compilers [23–25].



(a) Front-end to back-end scheme. (b) IR as middle-end in the scheme

Fig. 1: Comparison between integrating IRs into a simple compilation scheme.

However, as computing systems have evolved, the need for more advanced IRs has grown, particularly with the rise of heterogeneous systems. These systems combine different types of processors, such as central processing units (CPUs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs), each of which excels at different types of computations [26, 27]. In GPUs and FPGAs, where tasks are highly parallelized, traditional IRs had to be adapted for efficient task distribution and execution. OpenCL, CUDA, and other parallel computing platforms introduced specialized IRs to manage the complexity of these devices, ensuring that high-level code could be effectively translated into machine-level instructions capable of running on these specialized architectures.

The Multi-Level Intermediate Representation (MLIR) [28] is a significant extension of LLVM, designed to address the complexity of heterogeneous architectures by supporting multiple levels of abstraction. MLIR enables high-level optimizations for domains like machine learning while still providing low-level hardware-specific optimizations. This multi-level approach facilitates parallelism, concurrency, and communication across different devices, making it ideal for handling the needs of heterogeneous systems. This is achieved by implementing the concept of dialects. Each dialect is a specialized language with its own vocabulary—operations and types—tailored to a specific subject, for instance, as already mentioned, machine learning. Just as you would choose a language that best suits a topic, in MLIR, you choose dialects that best match the domain you are working in.

In addition to MLIR, Standard Portable Intermediate Representation (SPIR) [29] is another vital IR in the domain of heterogeneous systems. SPIR was developed for the OpenCL framework to provide a platform-neutral IR that enables code portability across different hardware platforms, including CPUs, GPUs, and FPGAs. SPIR simplifies the compilation of OpenCL kernels into optimized machine code, ensuring performance and compatibility across diverse architectures.

NVVM-IR (NVIDIA Virtual Machine Intermediate Representation) plays a similar role for NVIDIA hardware systems. Based on LLVM, NVVM-IR supports CUDA, NVIDIA's parallel computing platform. It abstracts CUDA code into a form that can be optimized and compiled for efficient execution on NVIDIA GPUs, thus improving performance and enabling parallelization on highly specialized hardware.

With the advent of quantum computing, the complexity of hardware has reached an entirely new level, requiring an entirely new class of intermediate representations. Quantum computing platforms, unlike classical ones, are based on principles such as superposition and entanglement, which require fundamentally different instruction sets. Quantum software frameworks such as Qiskit and Cirq were developed to allow high-level quantum programming. These frameworks generate code that can run on quantum hardware, such as superconducting qubits, trapped ions, and photonic systems. However, each of these hardware platforms has unique characteristics, making the development of a unified quantum IR essential for the long-term scalability of quantum programming.

3 Quantum Intermediate Representations: characteristics and classification

In this section, the characteristics that a standard IR should fulfil are defined by compiling information from different citations in the literature. In the following, the quantum IRs developed in the literature are classified and detailed, and a qualitative comparison of different characteristics important for quantum software is made. Finally, an example code for a quantum teleport circuit is shown.

3.1 Characteristics of an Intermediate Representation

An Intermediate Representation (IR) has to meet specific characteristics that distinguish it from a high-level or machine code language. A fundamental question has to be asked: Why is C language—or any other high-level language— not an intermediate representation?

Different characteristics can be analyzed to define an IR. The most important one we should focus on is that **an IR is created by and for machines**. Therefore, it does not need to be fully readable by the human encoder; in most situations, the IR code is encoded in binary.

First, an IR has to be abstract enough to represent a set of high-level languages (HLLs) rather than just one. This implies that it is at a different level from HLLs and machine codes, so there is a process of compilation, not transpilation.

On the other hand, the compilation process to the IR **must keep the information from the compiler's previous analysis phase**. For example, in quantum computing, it would be a mistake to transpose high-level gates to a set of elementary gates, essentially because hardware particularities such as the supported gate set itself or the error propagated by each gate are not known.

Some of the characteristics of an IR that can be taken into account are the following [30]:

- **Comprehensive Representation:** The IR must encapsulate all necessary constructs, abstractions, and concepts from programming languages to ensure precise execution across diverse computing platforms. A key measure of this capability is how easily the IR can be transformed into and from widely used IRs across multiple programming languages.
- **Device Independence:** The IR should remain neutral to specific hardware features. Its execution model should reflect the programming language's semantics rather than the underlying hardware, allowing it to be compiled across various devices. This neutrality must be achieved by carefully balancing the abstraction level.
- **Direct Programmability:** Like assembly languages, IRs offer programmers the ability to fine-tune their code manually. This is beneficial not only for optimization purposes but also for supporting compiler developers during the construction process. Typically, higher-level IRs make manual programming more straightforward.
- **Forward Compatibility:** As programming languages evolve, the IR must be flexible enough to integrate new paradigms without sacrificing backward compatibility. This adaptability is crucial to ensure the IR remains relevant and functional as programming practices change over time.

From the compiler design perspective, the following three attributes are critical for an IR's effectiveness as a program representation tool during the compilation process:

- **Simplicity in Design:** The ideal IR should limit the variety of its constructs while still capturing all computations expressible by source languages. This simplicity facilitates the canonicalization process, where source code is standardized before optimization, thereby reducing code variation and easing the compiler's workload.
- **Retention of Program Details:** The original source code contains the richest information about the program. If critical details are lost during translation, optimization can suffer. Therefore, the IR should include mechanisms to retain important high-level details, such as type information and pointer aliasing, which are essential for effective optimization.
- **Inclusion of Analytical Data:** Successful program transformations often rely on additional data, such as information on data dependencies and aliasing patterns. Embedding this analytical data in the IR allows it to be used by different parts of the compiler. However, this must be managed carefully to avoid the risk of invalidation by later transformations. Balancing the inclusion of analytical information with the complexity it adds to the IR is a crucial design consideration.

3.2 Intermediate Representations for Quantum Computing

In the compilation process, IR serves as a crucial intermediary between high-level programs and machine-executable instructions, as has already been shown. This representation improves translation efficiency and allows for optimization. When it comes to quantum computing, specialized IR becomes essential to take full advantage of the intrinsic characteristics of quantum computing. A variety of IR languages have been developed to bridge the gap between high-level quantum programming languages and low-level quantum machine code. In this subsection, we will attempt to

Language extensions	
Name	**Language**
Q-MLIR [3]	MLIR [28]
QIR [1]	LLVM [22]
QSSA [2]	SSA [31]
QIRO [32]	MLIR and SSA

(a)

Standalone
Nombre
SQIR [33]

(b)

IR Framework	
Name	**Framework**
XACC IR [18]	XACC
QBIR [34]	Yao
t|ket⟩ IR [35]	t|ket⟩

(c)

Table 1: Classification of existing IRs in the literature.

address the large number of IR languages proposed in the literature. Table 1 shows the classification of the IR of the literature in the following categories:

- **Language extensions:** IRs that extend an intermediate representation of classic computation to add components of quantum features.
- **Standalone languages:** IRs that have been designed for quantum computing without having any basis in another classic IR.
- **IR Framework:** an IR that is integrated within a complete compilation scheme or framework and has been created specifically for that scheme.

The initial intermediate language to be examined is **Quantum Intermediate Representation (QIR)** [1]. QIR, developed by the QIR Alliance, which counts Microsoft among its members, should be distinguished from the broader concept of quantum Intermediate Representation (qIR) previously outlined. It functions as a universal interface between quantum programming languages or frameworks and various quantum computing platforms. Moreover, it delineates a series of protocols for representing quantum programs in a language and hardware-neutral format within the LLVM IR [22]. Concerning the translation from high-level languages, QIR remains non-specific to any particular quantum programming framework, thereby facilitating its adoption for articulating quantum programs. Conversely, regarding the translation of IR to machine-specific instructions, QIR is designed to be hardware-independent, abstaining from prescribing a specific quantum instruction or gate set and instead deferring to the preferences of the target computation environment.

Moving on to other quantum IRs, Q-MLIR, an extension of quantum computing to the IR Multi-Level Intermediate Representation (MLIR) described in Section 2, is introduced in citeMcCaskey2021, highlighting the potential of this dialect to conform to the QIR standards recently proposed by Microsoft. This facilitates a shared optimization and execution generation framework across multiple source languages.

Furthermore, additional quantum computing oriented extensions of MLIR, such as **Quantum Intermediate Representation for Optimization (QIRO)** detailed in [32], are noteworthy. The QIRO framework is tailored for quantum-classical co-optimization and embeds data flow directly within the IR, enabling a range of optimizations through data flow analysis. It comprises two dialects: one for input and another for optimization. In contrast, Q-MLIR focuses on defining a quantum IR by extending MLIR but does not explore or implement optimizations that leverage MLIR's capabilities for quantum program improvement.

```
int a = 10;
int b = a + 5;
a = a * 2;
int c = a * b;
return c;
```

(a) Example of C++ source code.

```
%1 = 10
%2 = %1 + 5
%1 = %1 * 2
%4 = %1 * %2
return %4
```

(b) Representation in LLVM IR without SSA.

```
%1 = 10
%2 = %1 + 5
%3 = %1 * 2
%4 = %3 * %2
return %4
```

(c) Representation in LLVM IR with SSA.

Fig. 2: Comparison between an LLVM IR code with and without SSA.

Moreover, IRs can adhere to specific properties or constraints to facilitate optimization and verification processes. One such property, extensively explored in classical compilation literature, is the Static Single Assigment (SSA) form [31, 36, 37]. An SSA form mandates that each variable in the source code is assigned uniquely in the intermediate representation. This implies that additional variables must be introduced if a variable receives assignments at multiple points in the source code to denote the distinct versions required.

Figure 2 exemplifies the LLVM IR translation of C++ source code where the variable `a` undergoes reassignment. In an SSA compliant code, it is imperative to generate a new variable (`%3` as shown in Figure 2c) to signify the updated version, thereby preserving the original variable (`%1` as depicted in Figure 2b).

The usefulness of this property in verification and optimization, together with the extensive study in classic compilation, allows the use of this approach in quantum compilation as a suitable and perfectly tested basis. The IR previously discussed, QIRO, uses this SSA property for the co-optimization of quantum codes.

On the other hand, the work presented in [2] as **Quantum Static Single Assignment (QSSA)** performs an extension of the SSA properties to generate a quantum IR that performs a special emphasis on the verification in the compile-time of the physical constraints of quantum nature, such as the no-cloning theorem.

Figure 3 shows a graphical representation of the various quantum IRs that extend a classic representation such as LLVM. It is shown in a set format to characterize those representations that extend more than one language or feature.

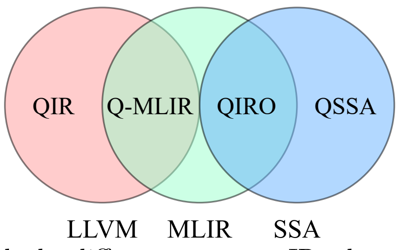


Fig. 3: Venn diagram with the different quantum IRs shown in the "Language extensions" section. Each set represents a language or feature of classic computing, and each element is a quantum IR.

Outside the classic IR extensions such as LLVM or MLIR, there are also standalone IR languages like **Small Quantum Intermediate Representation (SQIR)**, proposed in [33]. Its primary purpose is to serve as an intermediate language to verify the correctness of the quantum circuit once optimizations have been applied to it. This IR is built on top of the mathematical definition language Coq.

In addition, some frameworks implement the complete compilation scheme or a high-level language that develops its specific IR, such as **eXtreme-scale Accelerator programming framework (XACC)** [18] that implement the compilation scheme in full, based on a three-level scheme: a front-end that maps the quantum code to its own IR; a middle-end related to the transformation and optimization of the IR, and, finally, a backend that processes the IR to the specific machine code. Another example is the **Quantum Block Intermediate Representation (QBIR)** [34], which is used in the high-level quantum language Yao.jl.

3.3 Comparision of quantum IRs

The purpose of this section is to provide a comparative analysis of the different IRs discussed in this work. It is important to emphasize that this is a qualitative comparison to maintain objectivity, as a quantitative study is not viable due to the lack of standardized benchmarks and the different levels of maturity of the different IRs. To achieve this, a set of key properties has been established to evaluate how each IR performs in different contexts. This comparison aims not to declare one IR superior to another but to provide insights that will help readers decide which IR is best suited to their particular implementation priorities. This framework allows readers to make informed decisions based on their specific needs, whether hardware compatibility, optimization efficiency, or support for hybrid quantum-classical operations.

Table 2 presents a qualitative comparison of seven quantum IRs: QIR, SQIR, QSSA, QIRO, XACC, QBIR, and Q-MLIR, based on the next essential features for quantum software development:

1. **Hybrid Quantum-Classical Operations:** Quantum programs often combine classical and quantum computations, especially for hybrid algorithms like the Variational Quantum Eigensolver (VQE). An IR that supports hybrid operations allows efficient execution of both quantum instructions and classical control logic (e.g., conditionals, loops).
2. **Hardware Agnosticism:** an IR hardware agnosticism determines how well it abstracts away hardware-specific details, making code portable across various quantum architectures (e.g., superconducting qubits, trapped ions, photonic systems).
3. **Optimizations and Compiler Passes:** Quantum computations require significant optimizations (e.g., gate reduction, circuit simplifications) to be executable on quantum hardware, where resource constraints like qubit coherence times and gate fidelity are critical.
4. **Expressiveness:** refers to how flexibly the IR can describe quantum programs, supporting modularity, advanced quantum gates, and complex quantum operations.

5. **Simulator Compatibility:** The ability to simulate quantum circuits before executing them on actual hardware is a crucial step in quantum program development.
6. **Community Support:** IRs community support is critical for its development and adoption. A strong ecosystem ensures access to tools, libraries, and active collaboration.

In the context of table 2, qualifiers such as High or Medium are used to provide a qualitative assessment of how well an IR matches the characteristic being assessed. For example, in the case of Expressiveness, most IRs are rated High due to their ability to represent a wide range of quantum operations. However, Q-MLIR is rated Very High because its multi-level structure allows for the definition of new instructions, providing greater flexibility and extensibility compared to other IRs.

Each quantum IR brings unique strengths to quantum software development. QIR and Q-MLIR perform particularly well in handling hybrid quantum-classical operations, optimizations and hardware agnosticism, which makes them an excellent fit for scalable quantum computing applications. SQIR excels in formal verification, while XACC offers solid support across the software stack and full compilation. QSSA and QIRO emphasize optimizations and error correction, which are fundamental to long-term fault-tolerant quantum computing.

By comparing these features, developers can choose the most appropriate IR based on their specific needs, such as hardware compatibility, optimization and expressiveness.

Feature	QIR	Q-MLIR	QSSA	QIRO	XACC	QBIR	SQIR
Hybrid Quantum Classical Ops	Yes (via LLVM)	Yes	Yes	Yes	Yes	No	No
Hardware Agnosticism	High	High	High	Medium	High (via XACC compiler)	High	High (Coq-based)
Optimizations	Built-in (LLVM passes)	Built-in (via MLIR)	Compiler Optimizations	Limited	LLVM Optimizations	Circuit Simplifications	Formal Verification
Expressiveness	High	Very High	High	High	High	Medium	Medium
Simulator Compatibility	High (LLVM simulators)	High	Medium	Medium	High	Medium	High
Community Support	Strong (LLVM, Microsoft)	Growing (MLIR community)	Growing (Quantum research)	Limited	Strong (XACC framework)	Limited	Growing (Coq community)

Table 2: Qualitative comparison of the different IRs existing in the bibliography.

3.4 Example of code: teleport circuit

In this subsection, two IRs that extend LLVM, QIR and Q-MLIR, will be compared to analyze the differences between the two models. We aim to demonstrate how different IRs work when implementing a complex circuit, such as quantum teleportation, where several critical aspects are involved, such as quantum gates between qubit pairs, intermediate measurement, or physically separated qubits.

The circuit to be implemented is the one shown in Figure 4, corresponding to the classical teleport circuit between two physically separated qubits. The codes generated for this circuit are shown in Figure 5a for QIR and Figure 5b for Q-MLIR.

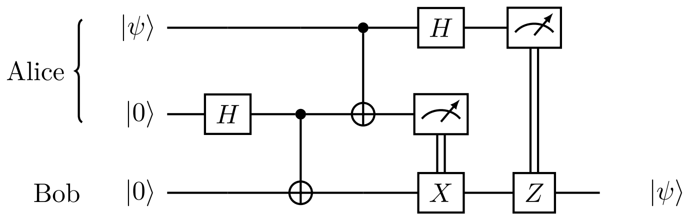


Fig. 4: Circuit for teleporting a quantum state $|\psi\rangle$ between two physically separated qubits (from Alice to Bob).

The QIR code for the teleport circuit is very similar to the LLVM IR format. It focuses on low-level quantum operations and directly invokes quantum gates like Hadamard and CNOT with explicit quantum instruction calls. Measurements are performed with conditions applied to control subsequent gates (X and Z corrections), reflecting the hybrid nature of QIR, which integrates quantum operations with classical control through conditional branches. This low-level representation ensures detailed control and optimization but requires more explicit operations management.

In contrast, the Q-MLIR code operates at a higher level of abstraction, using MLIR's modular structure. While the circuit structure is similar—applying Hadamard and CNOT gates followed by measurements—the code is more declarative. The quantum gates and measurements are invoked in a more abstract manner, emphasizing flexibility and modularity rather than fine-tuned control. Q-MLIR simplifies the representation, making it easier to map to different hardware backends or domain-specific optimizations while maintaining a clearer structure for higher-level quantum algorithms.

These examples show that both QIR and Q-MLIR effectively handle intermediate measurements (hybrid quantum-classical operations) and quantum gates between qubit pairs. However, the major unresolved challenge lies in communication between physically separated qubits. The current IRs do not specify whether qubits are located on different nodes, a crucial factor in distributed quantum computing (DQC). Without addressing this issue, managing the necessary quantum communication and synchronization between distant quantum processors becomes impossible.

To address this limitation, the following section will explore various intermediate languages specifically designed for DQC, which incorporate mechanisms to handle physically separated qubits and quantum communication.

```
define void @Teleportation__body(%Qubit* %msg, %Qubit* %ent1, %Qubit* %ent2) {
entry:
  ; Create entanglement between ent1 and ent2
  call void @__quantum__qis__h(%Qubit* %ent1)
  call void @__quantum__qis__cnot(%Qubit* %ent1, %Qubit* %ent2)

  ; Apply CNOT between msg and ent1 and next the Hadamard gate to msg
  call void @__quantum__qis__cnot(%Qubit* %msg, %Qubit* %ent1)
  call void @__quantum__qis__h(%Qubit* %msg)

  ; Measure msg and ent1
  %meas_msg = call %Result* @__quantum__qis__mz(%Qubit* %msg)
  %meas_ent1 = call %Result* @__quantum__qis__mz(%Qubit* %ent1)

  ; Apply the classical controlled gates
  call void @__quantum__qis__x(%Qubit* %ent2) if (%meas_ent1 == 1)
  call void @__quantum__qis__z(%Qubit* %ent2) if (%meas_msg == 1)

  ret void
}
```

(a) QIR code

```
module {
  func @Teleportation(%msg: !quantum.qbit, %ent1: !quantum.qbit, %ent2: !quantum.qbit) {
    // Entanglement creation
    "quantum.h"(%ent1) : (!quantum.qbit) -> ()
    "quantum.cnot"(%ent1, %ent2) : (!quantum.qbit, !quantum.qbit) -> ()

    // Teleportation step
    "quantum.cnot"(%msg, %ent1) : (!quantum.qbit, !quantum.qbit) -> ()
    "quantum.h"(%msg) : (!quantum.qbit) -> ()

    %m1 = "quantum.mz"(%msg) : (!quantum.qbit) -> !quantum.result
    %m2 = "quantum.mz"(%ent1) : (!quantum.qbit) -> !quantum.result

    "quantum.x"(%ent2) if %m2 == 1
    "quantum.z"(%ent2) if %m1 == 1
    return
  }
}
```

(b) Q-MLIR code

Fig. 5: Implementation of a teleport circuit in two quantum intermediate representations (QIR and Q-MLIR).

4 IRs for Distributed Quantum Computing (DQC)

In this section, we describe IRs for programming distributed quantum computers, trying to overcome the shortcoming of all quantum IRs presented in the previous section: the inability to define physically separated qubits. In DQC, where quantum systems are distributed over multiple physical locations, it becomes crucial to specify the locality of each qubit and to manage the communication between qubits located in different nodes.

In distributed quantum systems, operations like quantum teleportation and entanglement swapping require an IR that can effectively model both the quantum gates and the communication between distant qubits [38]. This need leads to the introduction of InQuIR (Intermediate Representation for Interconnected Quantum Computers) [4], an IR designed to overcome the limitations of previous representations by incorporating explicit support for qubit locality and inter-node communication.

As an illustrative example, consider the teleporting circuit, where the goal is to transfer the state of a qubit from one node to another. In a distributed quantum system, this involves entangling two qubits at different locations and using classical communication to transfer the qubit's state. The InQuIR code is shown in Figure 6, where it can be seen that there is a different code for each qubit node (Alice or Bob). Therefore, this IR allows to specify the physical separation of a register of qubits and, additionally, to program them in a different way depending on this characteristic.

It is important to highlight the functions incorporated in this new representation, such as `genEnt()` to generate the entanglement between the systems and `send()` and `recv()` to send measurements (classic bits).

```
0 {
  world = open[0,1];
  q0 = init();
  _cq0 = genEnt[1](10);
  CX q0 _cq0; H q0;
  _m0 = measure _cq0;
  _m1 = measure q0;
  free _cq0; free q0;

  send[1](world, l1:_m0);
  send[1](world, l1_2:_m1);
}
```

(a) InQuIR code, `0` node (Alice).

```
1 {
  world = open[0];

  q1 = init();
  _cq1 = genEnt[0](10);

  recv(world, l1:_m3);
  recv(world, l1:_m4);

  X[_m3] _cq1;
  Z[_m4] _cq1;
}
```

(b) InQuIR code, `1` node (Bob).

Fig. 6: InQuIR code to implement the teleport circuit in the different nodes.

InQuIR provides important tools for managing qubit locality and inter-node communication in distributed quantum computing. However, its limitations arise from its dependence on manual control over communication processes, which restricts its ability to fully abstract the complexities inherent in quantum networking. In particular, InQuIR's reliance on explicit entanglement generation (`genEnt`) and manual control over the transmission of classical data (`send` and `recv`) requires developers to work directly with the low-level details of the quantum communication process. This lack of abstraction contrasts strongly with the practice in classical distributed computing, where frameworks like MPI effectively hide the details of the underlying communication protocols.

The abstraction of the communication layer in traditional distributed systems provides flexibility to work with different network architectures and protocols without exposing their complexity to the user, which allows developers to focus on high-level logic without managing the intricacies of the interconnection network. However, InQuIR's dependence on explicitly managing communication through teleportation or entanglement protocols requires developers to handle the complexities of quantum networking, which can increase the risk of errors and limit scalability. As quantum networks grow in complexity, with more nodes and greater distances between them, this approach becomes less feasible and more cumbersome. The need for a more abstract and flexible system becomes evident, one that can manage communication efficiently and transparently.

To address these issues, NetQIR [5] emerges as an extension of QIR [1]. NetQIR introduces high-level instructions such as `qsend()` for transmitting qubits and `expose()` for making qubits available to other nodes in the network. These instructions abstract the communication process, allowing the system to automatically manage the complexities of quantum entanglement, teleportation, and classical communication. This approach mirrors the abstraction seen in classical distributed computing frameworks, where message-passing or data synchronization details are hidden from the user. By integrating such abstraction, NetQIR not only simplifies the development process but also increases the flexibility and scalability of distributed quantum systems.

Figure 7 shows the same teleport circuit discussed throughout this article, highlighting the reduction in code complexity achieved by abstracting the communication protocol. The implementation becomes more agile and efficient by delegating these low-level details to the backend, which has access to more specific information about the connecting network and the hardware involved.

A significant advantage of NetQIR is that, by extending QIR, it inherits a strong foundation from the QIR framework, which itself is built on LLVM. This allows NetQIR to leverage the extensive optimization and tooling infrastructure developed for QIR and LLVM, including compiler passes, simulation tools, and debugging environments. As a result, developers can utilize high-level abstractions for distributed quantum operations and benefit from the robust ecosystem of tools available in the LLVM framework. On the other hand, one of the main disadvantages of NetQIR is that it is currently still in a work-in-progress phase. While the IR has been defined conceptually, there is a significant gap in the availability of practical tools and infrastructure for working with it. Unlike established quantum IRs such as QIR, which benefit from a mature ecosystem of compilers, optimizers, and simulators, NetQIR lacks concrete implementations and tools at this stage. As a result, developers and researchers are not yet able to fully utilize NetQIR in real-world distributed quantum systems.

5 Conclusions

This work explored the design, evaluation, and limitations of intermediate representations (IRs) for monolithic and distributed quantum computing. To develop an effective IR, certain core characteristics must be defined, such as hardware agnosticism, optimization capabilities, modularity, and the ability to handle quantum-classical hybrid operations. These features ensure that an IR can bridge the gap between high-level quantum programming languages and the hardware, ensuring efficient performance and adaptability across various quantum architectures. This paper began by reviewing quantum IRs designed for monolithic quantum systems, followed by a more in-depth discussion of the challenges associated with IRs for distributed quantum computing.

```
define void @main(i32 noundef %0, ptr noundef %1) #0 {
    entry:
        ; Variable allocation
        %2 = alloca i32, align 4

        ; Init the NetQIR communication and get the rank of the process
        %3 = call i32 @__netqir__init(i32 noundef %0, ptr noundef %1)
        %4 = call i32 @__netqir__comm_rank(%Comm* @netqir_comm_world, ptr %2)

        ; Choose if it is the process receiving or sending the qubit
        %5 = load i32, ptr %2, align 4
        %6 = icmp eq i32 %5, 0
        br i1 %6, label %7, label %9

    ; Process sending
    7:
        %8 = call i32 @__netqir__qsend(%Qubit* null, i32 noundef 1,
                                       %Comm* @netqir_comm_world)
    ; Process receiving
    9:
        %10 = call i32 @__netqir__qrecv(%Qubit** null, i32 noundef 0,
                                        %Comm* @netqir_comm_world)

        ; End of the program
        %11 = call i32 @__netqir__finalize()
}

; Function declaration
declare i32 @__netqir__init()
declare i32 @__netqir__qsend(%Qubit*, i32, %Comm)
declare i32 @__netqir__qrecv(%Qubit**, i32, %Comm)
declare void @__netqir__finalize()
```

Fig. 7: NetQIR code for the state teleportation between Alice and Bob.

The study of monolithic quantum computing focused on IRs like QIR, Q-MLIR, SQIR or QSSA, which are primarily designed for quantum processors located in a single physical location. These IRs support features such as gate-level abstractions, hardware-agnostic instruction sets, and optimizations targeting performance across various quantum platforms. They are well-suited for tasks that do not involve inter-node communication or distributed architectures. However, they still face challenges in scaling with increasing qubit counts and handling the specificities of different quantum technologies.

The evolution of quantum IRs reflects the need for such tools to facilitate the development of new compilers and a complete quantum software stack. This work highlights the current limitations of quantum IRs, particularly the lack of support for managing qubit locality in systems where quantum processors are distributed across multiple physical locations. Therefore, IRs for DQC, such as InQuIR or NetQIR, have emerged in the literature.

InQuIR takes an important step by allowing programmers to define qubit locality and manage communication explicitly, but this approach has significant drawbacks. Namely, it requires developers to handle low-level communication protocols (e.g., `genEnt`, `send`, `recv`) manually, adding to the complexity and potential for error. In contrast, NetQIR introduces a higher level of abstraction, where instructions like `qsend` and `expose` delegate the details of quantum communication to the backend.

This allows for a more flexible and scalable programming model, similar to how classical distributed systems abstract network protocols through frameworks like MPI.

The key strength of NetQIR is its foundation in QIR, which is itself built on LLVM. This allows it to take advantage of LLVM's well-established ecosystem of optimization tools, compilers, and simulators. This strong base ensures that NetQIR not only simplifies the programming model for distributed quantum computing but also benefits from powerful optimizations and debugging capabilities developed for classical computing.

In summary, while InQuIR provides an initial solution for managing distributed qubit locality and communication, its explicit handling of communication protocols limits its scalability. NetQIR, as an extension of QIR, offers a more abstract, scalable solution for distributed quantum systems. By integrating high-level instructions that hide the complexity of quantum communication, NetQIR provides a pathway to more efficient and manageable distributed quantum computing frameworks. However, it is important to note that NetQIR remains in a **work-in-progress phase**, with the intermediate representation defined but without practical tools available yet for real-world implementations. Further development is required to fully realize its potential in distributed quantum systems.

Acknowledgements. This work was supported by MICINN through the European Union NextGenerationEU recovery plan (PRTR-C17.I1), and by the Galician Regional Government through the "Planes Complementarios de I+D+I con las Comunidades Autónomas" in Quantum Communication. This work was also supported by financial support from the Agencia Estatal de Investigación (Spain) (PID2022-141623NB-I00), the Xunta de Galicia - Consellería de Cultura, Educación, Formación Profesional e Universidades (Centro de investigación de Galicia accreditation 2024-2027 ED431G-2023/04 and Reference Competitive Group accreditation ED431C-2022/016) and the European Union (European Regional Development Fund - ERDF).

References

[1] Lubinski, T., Granade, C., Anderson, A., Geller, A., Roetteler, M., Petrenko, A., Heim, B.: Advancing hybrid quantum–classical computation with real-time execution. Frontiers in Physics **10** (2022) https://doi.org/10.3389/fphy.2022.940293

[2] Peduri, A., Bhat, S., Grosser, T.: QSSA: an SSA-based IR for quantum computing. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. CC 2022, pp. 2–14. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497776.3517772

[3] McCaskey, A., Nguyen, T.: A MLIR dialect for quantum assembly languages. Proceedings - 2021 IEEE International Conference on Quantum Computing and Engineering, QCE 2021, 255–264 (2021) https://doi.org/10.1109/QCE52317.2021.00043

[4] Nishio, S., Wakizaka, R.: InQuIR: Intermediate Representation for Interconnected Quantum Computers (2023). https://arxiv.org/abs/2302.00267

[5] Vázquez-Pérez, J., Cardama, F.J., Piñeiro, C., Pena, T.F., Pichel, J.C., Gómez, A.: NetQIR: An Extension of QIR for Distributed Quantum Computing (2024). https://arxiv.org/abs/2408.03712

[6] Alfred, V.A., Monica, S.L., Jeffrey, D.U.: Compilers Principles, Techniques & Tools. Pearson Education, Inc, London, UK (2007)

[7] Schneck, P.B.: A survey of compiler optimization techniques. In: Proceedings of the ACM Annual Conference, pp. 106–113 (1973)

[8] Tremblay, J.-P., Sorenson, P.G.: Theory and Practice of Compiler Writing. McGraw-Hill, Inc., New York (1985)

[9] Torczon, L., Cooper, K.: Engineering a Compiler. Morgan Kaufmann Publishers Inc., Cambridge, MA (2007)

[10] Conway, M.E.: Proposal for an UNCOL. Commun. ACM **1**(10), 5–8 (1958) https://doi.org/10.1145/368924.368928

[11] Ottenstein, K.J.: Intermediate program representations in compiler construction: A supplemental bibliography. SIGPLAN Not. **19**(7), 25–27 (1984) https://doi.org/10.1145/988574.988579

[12] Stanier, J., Watson, D.: Intermediate representations in imperative compilers: A survey. ACM Comput. Surv. **45**(3) (2013) https://doi.org/10.1145/2480741.2480743

[13] Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997) https://doi.org/10.1137/s0097539795293172

[14] Steane, A.: Quantum computing. Reports on Progress in Physics **61**(2), 117 (1998) https://doi.org/10.1088/0034-4885/61/2/002

[15] Ladd, T.D., Jelezko, F., Laflamme, R., Nakamura, Y., Monroe, C., O'Brien, J.L.: Quantum computers. Nature **464**(7285), 45–53 (2010) https://doi.org/10.1038/nature08812

[16] Chong, F.T., Franklin, D., Martonosi, M.: Programming languages and compiler design for realistic quantum hardware. Nature Publishing Group (2017). https://doi.org/10.1038/nature23459

[17] Häner, T., Steiger, D.S., Svore, K., Troyer, M.: A software methodology for compiling quantum programs. Institute of Physics Publishing (2018). https://doi.org/10.1088/2058-9565/aaa5cc

[18] McCaskey, A.J., Lyakh, D.I., Dumitrescu, E.F., Powers, S.S., Humble, T.S.: XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. Quantum Science and Technology **5**(2), 024002 (2020) https://doi.org/10.1088/2058-9565/ab6bf6

[19] Mintz, T.M., McCaskey, A.J., Dumitrescu, E.F., Moore, S.V., Powers, S., Lougovski, P.: QCOR: A language extension specification for the heterogeneous quantum-classical model of computation. J. Emerg. Technol. Comput. Syst. **16**(2) (2020) https://doi.org/10.1145/3380964

[20] McCaskey, A.J., Dumitrescu, E.F., Liakh, D., Chen, M., Feng, W., Humble, T.S.: A language and hardware independent approach to quantum-classical computing. SoftwareX **7**, 245–254 (2018) https://doi.org/10.1016/j.softx.2018.07.007

[21] Knuth, D.E., Pardo, L.T.: The early development of programming languages. A History of Computing in the Twentieth Century, 197–273 (1980) https://doi.org/10.1016/B978-0-12-491650-0.50019-8

[22] Foundation, L.: LLVM Assembly Language Reference Manual. https://releases.llvm.org/2.6/docs/LangRef.html

[23] Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665

[24] Terei, D.A., Chakravarty, M.M.: Low level virtual machine for glasgow haskell compiler. PhD thesis, Bachelor's Thesis, Computer Science and Engineering Dept., The University of New South Wales (2009)

[25] Ştirb, I., Gillich, G.-R.: A low-level virtual machine just-in-time prototype for running an energy-saving hardware-aware mapping algorithm on C/C++ applications that use pthreads. Energies **16**(19) (2023) https://doi.org/10.3390/en16196781

[26] Weaver, G.: Compiler representations for heterogeneous processing. Technical Report UM-CS-1995-102, University of Massachusetts (1995)

[27] Belwal, M., TSB, S.: Intermediate representation for heterogeneous multi-core: A survey. In: 2015 International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA), pp. 1–6 (2015). https://doi.org/10.1109/VLSI-SATA.2015.7050496

[28] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization,

2–14 (2021) https://doi.org/10.1109/CGO51591.2021.9370308

[29] Kessenich, J., Ouriel, B., Krisch, R.: SPIR-V specification. Khronos Group **3**, 17 (2018)

[30] Chow, F.: Intermediate representation. Queue **11**, 30–37 (2013) https://doi.org/10.1145/2542661.2544374

[31] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '89, pp. 25–35. Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/75277.75280

[32] Ittah, D., Häner, T., Kliuchnikov, V., Hoefler, T.: QIRO: A static single assignment-based quantum program representation for optimization. ACM Transactions on Quantum Computing **3**(3) (2022) https://doi.org/10.1145/3491247

[33] Hietala, K., Rand, R., Hung, S.-H., Wu, X., Hicks, M.: Verified Optimization in a Quantum Intermediate Representation (2019). https://arxiv.org/abs/1904.06319

[34] Luo, X.-Z., Liu, J.-G., Zhang, P., Wang, L.: Yao. jl: Extensible, efficient framework for quantum algorithm design. Quantum **4**, 341 (2020) https://doi.org/10.22331/q-2020-10-11-341

[35] Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., Duncan, R.: t$|$ket$\rangle$: a retargetable compiler for NISQ devices. Quantum Science and Technology **6**(1), 014003 (2020) https://doi.org/10.1088/2058-9565/ab8e92

[36] Van Emmerik, M.J.: Static single assignment for decompilation. PhD thesis, University of Queensland, Queensland, Australia (2007)

[37] Rastello, F., Tichadou, F.B.: SSA-based compiler design. Springer International Publishing, 1–382 (2022) https://doi.org/10.1007/978-3-030-80515-9

[38] Barral, D., Cardama, F.J., Díaz, G., Faílde, D., Llovo, I.F., Juane, M.M., Vázquez-Pérez, J., Villasuso, J., Piñeiro, C., Costas, N., et al.: Review of Distributed Quantum Computing. From single QPU to high performance quantum computing (2024). https://arxiv.org/abs/2404.01265