SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Computational Science and Engineering

Representing Quantum System Information and Topologies in sys-sage

Durganshu Mishra

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Computational Science and Engineering

Representing Quantum System Information and Topologies in sys-sage

Abbilden von Informationen über Quantensysteme und deren Topologien in sys-sage

Author:	Durganshu Mishra
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Stepan Vanecek
Submission Date:	Aug 15, 2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching bei München, Aug 15, 2024 Durganshu Mishra

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Stepan Vanecek, for offering me the remarkable opportunity to contribute to this project. His invaluable guidance, insightful feedback, and unwavering support have not only been instrumental but also influential in shaping the direction and enhancing the quality of this thesis. I am also deeply thankful to my co-supervisor, Jorge Echavarria (LRZ), for his time, expertise, and thoughtful feedback, which played a significant role in refining this research methodology.

I want to sincerely thank Prof. Dr. Martin Schulz for his supervision and support, which were crucial to the successful completion of this work. I am profoundly grateful to Ercüment Kaya (LRZ), Lukas Burgholzer (CDA, TUM), and Ludwig Schmid (CDA, TUM) for their valuable suggestions and feedback on the integration of sys-sage and QDMI within the Munich Quantum Software Stack (MQSS).

I am profoundly grateful to my friends and family for their steadfast encouragement and understanding throughout this academic journey. I would like to especially acknowledge Harish Ramachandran, Gaurav Gokhale, Aditya Phopale and Srishti Upadhyay, for their invaluable support, particularly during the completion of my master's thesis.

My sincere gratitude goes to my parents, Sunita Mishra and Ajay Kumar Mishra, and my brother, Yajush Mishra, whose unwavering support from India has been an enduring source of motivation and strength

This work has been a part of the DEEP-SEA project and Munich Quantum valley. The DEEP-SEA project has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606. National contributions from the involved state members match the EuroHPC funding. Munich Quantum Valley (MQV) is supported by the Bavarian State Government with funds from the Hightech Agenda Bayern.

Abstract

This thesis focuses on enhancing the sys-sage library to extend its functionalities for quantum systems by serving as a unified interface for various quantum technologies. The primary goal is to facilitate the management and representation of these technologies by creating static and dynamic topologies. The work explores the integration of the Quantum Device Management Interface (QDMI) within the compiler toolchain of the Munich Quantum Software Stack (MQSS), with a specific implementation proposed for the IBM backend. Significant modifications and extensions to the sys-sage library are also introduced to enable a higher-level representation of quantum systems. These extensions are complemented by proposed API calls designed to streamline the creation and management of these representations from a user's perspective. Performance and integration aspects of sys-sage within the broader Munich Quantum Software Stack (MQSS) ecosystem are examined, highlighting the potential for enhanced interoperability and functionality across different quantum computing tools. The results demonstrate the feasibility and effectiveness of the proposed enhancements, offering a robust framework for quantum system management.

Abbreviations

QDMI Quantum Device Management Interface

HPC High Performance Computing

QC Quantum Computing

QPU Quantum Processing Unit

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

MQSS Munich Quantum Software Stack

NISQ Noisy Intermediate-Scale Quantum

MQT Munich Quantum Toolkit

HPC-QC High Performance Computing-Quantum Computing

QRM Quantum Resource Manager

NA Neutral Atoms

TI Trapped Ions

FoMaC Figures of Merits and Constraints

QIR Quantum Intermediate Representation

Contents

1. Introduction

Quantum Computing (QC) presents a significant departure from classical computing paradigms by utilizing the principles of quantum mechanics to handle information in ways beyond the capabilities of classical computers. Unlike classical bits, which can strictly be either 0 or 1, quantum bits (qubits) can exist in multiple states simultaneously due to superposition. This characteristic allows quantum computers to perform intricate calculations more efficiently. The distinctive characteristics of qubits, such as superposition and entanglement, enable quantum computers to solve specific problems at exponentially faster rates than classical computers.

However, recent developments in the field have shown that QC does not replace traditional High Performance Computing (HPC). Instead, QC can be integrated into existing heterogeneous HPC infrastructures as an additional accelerator, like Graphics Processing Unit (GPU)s or Field-Programmable Gate Array (FPGA)s. This integration will allow both paradigms to be utilized optimally. This push for integration dramatically impacts the development of quantum computer software, subsequently affecting the required software infrastructure.

Munich Quantum Software Stack (MQSS) aims to address the challenges of integrating quantum computing with classical systems. The comprehensive and full-stack approach involves creating a dedicated quantum software stack that facilitates direct experimental access, integrating HPC systems, and using implicit quantum compilation and optimization toolkits, ultimately supporting multiple backends. The users can interact with the quantum system through a dedicated portal offering multiple language backends or traditional HPC systems using schedulers like SLURM. This enables hybrid applications to offload certain computations to quantum computing backends seamlessly. The quantum resource manager and associated quantum design tools, such as those from the Munich Quantum Toolkit (MQT)[1], process relevant parts of the programs. Appropriate backend systems then execute the resulting quantum circuits. This workflow is designed to be user-transparent, masking the underlying complexities while providing expert paths for experimental computations.

One of the core components of the MQSS is the Quantum Device Management Interface (QDMI). QDMI enables software tools to automatically retrieve and adapt to different Quantum platforms' changing physical characteristics and constraints. One of this thesis's focus areas was developing an initial implementation of QDMI for an IBM quantum backend. The developed backend is an example implementation for all the external vendors and manufacturers who wish to integrate their quantum platforms

[1] `https://www.cda.cit.tum.de/research/quantum/mqt/`

with the MQSS. The QDMI has been written in the C programming language and acts as a low-level query interface for different quantum backends. The information provided by QDMI is provided to compilers for target-specific compilation.

This is where sys-sage[2] comes into the picture. sys-sage is a comprehensive library designed to capture and manipulate the hardware topology of computing systems and their attributes. Initially focused on classical computing elements such as HPC nodes, CPUs, and GPUs, sys-sage provides a robust framework for describing and managing these systems' static and dynamic properties.

In this thesis, we aim to extend the functionalities of sys-sage for quantum systems by acting as a unified interface for different quantum technologies. Further, this work aims to provide a higher-level representation of different quantum technologies in the form of static/dynamic topologies.

The thesis is organized as follows: Chapter 2 focuses on some of the fundamentals of QC, relevant for HPC-QC integration. Concepts on quantum qubits, quantum gates, and the background of different technologies are presented. Chapter 3 outlines the role of QDMI in the compiler toolchain of MQSS. This chapter also presents the core structure of QDMI and proposed implementation for the IBM backend. Chapter 4 outlines the core structure of sys-sage and its internal representation. Further, the extensions to the library to represent the quantum systems are thoroughly discussed. This section also lists the proposed API calls to create such a representation from a user's viewpoint.

Chapter 5 discusses the possibility of integrating sys-sage with other tools within MQSS. Finally, we discuss the conclusions and possible future research directions in Chapter 6.

[2] `https://github.com/caps-tum/sys-sage`

2. Background and Related Works

In this chapter, the necessary foundations for this thesis are established through thorough discussions on the fundamentals and related works on QC, HPC-QC integration, and MQSS.

2.1. Quantum Computing

Quantum computing is a revolutionary paradigm harnessing the laws of quantum mechanics to process and compute information. Unlike classical computing, which uses bits as the basic unit of information, quantum computing uses quantum bits or qubits. By exploiting phenomena such as superposition and entanglement, these qubits can perform computations that are infeasible for classical computers. This potential for unprecedented computational power makes quantum computing highly promising across different fields, including cryptography, material science, optimization problems, and complex system simulations. Nielsen and Chuang [NC12] have comprehensively covered the fundamentals of quantum computing and quantum information science. Quantum algorithms like Shor's and Grover's play vital roles in cryptography, unstructured search problems, and quantum error correction, proving quantum computing's potential in addressing complex problems.

Preskill [Pre18] explores the advancements in quantum computing during the Noisy Intermediate-Scale Quantum (NISQ) era. This significant phase in quantum computing involves the development of devices with 50 to 100 qubits capable of performing computations beyond the capabilities of classical computers despite being hindered by substantial noise and error rates. The discussion addresses the current state and future potential of quantum computing, underscoring the necessity of designing near-term quantum algorithms that can tolerate noise, given that full quantum error correction is not yet practical for these devices.

2.1.1. Quantum Bits (Qubits)

Qubits are the fundamental units of information in a quantum computer. While classical bits can be either in 0 or 1 states, qubits can simultaneously exist in a superposition of states. Mathematically, a qubit is represented as a linear combination of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. This normalization condition ensures that the total probability is 1.

Moreover, qubits can become entangled, indicating that the condition of one qubit can be influenced by the condition of another, regardless of their spatial separation. For example, an entangled state of two qubits can be represented as:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.2}$$

This entanglement property is crucial for the parallelism and enhanced capabilities of quantum computing.

2.1.2. Quantum Technologies

Various technologies are being developed to realize practical quantum computers. These technologies differ in how they implement and manipulate qubits. Some of the most common technologies include superconducting qubits, neutral atoms, and trapped ions-based quantum systems [Lad+10].

Olivier Ezratty [Ezr23] provides an in-depth perspective on superconducting qubit-based systems, representing some of the most advanced and widely utilized quantum computing technologies. These qubits are fabricated from superconducting materials with zero electrical resistance at extremely low temperatures. Typically, superconducting qubits are formed using Josephson junctions, enabling precise control of quantum states via microwave pulses. In the case of superconducting qubits, the parameters α and β in Equation 2.1 are manipulated through external microwave signals. Leading companies like IBM and Google employ superconducting qubits in their quantum processors.

Kjaergaard [Kja+20] reviews several recent advancements in superconducting qubits, including gate implementation techniques, enhanced readout capabilities, early implementations of NISQ algorithms, and quantum error correction strategies.

Mckay et al. [McK+18] explore the specifications and standards for interfacing with quantum devices and simulators through IBM's Qiskit framework, which focuses on developing a common interface and standardized data structures to facilitate quantum experiment execution. Two primary languages are discussed: OpenQASM, which defines quantum circuits, and OpenPulse, which provides pulse-level control of quantum devices, enabling users to specify the time dynamics of their experiments.

One of the most critical properties for representing the topology of superconducting qubit-based systems is the coupling map of the qubits. This map describes the physical layout of the qubits and their interconnections. In the context of quantum gates, the coupling map specifies which pairs of qubits a given gate can operate on, highlighting the permissible interactions based on the physical configuration. Figure 2.1 shows the coupling map of a hypothetical 7-qubit quantum computer.

Wintersperger et al. [Win+23] provide valuable insights into neutral atom-based quantum systems, which utilize neutral atoms as qubits by exploiting their internal quantum states for computation. These systems typically employ optical tweezers to

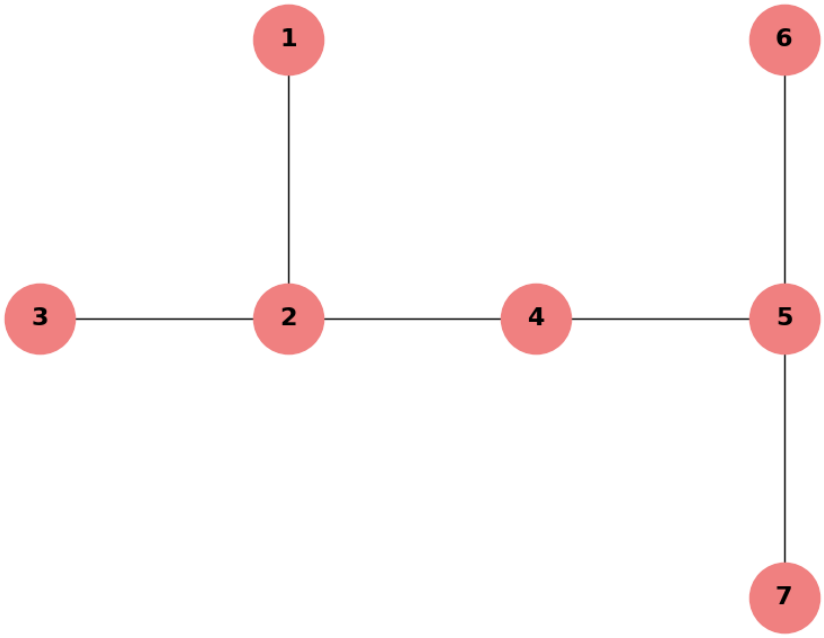

Figure 2.1.: Possible Coupling Map of a 7-qubit Quantum Computer

trap and manipulate individual atoms, enabling precise control over their interactions and states. A key advantage of neutral atom systems compared to other quantum technologies, such as superconducting qubits or trapped ions, lies in their scalability and connectivity. Neutral atoms can be arranged in large arrays, potentially integrating thousands of qubits into a single device, with relatively low cross-talk due to the greater distances between them. Furthermore, neutral atom systems often exhibit longer coherence times, which is crucial for maintaining quantum information integrity. In contrast, superconducting qubits, while powerful, require complex microwave control and are more susceptible to noise. Trapped ions, although offering high fidelity, face scalability challenges due to their reliance on ion traps and laser systems.

The concept of a coupling map does not apply to neutral atom-based quantum computers like it does to superconducting qubits. In neutral atom systems, qubits occupy fixed atom sites, and atoms can be dynamically rearranged using mobile optical traps, known as shuttling of atoms. Shuttling allows for rearranging neutral atom qubits during computation, enabling dynamic, non-local entanglement. Initially, qubits are arranged in pairs with small relative distances, and a global controlled-Z (CZ) gate is applied to entangle these pairs. After entanglement, the pairs can be rearranged using mobile traps, allowing previously entangled atoms to interact with other atoms. This shuttling process, performed at a controlled speed to maintain qubit coherence, can transport information over a spatial range of about 2000 qubits. This capability enhances the flexibility and connectivity of the quantum computing system, enabling more complex operations and interactions between qubits.

Haffner et al. [HRB08] describe the fundamentals of trapped ion-based quantum computing. These systems utilize ions confined in electromagnetic fields as qubits, leveraging their long coherence times and precise control through laser manipulation. Trapped ions excel in achieving high-fidelity quantum gates and entanglement, making them particularly suitable for implementing complex quantum algorithms and error correction protocols, in contrast to other technologies.

In addition to the three leading quantum computing technologies discussed, several others exist. Some of these include photonic qubits, which use particles of light (photons) to represent quantum information, and topological qubits, which are based on manipulating quasi-particles called anyons in two-dimensional materials [LP17]. Unlike conventional qubits, which are susceptible to decoherence and noise, topological qubits leverage the unique properties of anyons. These exotic quasi-particles exhibit non-Abelian statistics, meaning the information is stored in the global properties of the system rather than in local states, providing a natural form of error protection. In the scope of this thesis, we focus on the three primary technologies—superconducting qubits, neutral atom-based systems, and trapped ions—and their representation in sys-sage.

2.1.3. Quantum Gates

Quantum gates are the fundamental units of quantum circuits. They have unique properties that set them apart from the classical logic gates. To perform computations, quantum gates manipulate the states of qubits. Unlike classical gates, they are reversible, i.e., their operations can be undone without loss of information, a fascinating difference from classical gates that intrigues us with the unique properties of quantum computing.

Barenco et al. [Bar+95] explore the quantum gates and complexities of quantum gate implementation, demonstrating how combining one-bit quantum gates with the two-bit exclusive-or gate yields a universal set of gates capable of performing any unitary operation on multiple bits. A detailed analysis of how these gates can be utilized to construct versatile quantum computational networks is provided.

Types of Quantum Gates

- **1-qubit gates**

  1-Qubit gates manipulate the state of a single qubit. Some of the common 1-qubit gates include:

  - **Pauli-X Gate (X):** Flips the qubit's state (analogous to the NOT gate in classical computing).

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

  - **Pauli-Y Gate (Y):** Rotates the qubit state around the Y-axis of the Bloch sphere.

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

  - **Pauli-Z Gate (Z):** Rotates the qubit state around the Z-axis of the Bloch sphere.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

  - **Hadamard Gate (H):** Creates a superposition state from a basis state, enabling the qubit to be in both $|0\rangle$ and $|1\rangle$ simultaneously.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

- **2-qubit gates** 2-Qubit gates manipulate the states of two qubits simultaneously and can create entanglement between them. Common 2-qubit gates include:
  - **Controlled NOT Gate (CNOT):** Flips the state of the target qubit if the control qubit is in the $|1\rangle$ state.

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

  - **Controlled-Z Gate (CZ):** Applies a Pauli-Z operation on the target qubit if the control qubit is in the $|1\rangle$ state.

$$\text{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

  - **SWAP Gate:** Exchanges the states of two qubits.

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Higher-order multi-qubit gates also exist (like Toffoli, Deutsch, etc.), but 1-qubit and 2-qubit gates are sufficient for universal quantum computation, meaning any quantum algorithm can be decomposed into a sequence of these gates.

2.2. High Performance Computing-Quantum Computing (HPC-QC) integration

HPC involves leveraging advanced computational systems and methods to execute intricate calculations and process substantial volumes of data with remarkable speed—tasks that would be infeasible for a single computer to handle. The speed and efficiency of HPC systems are truly impressive, outperforming traditional single-core computing by a significant margin. Unlike traditional single-machine computing, HPC systems introduce additional complexities such as scheduling, communication, and synchronization. Since their inception in the early 1970s, HPC systems have been integral to numerous fields. Extensive efforts have been dedicated to developing versatile, workload-agnostic systems and standardized interfaces to efficiently request and utilize HPC resources [Els+23].

As previously noted, QC offers distinct advantages over classical computing, particularly in specific computational tasks. However, current quantum systems face limitations, such as the restricted number of available qubits and the challenges associated with efficiently managing classical data within quantum environments. Consequently, a practical approach to leveraging quantum computing involves treating Quantum Processing Unit (QPU)s as specialized accelerators for particular tasks, akin to using GPUs and FPGAs for specific applications. This paradigm offloads certain computations to QPUs while integrating HPC resources as needed. Many promising algorithms in the near term rely on a hybrid model that combines classical and quantum information processing. Effective execution of these hybrid algorithms necessitates a seamless integration of classical and quantum resources, ensuring that data can be efficiently transferred and processed across both systems.

Seitz et al. [Sei+23] explore the creation of an advanced toolchain to effectively bridge HPC and QC. They address the complexities of achieving this integration, emphasizing the need for seamless interoperability between classical and quantum systems. Key challenges include developing a robust software infrastructure that can handle the intricacies of both computing paradigms. Additionally, the varied landscape of quantum hardware presents difficulties in executing hybrid algorithms, requiring hardware-agnostic and hardware-specific solutions to support diverse technologies. Furthermore, the rapidly advancing field of quantum technology necessitates that the integration toolchain be flexible and forward-compatible, accommodating new algorithms and hardware innovations as they become available.

Humble et al. [Hum+21] examine the integration of QC with HPC systems, highlighting how quantum computers could significantly enhance computational capabilities for tackling complex problems. While substantial technical challenges remain—such as developing cryogenic controls for low-latency communication and improving device metrics like gate fidelity and coherence times—the potential for successful integration is encouraging. Various macro-architectures for hybrid computing are also discussed, including distributed quantum computing systems that could facilitate entangling

operations across different computational tasks.

2.3. Munich Quantum Software Stack (MQSS)

Schulz et al. [Sch+23] present the development of a comprehensive software stack to integrate quantum computing systems into production environments, especially as accelerators within HPC frameworks. The authors emphasize the importance of a unified software environment to expand the user base beyond quantum physicists, including domain experts from various fields. This involves providing higher-level programming abstractions and extensive design automation tools. MQSS supports seamless access to quantum resources, enables hybrid programming approaches, and facilitates implicit quantum compilation and optimization. This initiative aims to create a transparent workflow for users, addressing the complexities of quantum-classical hybrid systems. Developed as part of the Munich Quantum Valley, this project collaborates with the Leibniz Supercomputing Centre.

Figure 2.2 shows a higher-level overview of MQSS. As shown in the figure, MQSS aims to enable user communities from different backgrounds to compute on quantum devices by providing a unified software stack for managing quantum accelerators and submitting and compiling quantum circuits for different applications.

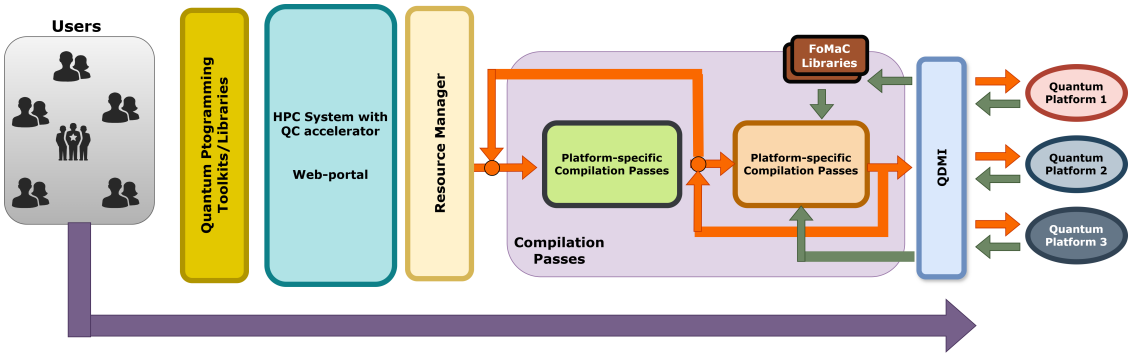

Figure 2.2.: Munich Quantum Software Stack

The scope of work during this thesis aims to support MQSS in the smooth integration of QCs with HPC systems by creating a unified interface for retrieving the topologies of different quantum computing systems and providing them in technology-agnostic fashion. The compatibility of sys-sage with MQSS components such as Quantum Device Management Interface (QDMI), compiler passes, and Figures of Merits and Constraints (FoMaC) has also been investigated.

Finally, Wille et al. [WB23] introduce the MQT QMAP tool, a Munich Quantum Toolkit (MQT) component designed to address the quantum circuit mapping problem—a fundamental challenge in quantum computing. QMAP, with its efficient, automated methods, provides a reliable solution for mapping quantum circuits to various quantum computing architectures. The importance of this mapping process is highlighted, and

comparisons are made to compile high-level code into low-level instructions for classical processors.

QMAP is also integrated within the MQSS framework. To map quantum circuits, QMAP retrieves the properties of quantum backends and stores their topology in the C++ class `Architecture`. This class can import backend properties from multiple sources, including files, strings, and QDMI (discussed later). Although the `Architecture` class is an excellent implementation for QMAP's internal representation, it is not universally compatible with other tools. This thesis also explores the possibility of integrating sys-sage with QMAP to streamline the process of retrieving the topology of quantum systems.

3. Quantum Device Management Interface (QDMI)

This chapter delves into the Quantum Device Management Interface (QDMI), a pivotal component within MQSS. The primary objective of QDMI is to provide a standardized and simplified interface for retrieving the dynamic physical properties and constraints of various quantum platforms. Section 3.1 describes the objectives and core structure of QDMI.

QDMI comprises several integral components, each serving a distinct function. Section 3.2 delves into the core components that form the basis of the QDMI architecture, detailing their roles and functionalities within the system.

In Section 3.3, we propose the implementation of QDMI suitable for IBM's quantum backends. This section details the API calls used to interact with IBM devices, demonstrating the process of querying device properties and managing device states.

Section 3.4 focuses on the testing methodologies employed, particularly the OpenClose tests and the integration of IBM's backend with the Quantum Resource Manager (QRM).

3.1. What is Quantum Device Management Interface (QDMI)?

The Quantum Device Management Interface (QDMI) is a pivotal element within the MQSS, an advanced software infrastructure designed to foster collaboration between users and a diverse array of quantum backends. QDMI's role is crucial in enabling software tools to dynamically retrieve and adapt to the evolving physical characteristics and constraints inherent to various quantum platforms. By mediating between the often divergent interests of software and hardware developers, QDMI provides essential figures of merit and constraints that must be considered. This interface thus emerges as the preferred method for incorporating new platforms or software tools into the MQSS ecosystem, inviting all stakeholders to contribute to its evolution.

QDMI is defined as a C header file, offering a flexible and swift integration into HPC environments. The MQSS seeks to offer a standardized implementation of QDMI, serving as a template. This empowers hardware vendors to integrate their quantum computers with the MQSS by developing customized versions of QDMI based on this template. QDMI consists of 4 types of standard API calls (See section 3.2.3) that will be common to all the implementations. The consistency of API calls across different backends allows other MQSS components, such as the FoMaC library, pass selector, and platform selector to seamlessly integrate with any given backend, irrespective of the

specific internal implementations of QDMI, giving you full control over the integration process (See Chapter 5).

Figure 3.1 shows different components of MQSS and their role in the compilation of quantum circuits. QDMI is placed very close to different Quantum Platforms and serves as an interface to these backends. So, any other tool that requires to access any platform-specific information can do so by making calls to their QDMI implementation.

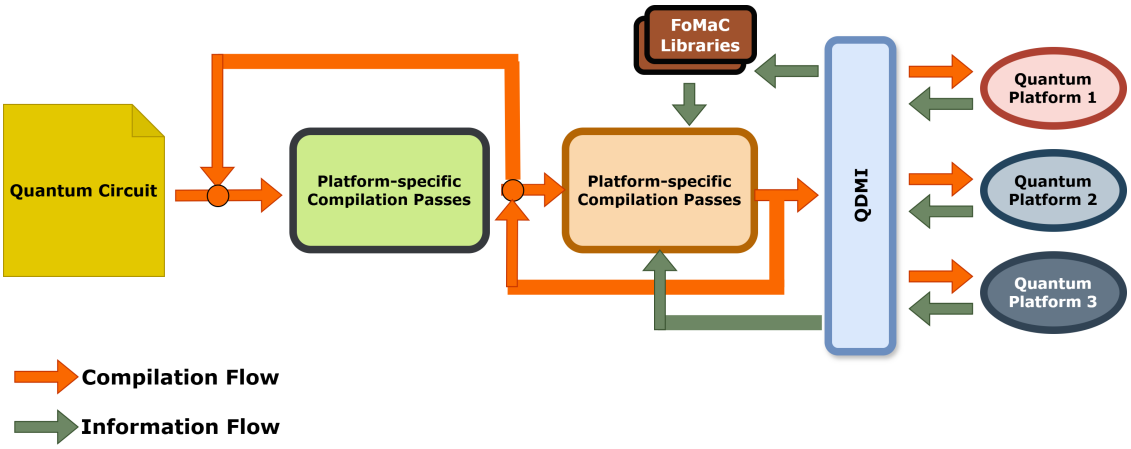

Figure 3.1.: Compilation flow of Quantum Circuits

In the context of this thesis, a practical implementation of QDMI has been developed, utilizing IBM's dummy quantum backend, Athens[4], as a reference model. This implementation serves as a foundational example, demonstrating the process and feasibility of integrating quantum computing platforms within the MQSS through the standardized QDMI implementation. It demonstrates how QDMI can be used in real-world scenarios, providing a clear understanding of its practical application.

3.2. Core Architecture of QDMI

3.2.1. Fundamental Data Types and Definitions in QDMI

QDMI utilizes a variety of core data types and definitions to facilitate the management and interaction with quantum devices, qubits, and quantum gates. This section delves into these fundamental components, explaining their structure, purpose, and significance. These components, data types, and definitions form the basis of all the QDMI calls.

QInfo Objects

The QInfo library offers a versatile and user-friendly mechanism for passing supplementary information to the routines within MQSS. Modeled after the `MPI_Info` object from

[4] `https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.fake_provider.FakeAthens`

the Message Passing Interface (MPI) standard, a QInfo object is a flexible container for additional metadata related to quantum devices or library routines.

At its core, a QInfo object is designed to hold an unordered collection of key-value pairs, where each key is represented as a string, and the corresponding values can be integers, floating-point numbers, or strings. This design allows users to embed a wide range of auxiliary information directly into the library routines, such as configuration hints, device-specific parameters, or runtime options.

QDMI Core Structures

The QDMI core structures, which serve as the foundational elements for representing devices, qubits, and gates within QDMI, were heavily updated and extended during this thesis. These enhancements aimed to provide a more comprehensive and flexible representation of quantum hardware components. The updates included introducing new members to the structures, such as additional properties for gates and a new structure for qubits. These changes enable more detailed and accurate modeling of quantum devices, thereby improving the overall functionality and usability of the QDMI.

- **QDMI_Device**

  The `QDMI_Device` structure represents a quantum device, encapsulating the library, session, and device state information. It is foundational for managing a quantum device's state and interactions. Listing 3.1 shows the definition of `QDMI_Device`.

```
typedef struct QDMI_Device_impl_d {
    QInfo   libinfo;
    QInfo   sessioninfo;
    QDMI_Library_impl_t    library;
    void    *device_state;
} QDMI_Device_impl_t;
typedef struct QDMI_Device_impl_d *QDMI_Device;
```

Listing 3.1: Definition of QDMI_Device

- **QDMI_Gate**

  As shown in Listing 3.2, `QDMI_Gate` structure encapsulates the properties of a quantum gate, including its name, coupling mapping, unitary matrix representation, fidelity, and size. This structure is crucial for managing gate operations and their characteristics.

  Initially, the `QDMI_Gate` structure included only the properties `name`, `coupling_mapping`, and `size_coupling_mapping`. However, it was realized that most of the quantum backends also provide additional information necessary for creating the topology and performing target-specific compilation passes. Subsequently, all the members, including `fidelity, unitary, and gate_size`, were added during the thesis to encompass additional information, providing a more comprehensive representation of a quantum gate's attributes.

```c
typedef struct QDMI_Gate_impl_d {
    const char *name;
    QDMI_Gate_property** coupling_mapping;
    char *unitary;
    double fidelity;
    size_t size_coupling_map;
    size_t gate_size;
} QDMI_Gate_impl_t;
typedef struct QDMI_Gate_impl_d *QDMI_Gate;
```

Listing 3.2: Definition of QDMI_Gate

- **QDMI_Qubit**

  The `QDMI_Qubit` structure, introduced as part of this thesis, is a comprehensive representation of an individual qubit within a quantum device. This data type is meticulously designed to encompass all the necessary properties of a qubit. Using a distinct data type for qubits ensures the organized storage and retrieval of related properties in a consolidated manner. As illustrated in Listing 3.3, the `QDMI_Qubit` structure includes properties such as the qubit's index, coupling mapping, and various physical characteristics like decoherence times(`t1` and `t2`), `readout_error`, and `readout_length`.

```c
typedef struct QDMI_Qubit_impl_d {
    QDMI_qubit_index index;
    QDMI_qubit_index* coupling_mapping;
    int size_coupling_mapping;
    double t1;
    double t2;
    double readout_error;
    double readout_length;
} QDMI_Qubit_impl_t;
typedef struct QDMI_Qubit_impl_d *QDMI_Qubit;
```

Listing 3.3: Definition of QDMI_Qubit

QDMI Property Structures

The properties of devices, qubits, and gates are now represented as structures with `name` and `type` members. Previously, these properties were defined as `typedef` to `int` values (e.g., `typedef int QDMI_Device_property`). This approach was changed during the thesis to store the data type of the properties. For example, hardware vendor *A* provides `backend_version` as a `string` value while vendor *B* provides the same property as a `double` value. Having an extra attribute to store the type of property is beneficial for both the vendor as well as the user.

This enhancement ensures that quantum hardware vendors have greater flexibility in defining the data types of properties. It also ensures that users can easily and safely access and interpret these properties by including the data type in the property

definition, regardless of the vendor's implementation. This user-centric design makes the system more accessible and user-friendly.

- **QDMI_Device_property**

  As shown in Listing 3.4, `QDMI_Device_property`, structure defines properties specific to a quantum device, such as its name and type. This abstraction allows for the flexible definition and retrieval of device attributes.

```
typedef struct QDMI_Device_property_impl_d {
    int name;  // e.g., 15 for backend_name
    int type;  // INT_PROPERTY, etc.
} QDMI_Device_property_impl_t;
typedef struct QDMI_Device_property_impl_d *QDMI_Device_property;
```

Listing 3.4: Definition of QDMI_Device_property

- **QDMI_Gate_property**

  As shown in Listing 3.5, the `QDMI_Gate_property` structure defines attributes for a quantum gate, aiding in the efficient representation of quantum-gate-related information.

```
typedef struct QDMI_Gate_property_impl_d {
    int name;
    int type;
} QDMI_Gate_property_impl_t;
typedef struct QDMI_Gate_property_impl_d *QDMI_Gate_property;
```

Listing 3.5: Definition of QDMI_Gate_property

- **QDMI_Qubit_property**

  Similar to `QDMI_Device_property` and `QDMI_Gate_property`, the `QDMI_Qubit_property` structure defines attributes for a qubit, aiding in the organized management of qubit-specific information.

```
typedef struct QDMI_Qubit_property_impl_d {
    int name;
    int type;
} QDMI_Qubit_property_impl_t;
typedef struct QDMI_Qubit_property_impl_d *QDMI_Qubit_property;
```

Listing 3.6: Definition of QDMI_Qubit_property

3.2.2. QDMI Constants and Macros

QDMI utilizes several constants and macros to define the behavior and types of properties, as well as the status values for various operations. Some of these macros are as shown in Listing 3.7:

```
// Property Behavior
#define QDMI_PROPERTY_NOTEXIST -1
#define QDMI_PROPERTY_STATIC   1
#define QDMI_PROPERTY_DYNAMIC  2

// Property Types
#define QDMI_INT     0
#define QDMI_FLOAT   1
#define QDMI_DOUBLE  2
// and so on...

// Backend Properties
#define BACKEND_NAME 0
#define BACKEND_VERSION 1
#define NUM_QUBITS 2
// and so on...

// Qubit and Gate Properties
#define QDMI_FIDELITY         0
#define QDMI_QUBIT_COUPLING_MAP 6
#define QDMI_T1_TIME          1
#define QDMI_T2_TIME          2
// and so on...
```

Listing 3.7: Definition of QDMI_Qubit_property

3.2.3. QDMI Calls

The core design of QDMI consists of 4 kinds of function calls.

QDMI Core

These calls provide core functionality for managing sessions and opening and closing connections to devices. Some of these calls include:

- `QDMI_core_version(...)`
- `QDMI_core_device_count(...)`
- `QDMI_core_open_device(...)`
- `QDMI_core_query_device(...)`
- `QDMI_core_close_device(...)`
- `QDMI_session_init(...)`
- `QDMI_session_finalize(...)`

Currently, the concrete implementation and documentation for these QDMI calls are not yet available. Moreover, since IBM's Athens serves as a dummy backend, many of these calls are not crucial. However, to ensure consistency and enable thorough testing, a minimalistic implementation of these calls has been provided. Details can be found in Section 3.3.1.

QDMI Control

These calls enable the control of the quantum device. Users can submit quantum circuits, control the job queue, and read measurement results. Some of these calls include:

- `QDMI_control_pack_qasm2(...)`
- `QDMI_control_pack_qir(...)`
- `QDMI_control_submit(...)`
- `QDMI_control_readout_hist_size(...)`
- `QDMI_control_readout_hist_top(...)`
- `QDMI_control_readout_raw_num(...)`
- `QDMI_control_readout_raw_sample(...)`

Currently, the concrete implementation and documentation for these QDMI calls are not yet available. Moreover, since IBM's Athens serves as a dummy backend, many of these calls are not crucial. However, to ensure consistency and enable thorough testing, a minimalistic implementation of these calls has been provided. Details can be found in Section 3.3.1.

QDMI Device

These calls provide functionality for handling devices, such as initiating calibration or checking the device status. Some of these calls include:

- `QDMI_device_status(...)`
- `QDMI_device_quality_check(...)`
- `QDMI_device_quality_limit(...)`
- `QDMI_device_quality_calibrate(...)`

Currently, the concrete implementation and documentation for these QDMI calls are not yet available. Moreover, since IBM's Athens serves as a dummy backend, many of these calls are not crucial. However, to ensure consistency and enable thorough testing, a minimalistic implementation of these calls has been provided. Details can be found in Section 3.3.1.

QDMI Query

These calls allow querying properties of the device, such as supported gates, error rates, gate duration, etc. These calls are most relevant for extracting the topology and mapping the quantum circuits to the physical configuration of the quantum computer. Section 3.3.2 discusses in detail the role and purpose of these calls. Some of these calls include:

- `QDMI_query_device_property_exists(device, property, ...)`
- `QDMI_query_device_property_type(device, property, ...)`
- `QDMI_query_device_property(device, property, ...)`
- `QDMI_query_gateset_num(device, ...)`
- `QDMI_query_all_gates(device, ...)`
- `QDMI_query_byname(device, ...)`
- `QDMI_query_gate_property_exists(device, gate, property, ...)`
- `QDMI_query_gate_property_type(device, gate, property, ...)`
- `QDMI_query_gate_property(device, gate, property, ...)`
- `QDMI_query_all_qubits(device, ...)`
- `QDMI_query_qubit_property_exists(device, qubit, property, ...)`
- `QDMI_query_qubit_property_type(device, qubit, property, ...)`
- `QDMI_query_qubit_property(device, qubit, property, ...)`

3.3. IBM Implementation

This section delves into the crucial design and architectural details of the QDMI implementation specifically tailored for IBM's quantum backend. This implementation enables efficient querying and retrieval of the physical properties of IBM's dummy backends, including IBM Athens.

IBM provides JSON files for its dummy backends, which include comprehensive information about their physical properties [McK+18]. These properties encompass both static and dynamic aspects. Some of these properties include:

- *backend_version*: The specific backend version indicates updates or changes in the hardware or software.
- *n_qubits*: The total count of qubits available on the backend is critical for determining the scale of quantum computations possible.

- *basis_gates*: A list of the available gates on the backend as an array of gate names.
- *coupling_map*: The representation of the physical coupling map on the device.
- *gate_error*: This is the measure of the fidelity of a gate (1 - F_avg, where F_avg is the average gate fidelity). The fidelity of a quantum gate measures how accurately the gate performs its intended quantum operation, serving as a critical metric for assessing gate quality in quantum computing. High fidelity, close to 1, indicates that the gate operation closely matches the ideal, minimizing computational errors.
- *Decoherence Times (T1, T2)*: These refer to the timescales over which qubits maintain their quantum states before decoherence occurs, affecting the reliability and duration of quantum computations.

The QDMI implementation is designed to efficiently query these properties, providing them easily retrievable. For instance, a potential user could use the QDMI to quickly check these properties to map the quantum circuit or optimize their algorithms.

3.3.1. Implementation Details

The implementation uses Jansson Json reader[5] to parse the JSON files and provide all the information. Jansson is recognized as a highly suitable library for JSON file parsing, attributed to its simplicity and user-friendly interface. It boasts an intuitive API that significantly reduces the initial learning curve. Jansson is designed efficiently for time and memory consumption, rendering it an exemplary choice for high-performance applications. This library offers extensive functionalities for parsing, generating, and manipulating JSON data. Its comprehensive documentation facilitates a swift and effective comprehension and utilization of the library. Furthermore, Jansson incorporates advanced error-handling mechanisms that effectively manage parsing errors.

These features encouraged us to use Jansson to parse the JSON files provided by IBM.

The interface of QDMI does not store any information itself because it was designed as a lightweight implementation. This design choice ensures minimal resource usage and high performance. Instead, all JSON data is stored within the JSON root. The JSON root dynamically provides the requested information when queried, acting as a central repository for all data. This approach allows QDMI to operate efficiently, retrieving and processing data on demand without the overhead of maintaining persistent states within the interface.

The base version of QDMI is available in the main branch of the QDMI repository. This version was the foundation for further development, particularly in integrating with IBM's backend.

The extension of QDMI involves implementing backend-specific calls, which are provided as shared libraries. The user loads these shared libraries at runtime, allowing for a flexible and modular architecture. The loading process is straightforward, enabling

[5] `https://github.com/akheron/jansson`

QDMI to interface seamlessly with a diverse range of hardware backends. Each hardware vendor can develop their version of QDMI, tailored to their specific hardware capabilities and features. Users can then link to these vendor-specific libraries, enabling QDMI to seamlessly interface with a diverse range of hardware backends.

QDMI's implementation supports this modularity by allowing backends to define common and unique properties. This flexibility ensures that while there is a consistent interface for general functionalities, hardware vendors can extend the interface to expose specialized features unique to their hardware. This design facilitates interoperability and extensibility, making QDMI a versatile tool for various backend integrations.

As discussed in Section 3.2.3, QDMI consists of four core API calls: `QDMI Core, QDMI Control, QDMI Device, and QDMI Query`. These calls collectively enable QDMI to manage and interface with quantum backends. However, IBM's Athens, serving as a dummy backend, is not an actual quantum backend, and only represents one. Consequently, it lacks essential features such as connections, job queues, and quantum compilation capabilities. This limitation makes it impossible to submit quantum circuits or jobs or to perform any measurements.

Given these constraints, implementing the `QDMI Core, QDMI Control, or QDMI Query` calls for IBM's Athens backend is impractical. These calls are designed to handle actual quantum operations and interactions, which do not apply to a dummy backend. Therefore, implementing these calls would serve no functional purpose.

However, for the sake of consistency and thorough testing, a minimalistic implementation of these calls has been put in place. While these implementations do not perform actual operations, they are designed to return valid return values. This approach ensures the QDMI can be rigorously tested and validated, even when interfacing with non-functional backends like IBM's Athens. It allows developers to verify that the interface behaves correctly regarding structure and response without relying on actual quantum backend capabilities.

As shown in Listing 3.8, the function `QDMI_control_submit` provides a minimalistic implementation for testing purposes.

```
1 int QDMI_control_submit(QDMI_Device dev, QDMI_Fragment *frag, int numshots, QInfo info,
      QDMI_Job *job)
2 {
3     printf("   [Backend].............QDMI_control_submit\n");
4     return QDMI_SUCCESS;
5 }
```

Listing 3.8: Minimalistic implementation of QDMI_control_submit

3.3.2. QDMI Query Calls

The `QDMI Query` calls are essential for retrieving various properties and configurations of quantum devices and qubits. These calls allow users to obtain detailed information about the physical and logical configuration of quantum systems. For instance, information on the coupling maps of a qubit can be used to visualize the physical connectivity of

superconducting qubits and perform the circuit mapping accordingly. In the scope of this thesis, we have implemented the query calls for IBM's quantum backends.

Usage of Setters

Setters in QDMI are used to initialize or update the properties of quantum devices and qubits. The setters are helper functions used internally by other `QDMI Query` calls. This is because letting the user set values or update properties is counterintuitive. This has to be the other way around, and the user must only have a read-only interface when querying properties.

For instance, the `QDMI_set_qubit_coupling_map` and `QDMI_set_qubit_properties` functions are responsible for setting up the coupling map and various properties of a qubit, respectively. However, they are called internally by `QDMI_query_qubit_info`, a public function that users can call.

```
void QDMI_set_gate_properties(QDMI_Gate gate)
{
    gate->unitary  = "Unitary_Matrix";
    gate->fidelity = 1.0;
}
```

Listing 3.9: Helper Function for Setting Gate Properties

Querying Properties

There are various functions to query a wide range of quantum devices and qubits properties using `QDMI Query` calls. These properties include qubit properties, gate properties, and device properties. Each query function is designed to retrieve specific information. Further, the name and signature of a function give a clear overview of what that function might be doing. This is consistent across all types of query functions.

There are two categories of functions for querying properties: those that query all properties of a single qubit or gate, or all qubits and gates, and those that query specific properties.

- **Querying all the properties**

  As discussed in Section 3.2.1, `QDMI_Qubit` is a pointer to a C-structure with several members such as `coupling_mapping`, `t1`, `t2`, `readout_error`, and others. Similarly, `QDMI_Gate` is also a pointer to a C-structure that includes members like `unitary`, `fidelity`, `gate_size`, etc.

  As shown in the Listing 3.10, to query all these properties simultaneously, functions such as `QDMI_query_qubit_info` and `QDMI_query_gate_info` can be used. These functions take an instance of `QDMI_Qubit` or `QDMI_Gate` as an output parameter along with their respective indices. Internally, these functions use setters to retrieve and populate the corresponding values from the backend. This approach ensures

that all relevant properties are efficiently retrieved and stored within the respective structures, allowing easy access and querying.

```
int QDMI_query_qubit_info(QDMI_Qubit qubit, int qubit_index)
{
    QDMI_set_qubit_coupling_map(qubit, qubit_index);
    QDMI_set_qubit_properties(qubit);
}
```

Listing 3.10: Getting all the information of a Qubit

Additionally, it is possible to query all qubits and their properties for a single quantum device. As shown in Listing 3.11, functions such as `QDMI_query_all_qubits` and `QDMI_query_all_gates` facilitate this process. These functions are highly convenient for retrieving comprehensive information with a single call. By invoking these methods, users can efficiently gather detailed data on all qubits or gates in the device, streamlining the process of obtaining and managing quantum device properties. This capability is handy for applications that require a complete overview of the device's configuration and performance characteristics, enabling effective diagnostics, optimization, and utilization of quantum resources.

```
int QDMI_query_all_qubits(QDMI_Device dev, QDMI_Qubit *qubits)
{
    int err, num_qubits;

    err = QDMI_query_qubits_num(dev, &num_qubits);

    if (err != QDMI_SUCCESS) {
        printf("QDMI failed to return number of qubits\n");
        return QDMI_WARN_GENERAL;
    }

    *qubits = (QDMI_Qubit)malloc(num_qubits *   sizeof(QDMI_Qubit_impl_t));

    if (*qubits == NULL) {
        printf("Couldn't allocate memory for the qubit  array\n");
        return QDMI_WARN_GENERAL;
    }

    for (int i = 0; i < num_qubits; i++) {
        QDMI_query_qubit_info((*qubits) + i, i);
    }

    printf("Returning available qubits\n");
    return QDMI_SUCCESS;
}
```

Listing 3.11: Getting all the information of all the Qubits

- **Querying specific properties**

It is not always practical or necessary to query all qubits or gates and all their properties simultaneously. Depending on the specific usage and requirements of the user, querying only a particular property may be sufficient. To address this need, three additional query functions have been introduced:`QDMI_query_yyy_property_exists`, `QDMI_query_yyy_property_type`, and `QDMI_query_yyy_property_z`, where *yyy* can be either `qubits` or `gates`, and *z* can be *c*, *i*, *d*, or *f*, as explained:

1. **QDMI_query_yyy_property_exists**: This function checks whether a specific property exists for a given qubit or gate. It helps determine the availability of specific characteristics before attempting to access or manipulate them. This function serves another critical purpose: while there are specific properties that every backend is expected to provide, some properties may only be available on specific backends. For example, only superconducting backends may provide the `coupling_mapping` of qubits. Therefore, it is more practical for a user to check if a property exists (or if a particular backend provides that property) before attempting to retrieve it. This practical approach ensures compatibility and avoids errors when working with different quantum devices. Listing 3.12 shows one of such functions.

```
int QDMI_query_qubit_property_exists(QDMI_Device dev, QDMI_Qubit qubit,
    QDMI_Qubit_property prop, int* scope)
{
    *scope = 1;
    if(prop->name == QDMI_T1_TIME)
    {
        if(qubit->t1 == QDMI_PROPERTY_NOT_DEFINED){
            // If the property isn't defined/retrieved
            return QDMI_PROPERTY_NOTEXIST;
        }
        else{
            // If the property is defined/retrieved
            return QDMI_SUCCESS;
        }
    }
    // Similarly, for other properties...
}
```

Listing 3.12: Checking if a backend provides that property

2. **QDMI_query_yyy_property_type**: This function retrieves the data type of a specified property. Knowing the type (e.g., integer, double, float, or string) is essential for correctly processing and interpreting property values. This function ensures type safety and prevents invalid type casting, thereby maintaining the integrity and reliability of data handling within the system. Listing 3.13 shows one of such functions.

```
int QDMI_query_qubit_property_type(QDMI_Device dev, QDMI_Qubit qubit,
    QDMI_Qubit_property prop)
{
```

```
    if(prop->name == QDMI_T1_TIME)
    {
        prop->type = QDMI_DOUBLE;
    }
    // Similarly for other properties with QDMI_DOUBLE and other data types...
    return QDMI_SUCCESS;
}
```

Listing 3.13: Checking the type of a property

3. **QDMI_query_yyy_property_z**: This function fetches the value of a specified property. The suffix z indicates the expected return type:

- c for character or string values.

- i for integer values.

- d for double-precision floating-point values.

- f for single-precision floating-point values.

Listing 3.14 shows one of such functions.

```
int QDMI_query_qubit_property_d(QDMI_Device dev, QDMI_Qubit qubit,
    QDMI_Qubit_property prop, double *value)
{
    // Ideally should be called after QDMI_query_qubit_property_exists
    if(prop->name == QDMI_T1_TIME)
        *value = qubit->t1;
    else if(prop->name == QDMI_T2_TIME)
        *value = qubit->t2;
    else if(prop->name == QDMI_READOUT_ERROR)
        *value = qubit->readout_error;
    else if(prop->name == QDMI_READOUT_LENGTH)
        *value = qubit->readout_length;

    return QDMI_SUCCESS;
}
```

Listing 3.14: Querying a particular property

These functions provide a flexible, safe, and efficient way to access specific properties of qubits and gates, allowing users to obtain only the necessary information without the overhead of querying all properties. This targeted querying capability is precious in critical performance and resource management scenarios.

3.4. Results

3.4.1. OpenClose

The QDMIOpenClose test is designed to verify the fundamental connectivity and basic functionality of a QDMI device. This test ensures that the QDMI library can successfully initialize a session, interact with the quantum device to retrieve basic information,

and properly finalize the session. The complete code can be seen in Listing A.1. The following code snippet illustrates some of the important details of this test.

```
// Initialization of Structures
QInfo info;
QDMI_Session session = NULL;
QDMI_Library lib;
QDMI_Device device;
int err;

// Session Initialization
err = QDMI_session_init(info, &session);
CHECK_ERR(err, "QDMI_session_init");

// Loading IBM Backend Library
lib =find_library_by_name("/home/diogenes/bin/lib/libackend_ibm.so");

// Querying Device Information
int num_qubits;
err = QDMI_query_qubits_num(device,&num_qubits);
```

Listing 3.15: QDMIOpenClose Test

The test begins by initializing essential structures, including `QInfo`, `QDMI_Session`, `QDMI_Library`, `QDMI_Device`, and `QDMI_Job`. Memory is allocated for the job and device objects. If the allocation fails for either object, an error message is displayed, and the test terminates.

Next, the `QDMI_session_init` function initializes a QDMI session using the `QInfo` object. This function handles internal startup routines and loads the necessary libraries to prepare the session.

Further, the `find_library_by_name` function searches for the IBM backend library. If the library cannot be located, an error message is printed, and the test exits.

Subsequently, the test queries the device to retrieve relevant information, such as the number of qubits available, using `QDMI_query_qubits_num`. It also checks the status of the device with `QDMI_device_status` to ensure it is operational.

Finally, the `QDMI_session_finalize` function closes the active QDMI session, ensuring all resources are properly released. The `QInfo_free` function is called to free the `QInfo` object. The test concludes by freeing the allocated memory for the device object ensuring there are no memory leaks.

The console output of the test is as follows:

```
[Backend].............Initializing IBM via QDMI
[Backend].............IBM query device status
[QDMIOpenClose].............Found 5 qubits.
[QDMIOpenClose].............Closed connection to the device.

 [DEBUG]: Test Finished
```

Listing 3.16: Result of QDMIOpenClose Test

3.4.2. Quantum Resource Manager (QRM) Integration

QRM is a critical component of MQSS, acting as a comprehensive resource manager that interacts with the QDMI to efficiently manage quantum computing resources. QRM's core functionality includes querying available quantum backends, submitting jobs to quantum devices, and integrating components such as pass selection heuristics, generators, schedulers, compiler passes, and submitters. The entry point for QRM to select and apply LLVM passes to a Quantum Circuit described in Quantum Intermediate Representation (QIR) is the `qresourcemanager_d` daemon.

Listing 3.17 shows the integration of `qresourcemanager_d` with IBM's QDMI implementation. When `qresourcemanager_d` runs with IBM's integrated QDMI implementation, it performs several vital operations. The console output provides a detailed log of these processes, showcasing the interaction between QRM and QDMI.

Initially, `qresourcemanager_d` listens on the `queue_manager` and initializes the IBM backend via QDMI, fetching the necessary configuration files. Upon receiving a QuantumTask, the generator runner invokes the appropriate generator, such as `libgenerator_cutter.so`, to generate sub-circuits, which are then passed to the scheduler runner.

```
[qresourcemanager_d]..Listening on queue queue_manager
[Backend].............Initializing IBM via QDMI
[Backend].............Fetching config file
[qresourcemanager_d]..Received a QuantumTask
[FoMaC]...............Available device found: /libbackend_ibm.so
[Scheduler]...........1 available device(s)
 [Backend].............Returning available qubits
[FoMaC]...............Coupling mapping of qubit[0]: { 1 }
[FoMaC]...............Coupling mapping of qubit[1]: { 0 2 }
[FoMaC]...............Coupling mapping of qubit[2]: { 1 3 }
[FoMaC]...............Coupling mapping of qubit[3]: { 2 4 }
[FoMaC]...............Coupling mapping of qubit[4]: { 3 }
...
[Pass Runner].........Applying pass: QirCommuteRxCnotPass
[Pass Runner].........Applying pass:
[Pass]................Reversible gate found: __quantum__qis__cx__body
[Pass]................Reversible gate found: __quantum__qis__x__body
...
[Pass Runner].........Applying pass: QirXGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirYGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirZGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirSToSDaggerPass
[Pass Runner].........Applying pass: QirSDaggerToSPass
[Pass Runner].........Applying pass: QirReverseCnotPass
[Pass]................Reversing Cnot
[Pass Runner].........Applying pass: QirSwapAndCnotReplacementPass
[Pass Runner].........Applying pass: QirFunctionReplacementPass
[Pass]................Function has 'replaceWith' attribute:
    __quantum__qis__cz__body
[Pass]................Function is a replacement          :
    __quantum__qis__cz_to_hcnoth__body
[Pass Runner].........Applying pass: QirReplaceConstantBranchesPass
```

```
[Pass Runner].........Applying pass: QirRemoveNonEntrypointFunctionsPass
[Backend].............QDMI_control_pack_qir
[Backend].............QDMI_control_submit
[qresourcemanager_d]..Adapted QIR sent to the Quantum Daemon
```

Listing 3.17: QDMI Integrated with Quantum Resource Manager

The scheduler runner invokes the round-robin scheduler, identifies available devices, and selects the appropriate target device for the job. In this example, the scheduler finds and chooses the IBM backend. QDMI then returns the available qubits and their coupling mappings, which are crucial for the subsequent steps in the quantum task execution.

Next, the selector runner invokes the manual selector, which returns a list of compiler passes to be applied to the quantum circuit. The pass runner inserts necessary flags and applies a series of passes, each optimizing the QIR in various ways. These passes include gate transformations, gate commutations, redundant gate cancellations, and function replacements. This process generates warnings for unsupported gates, and reversible gates are identified.

Once all the required passes are applied, the QIR is adapted sent for submission. This integration showcases the seamless interaction between QRM and QDMI, enabling efficient management and optimization of quantum tasks on IBM's quantum backend.

The entire results and the complete console output can be seen in Listing A.2.

4. sys-sage

This chapter delves into the core structure and the significant extensions made to the sys-sage library during this thesis. This chapter aims to provide a comprehensive understanding of sys-sage, including its goals, internal representation, and the modifications implemented to accommodate various quantum systems.

Section 4.1 section elaborates on the goals of sys-sage, its core structure, and the internal representation of HPC systems.

In Section 4.2, we explore the extensions made during this thesis. This includes detailed explanations of how QCs, qubits, and quantum gates are represented within sys-sage.

Section 4.3 presents the results of API calls and example usages with the QDMI implementations for the IBM backend.

Finally, Section 4.4 discusses the proposed extensions and considerations for neutral-atom and trapped-ion systems.

4.1. What is sys-sage?

sys-sage [VS24] is a user-side library focused on collecting, storing, and providing arbitrary hardware-relevant information within an HPC system. sys-sage bridges the gap between various applications, tools, and benchmarks that describe the increasingly complex modern HPC systems. It manages both static and dynamic data contexts, available at different stages of an application's lifecycle.

It enables acquiring, storing, updating, and querying all relevant information about the hardware and software configuration of an HPC node/system, including its dynamic hardware topology, current state, capabilities, and other related information. All these elements are logically connected. sys-sage addresses its limitations for modern systems and workloads by building upon and generalizing the concepts of hwloc [Bro+10]. It integrates data from existing sources like hwloc, provides additional dynamic information, and connects different topology-related pieces of information in its internal representation.

4.1.1. Internal Representation

In sys-sage, the HPC system is represented using a comprehensive structure that includes Components, their hierarchical physical structure known as the Component Tree, and Data Paths. This structure allows for a detailed and multifaceted representation of the system's architecture and its dynamic characteristics.

Components are fundamental units within the HPC system, each representing distinct hardware or software elements. The Component Tree captures the hierarchical organization of these Components. This tree structure illustrates the static physical composition of the system, detailing how various Components, such as processors, memory units, and storage devices, are arranged and interconnected.

In addition to the Component Tree, sys-sage utilizes Data Paths to provide a more nuanced view of the system. Data Paths encapsulate additional attributes and relationships between Components that extend beyond the hierarchical structure of the Component Tree. These paths form a Data-path Graph, which captures information orthogonal to the Component Tree. While the Component Tree focuses on the static, physical organization of the system, the Data-path Graph provides insights into dynamic and supplementary relationships, such as performance metrics and communication pathways.

The Component Tree and the Data-path Graph in sys-sage are necessarily not static representations. They are designed to be modifiable at runtime, allowing sys-sage to dynamically adapt to changes within the HPC system. This dynamic adaptability ensures that sys-sage captures the evolving characteristics and performance metrics in real-time, giving users an informed view of the system's current state.

Within sys-sage, Components can describe a range of hardware attributes, providing a granular understanding of the system's hardware characteristics. For instance, they can detail each cache level's capacity, associativity, and hit rate derived from hardware counters. Similarly, Components can represent the number of registers and the current operating frequency of a CPU core, offering a detailed description of the system's hardware characteristics.

Data Paths, conversely, convey information about the relationships and interactions between Components. For example, they can provide data on cache-to-core latency across different cache levels, offering insights into the efficiency and speed of data retrieval. Additionally, Data Paths can illustrate the percentage of cross-NUMA (Non-Uniform Memory Access) accesses for different NUMA regions, highlighting the communication overhead and performance implications of memory access patterns in multi-node systems.

sys-sage offers a robust and flexible tool for modeling and analyzing HPC systems by integrating the Component Tree with the Data-path Graph. This integrated approach provides a comprehensive understanding of the system's physical composition and dynamic behavior, instilling confidence in sys-sage's ability to facilitate effective performance monitoring and optimization.

- **Components and Component Tree:**

  An HPC system is a complex entity comprising multiple physical and logical elements called Components. These components, with their hierarchical relationships, are elegantly captured in the Component Tree.

  This tree structure provides a comfortable and easily navigable representation for the users. As the backbone of sys-sage, the Component Tree seamlessly links

and references all additional static and dynamic information. Inspired by hwloc's method of representing CPU data, sys-sage extends this approach to encompass a more comprehensive array of components, ensuring that all on-node resources, including both hardware and system-related elements, are captured rather than being limited to CPU resources. This comprehensive approach provides a sense of security to our users.

Multiple Component Types, derived from various parts of computer systems, represent specific attributes and functionalities. These Component Types are implemented in a class hierarchy, with each type inheriting basic attributes and API calls from the generic `Component` type. This hierarchical structure ensures that each Component contains essential information such as:

- Its position in the Component Tree, with links to parent and child Components for easy navigation.
- Basic attributes such as `id` and `name`.
- A wildcard map, `attrib`, allows users to add arbitrary information or data. This key-value store uses the key to denote the attribute name and the value to point to the data.

Figure 4.1 illustrates an example of a component tree, where different components are represented in various colors. The tree follows a hierarchical structure, with each component depicted as a parent or a child, showcasing their relationships within the system.

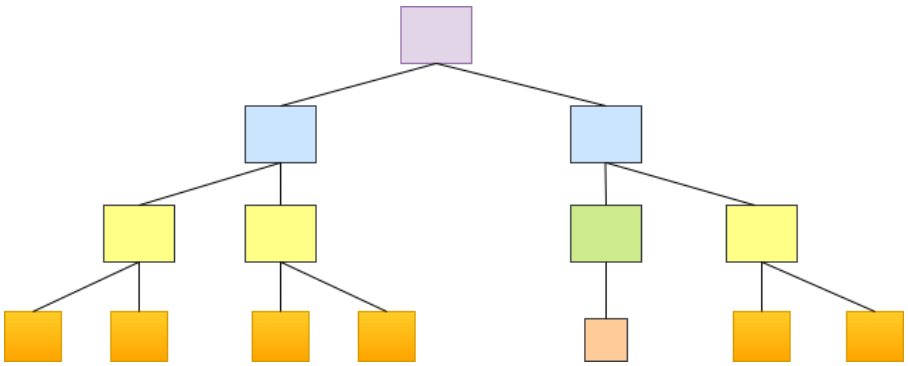

Figure 4.1.: Component Tree showing different Component Types in different colors

- **Data Paths and Data-path Graph**

  A Data Path is a construct that carries dynamic information about the relationship between two arbitrary Components. The aggregation of all Data Paths forms a Data-path Graph. Each Data Path connects a source and a target Component and can be either oriented (differentiating source and target) or bidirectional. Multiple Data Paths can connect the same pair of Components, representing

various dependencies or relationships. The attribute `dp_type` differentiates the types of information conveyed by these Data Paths.

Like Components, Data Paths possess default properties, such as bandwidth and latency, and include a wildcard map `attrib` for storing additional data. This feature enhances the flexibility and adaptability of Data Paths, as it can store any information not covered by the default properties.

Data Paths convey a wide range of information, including but not limited to data transfer, performance, power consumption, or even application-specific data. They can model any property regarding the connection or relationship between two Components.

Data Paths are closely associated with the Components they connect. Each Component maintains references to all Data Paths linked to it, while each Data Path includes references to its source and target Components.

Data Paths provide a highly flexible mechanism for expressing dynamic relationships between Components. This flexibility contrasts with approaches like hwloc, where separate distance matrices represent distance information. These matrices are limited to latency and bandwidth information, are not user-friendly, and lack versatility. Data Paths in sys-sage, however, offer a more comprehensive and adaptable solution for modeling complex relationships in HPC systems.

As shown in Figure 4.2, a data path is depicted with arrows of different colors. An arrow represents an interaction between two components or an exchange of some form of information which may encompass bandwidth and latency, cache partitioning, or data transfer energy.

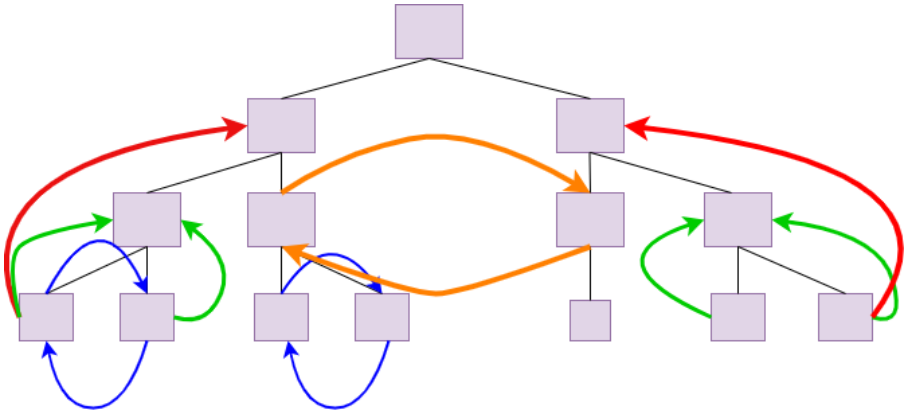

Figure 4.2.: Data-path Graph with Data Paths carrying different information in different colors.

4.1.2. Example Usage

The sys-sage repository contains several examples that were crucial in the early stages of this thesis, illustrating the usage of API calls and the topological representation of HPC systems. These examples provided a clear visualization of the topology of existing HPC systems and helped in the extensions for quantum systems. Here, we demonstrate the "*basic_usage*" example, which demonstrates several existing API calls and presents an HPC system's topology.

sys-sage uses parsers to load information from various sources. One notable API call, *parseHwlocOutput*, is used to parse hwloc output XML files. This function requires two parameters: a pointer to a Node component and the path to the XML file. Once invoked, *parseHwlocOutput* constructs the topology autonomously without needing additional inputs or calls from the user.

```
string topoPath;
// Store path to the XML file in topoPath
...
//create root Topology and one node
Topology* topo = new Topology();
Node* n = new Node(topo, 1);

cout << "-- Parsing Hwloc output from file " << topoPath << endl;
if(parseHwlocOutput(n, topoPath) != 0) { //adds topology to the next node
    usage(argv[0]);
    return 1;
}
```

Listing 4.1: Parsing hwloc output files (XML)

When the call returns successfully (and the topology has been created), various attributes and information can be queried), various attributes and information can be queried.

```
cout << "Total num HW threads: " << topo->GetNumThreads() << endl;
```

Listing 4.2: Querying Number of threads

Finally, the complete topology can be visualized/printed on the console window.

```
cout << "---------------- Printing the whole tree ----------------" << endl;
topo->PrintSubtree();
cout << "----------------                        ----------------" << endl;
```

Listing 4.3: Querying information from the Topology

The complete retrieved topology is as shown in Listing A.3. A snippet of the topology is as follows:

```
-- Parsing Hwloc output from file ./example_data/skylake_hwloc.xml
-- End parseHwlocOutput
Total num HW threads: 24
------------------ Printing the whole tree ------------------
```

```
Topology (name sys_sage Topology) id 0 - children: 1 level: 0
  Node (name Node) id 1 - children: 2 level: 1
    Chip (name socket) id 0 - children: 1 level: 2
      Cache (name Cache) id 7 - children: 2 level: 3
        NUMA (name Numa) id 0 - children: 6 level: 4
          Cache (name Cache) id 6 - children: 1 level: 5
            Cache (name Cache) id 5 - children: 1 level: 6
              Core (name Core) id 0 - children: 1 level: 7
                HW_thread (name HW_thread) id 0 - children: 0 level: 8
          Cache (name Cache) id 11 - children: 1 level: 5
            Cache (name Cache) id 10 - children: 1 level: 6
              Core (name Core) id 1 - children: 1 level: 7
                HW_thread (name HW_thread) id 1 - children: 0 level: 8
          Cache (name Cache) id 15 - children: 1 level: 5
            Cache (name Cache) id 14 - children: 1 level: 6
              Core (name Core) id 2 - children: 1 level: 7
                HW_thread (name HW_thread) id 2 - children: 0 level: 8
          Cache (name Cache) id 19 - children: 1 level: 5
            Cache (name Cache) id 18 - children: 1 level: 6
              Core (name Core) id 8 - children: 1 level: 7
                HW_thread (name HW_thread) id 3 - children: 0 level: 8
          Cache (name Cache) id 23 - children: 1 level: 5
            Cache (name Cache) id 22 - children: 1 level: 6
              Core (name Core) id 9 - children: 1 level: 7
                HW_thread (name HW_thread) id 4 - children: 0 level: 8
          Cache (name Cache) id 27 - children: 1 level: 5
            Cache (name Cache) id 26 - children: 1 level: 6
              Core (name Core) id 10 - children: 1 level: 7
                HW_thread (name HW_thread) id 5 - children: 0 level: 8
        ...
        ...
        ...
_________________                    _________________
```

Listing 4.4: Sample Topology of a HPC system

In a similar fashion, DataPaths can also be queried and visualized.

```
------------------ Printing all DataPaths ------------------
DataPaths regarding Component (NUMA) id 0
    DataPath src: (NUMA) id 0, target: (NUMA) id 0 - bw: 8621, latency: 244
    DataPath src: (NUMA) id 0, target: (NUMA) id 1 - bw: 8237, latency: 203
    DataPath src: (NUMA) id 0, target: (NUMA) id 0 - bw: 8621, latency: 244
    DataPath src: (NUMA) id 1, target: (NUMA) id 0 - bw: 8502, latency: 195
DataPaths regarding Component (NUMA) id 1
    DataPath src: (NUMA) id 1, target: (NUMA) id 0 - bw: 8502, latency: 195
    DataPath src: (NUMA) id 1, target: (NUMA) id 1 - bw: 8663, latency: 237
    DataPath src: (NUMA) id 0, target: (NUMA) id 1 - bw: 8237, latency: 203
    DataPath src: (NUMA) id 1, target: (NUMA) id 1 - bw: 8663, latency: 237
```

Listing 4.5: Visualizing DataPaths Between Two Nodes

4.2. Extension for Superconducting-qubit based systems

The preceding sections of this study have outlined the fundamental architecture of the sys-sage library, which aims for an accurate and user-friendly representation of heterogeneous HPC systems. This thesis intends to build on this foundational work and presents an experimental attempt to encapsulate and elucidate the complex topology inherent in quantum computing systems. This advancement attempts to develop a universal query interface, a feature that allows users to interact with different quantum computing platforms using a single set of commands. This interface offers broad compatibility across a spectrum of quantum computing platforms, including but not limited to systems based on superconducting qubits, neutral atoms, and trapped ion technologies.

sys-sage uses the QDMI library by leveraging `QDMI Query` calls to obtain essential information about quantum systems. Unlike QDMI, which does not have data structures to retain properties or details about the backends, sys-sage addresses this by caching the information and presenting it in a technology-agnostic manner. sys-sage specializes in integrating different sources of partial information in heterogeneous HPC systems, focusing on their dynamic aspects. We envision to extend the same concepts for quantum systems. This functionality can be used by many tools within the MQSS, such as compiler passes, other Figures of Merits and Constraints (FoMaC) libraries, and QMAP, among others (See Chapter 5).

Like its use in HPC systems, sys-sage is adept at collecting and representing static and dynamic information about quantum systems.

1. **Static Information**: This includes immutable details such as the number of qubits in a backend and the coupling map for superconducting qubit-based systems.

2. **Dynamic Information**: This covers variable data such as the coupling maps for neutral atom-based systems and decoherence times, which can change over time or with different operational conditions.

As sys-sage caches the information after retrieving it from QDMI, it is also possible to refresh the properties and get their updated values later.

This section introduces an extension to sys-sage specifically for superconducting-qubit-based QC systems. We focus on superconducting-qubit systems first because, currently, QDMI implementations exist only for these systems. This choice enables thorough testing and integration of the new features. Moreover, this extension serves as a foundational model, facilitating the representation of quantum systems based on other technologies as a natural progression.

As discussed in Section 4.1.1, sys-sage utilizes the concepts of `Components` and `DataPaths` to represent the internal structure of HPC systems. Building upon these foundational concepts, we extend their application to represent quantum systems.

4.2.1. sys-sage Macros

The following macros have been defined to represent quantum backends, qubits, and quantum gate types within sys-sage. These macros are used to categorize and handle different quantum and classical hardware architecture elements.

```
#define SYS_SAGE_COMPONENT_QUANTUM_BACKEND 2048 /**< class QuantumBackend */
#define SYS_SAGE_COMPONENT_QUBIT 4096 /**< class Qubit */

// Size of QuantumGate
#define SYS_SAGE_1Q_QUANTUM_GATE 1 /**< Quantum Gate of size 1-Qubit. */
#define SYS_SAGE_2Q_QUANTUM_GATE 2 /**< Quantum Gate of size 2-Qubits. */
#define SYS_SAGE_MQ_QUANTUM_GATE 4 /**< Quantum Gate of size M-Qubits (where M >2). */
#define SYS_SAGE_NO_TYPE_QUANTUM_GATE 0 /**< Quantum Gate of size 0 or invalid size. */

// Type of QuantumGate
#define SYS_SAGE_QUANTUMGATE_TYPE_ID 32          /**< Identity Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_RZ 64          /**< RZ Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_CNOT 128       /**< CNOT Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_SX 256         /**< SX Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_X 512          /**< X Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_TOFFOLI 1024   /**< Toffoli Gate */
#define SYS_SAGE_QUANTUMGATE_TYPE_UNKNOWN 2048   /**< Unknown Gate */
```

Listing 4.6: sys-sage Macros for quantum Backends, Qubits, and Quantum Gates

4.2.2. Quantum Backend

In the proposed HPC-QC architectures, QCs act as accelerators attached to HPC systems, making them an integral part of the overall computing environment. Despite being components like any other in the system, QCs stand out due to their unique characteristics and operational principles, which differ significantly from traditional computers. Regarding their topology, each quantum computer has a distinct set of physical and logical properties that must be accurately represented to ensure logical storage and retrieval and effective integration with HPC systems.

To address this need and align with the concept of HPC-QC integration, it is natural to consider QCs as a specific type of sys-sage `Component`. Thus, we introduce a new C++ class derived from the `Component` class to represent a quantum system. This new class, called `QuantumBackend`, encapsulates quantum computers' unique attributes and functionalities. Importantly, this approach ensures that the distinctive nature of quantum computing is acknowledged and leveraged while maintaining a strong coherence with the established HPC infrastructure, providing reassurance about the proposed changes.

`QuantumBackend` is declared as follows:

```
class QuantumBackend : public Component
{
// Declaration of members
};
```

Listing 4.7: Declaration of QuantumBackend

The complete declaration of all the members is as shown in listing A.4. Some of the important members of the class are as follows:

Constructors

- `QuantumBackend(Component *parent, int _id = 0, string _name = "QuantumBackend")`
  - **Purpose**: Initializes a `QuantumBackend` object and inserts it into the Component Tree as a child of the specified parent component.
  - **Parameters**:
    - `parent`: The parent `Component` in the Component Tree.
    - `_id`: The unique identifier for the `QuantumBackend`, with a default value of 0.
    - `_name`: The name of the `QuantumBackend`, with a default value of "QuantumBackend".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUANTUM_BACKEND`.
- `QuantumBackend(int _id = 0, string _name = "QuantumBackend")`
  - **Purpose**: Initializes a `QuantumBackend` object without automatically inserting it into the Component Tree.
  - **Parameters**:
    - `_id`: The unique identifier for the `QuantumBackend`, with a default value of 0.
    - `_name`: The name of the `QuantumBackend`, with a default value of "QuantumBackend".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUANTUM_BACKEND`.

Members

- `int GetNumberofQubits() const`
  - **Purpose**: Retrieves the number of qubits in the quantum backend.
  - **Returns**: The number of qubits as an integer.
- `void addGate(QuantumGate *gate)`
  - **Purpose**: Adds a `QuantumGate` to the quantum backend.
  - **Parameters**:
    - `gate`: A pointer to the `QuantumGate` to be added.

- `std::vector<QuantumGate*> GetGatesBySize(size_t _gate_size) const`
  - **Purpose**: Retrieves all gates of a specific size.
  - **Parameters**:
    - `_gate_size`: The size of the gates to be retrieved.
  - **Returns**: A vector of pointers to `QuantumGate` objects of the specified size.
- `std::vector<QuantumGate*> GetAllGateTypes() const`
  - **Purpose**: Retrieves all gates in the quantum backend.
  - **Returns**: A vector of pointers to all `QuantumGate` objects.
- `int GetNumberofGates() const`
  - **Purpose**: Retrieves the total number of gates in the quantum backend.
  - **Returns**: The number of gates as an integer.
- `std::set<std::pair<std::uint16_t, std::uint16_t» GetAllCouplingMaps()`
  - **Purpose**: Retrieves the coupling map for all qubits in the quantum backend.
  - **Returns**: A set of pairs, each representing a coupling between two qubits.
  - **Details**: This function iterates over all qubits, retrieves their coupling maps, and stores the couplings as pairs in a set to ensure uniqueness.
- `QDMI_Device GetQDMIDevice()`
  - **Purpose**: Retrieves the `QDMI_Device` associated with the quantum backend.
  - **Returns**: The `QDMI_Device` object.
- `void RefreshTopology()`
  - **Purpose**: Refreshes the topology of the quantum backend, updating the internal representation of the qubits and gates. This function calls the `RefreshProperties()` method of every child `Qubit` and updates the properties. This is meant to update the dynamic properties when the user requests them.

The complete documentation, including the private members of this class, can be seen in Appendix A.4.

4.2.3. Qubit

Qubits are the fundamental units of quantum information. We introduce a new C++ class derived from the `Component` class to represent qubits accurately. This new C++ class, called `Qubit`, encapsulates the unique attributes and functionalities of qubits. It is beneficial to represent qubits as another type of component as they are hierarchically

associated with quantum computers and form their architecture. At the same time, they have certain unique properties that should be treated and represented differently.

`Qubit` is declared as follows:

```
class Qubit: public Component
{
// Declaration of members
};
```

Listing 4.8: Declaration of Qubit

The complete declaration of all the members is as shown in listing A.5. Some of the important members of the class are as follows:

Constructors

- `Qubit(Component *parent, int _id = 0, string _name = "Qubit")`
  - **Purpose**: Initializes a `Qubit` object and inserts it into the Component Tree as a child of the specified parent component.
  - **Parameters**:
    - `parent`: The parent `Component` in the Component Tree.
    - `_id`: The unique identifier for the `Qubit`, with a default value of 0.
    - `_name`: The name of the `Qubit`, with a default value of "Qubit".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUBIT`.
- `Qubit(int _id = 0, string _name = "Qubit")`
  - **Purpose**: Initializes a `Qubit` object without automatically inserting it into the Component Tree.
  - **Parameters**:
    - `_id`: The unique identifier for the `Qubit`, with a default value of 0.
    - `_name`: The name of the `Qubit`, with a default value of "Qubit".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUBIT`.

Members

- `void SetProperties(double t1, double t2, double readout_error, double readout_length)`
  - **Purpose**: Sets the physical properties of the qubit.
  - **Parameters**:

    * `t1`: The relaxation time of the qubit.
    * `t2`: The dephasing time of the qubit.
    * `readout_error`: The readout error rate of the qubit.
    * `readout_length`: The readout length of the qubit.

- `const std::vector<int>& GetCouplingMapping() const`
  - **Purpose**: Retrieves the coupling mapping of the qubit.
  - **Returns**: A constant reference to a vector representing the coupling mapping.
- `void RefreshProperties()`
  - **Purpose**: Refreshes the properties of the qubit, updating the internal representation of the qubit's properties. This function updates the dynamic properties when the user requests them.
- `std::vector<int> _coupling_mapping`
  - **Purpose**: Stores the coupling mapping for the qubit.
- `std::string _calibration_time`
  - **Purpose**: Stores the last calibration time of the qubit.

Figure 4.3 shows the representation of a single quantum system in sys-sage. The qubits are represented as children of the quantum backend.

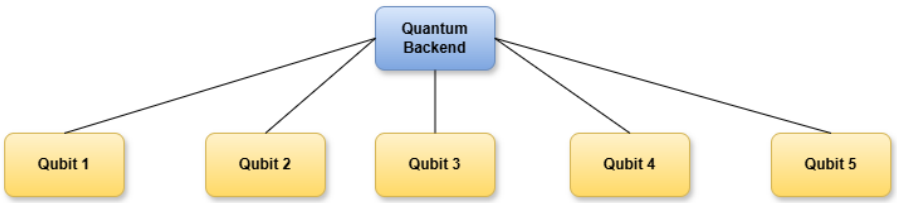

Figure 4.3.: Representation of a Single Quantum System in sys-sage

Similarly, when multiple quantum systems are available, the representation would be as shown in Figure 4.4

4.2.4. Relations and Quantum Gate

To accurately model the interactions between system components, we have introduced a more versatile representation of these interactions through a `Relation`. Unlike previous implementations that primarily focused on `DataPaths` — pathways that represent information transfer between components—our new approach broadens the scope to encompass a wider range of interactions, particularly in quantum systems where the notion of traditional data flow does not always apply. In classical HPC systems, `DataPath`

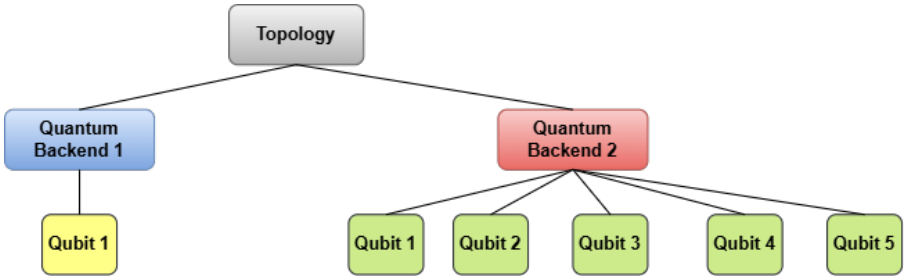

Figure 4.4.: Representation of Multiple Quantum Systems in sys-sage

is a valuable abstraction for representing interactions such as data transfer between processors, memory units, and other components. These pathways are often characterized by metrics like bandwidth, latency, and energy consumption, and they inherently involve a source and a target—however, the dynamics change when we move into the realm of quantum computing. Quantum systems do not always have a clear-cut source and target in the same way classical systems do. Instead, they operate with quantum gates that perform operations on one or multiple qubits, and interactions between qubits are often defined by a coupling map—a representation of which qubits can interact with each other.

To address these nuances, we propose a new C++ class called `Relation`, designed to capture the general concept of interaction between two components, regardless of whether those components are part of an HPC system or a quantum computing system. This class serves as a foundation, with specialized classes derived from it to handle the specific needs of different computing paradigms.

The `Relation` class is not just a solution to the challenges of quantum computing. It is a versatile abstraction designed with flexibility in mind, offering a general interface for setting and retrieving various properties, such as the type and ID of the interaction and the components involved. This broad applicability allows for a unified representation of interactions across different computing domains, facilitating easier integration, simulation, and analysis of hybrid systems that combine both classical and quantum elements.

Listing 4.9 shows the basic structure of `Relation`:

```
class Relation {
public:
    void SetType(int _type);
    int GetType();
    void SetId(int _id);
    int GetId();
    void SetName(std::string _name);
    std::string GetName();
    virtual ~Relation() = default;
    virtual void Print() = 0;
    virtual void DeleteRelation() = 0;
private:
```

```
    int id;
    int type;
    std::string name;
    std::vector <Component *> components;
};
```

Listing 4.9: Declaration of Relation

As shown in Listing 4.10, the `DataPath` class is now derived from `Relation`, preserving its utility for classical systems while allowing it to fit within a broader framework.

```
class DataPath : public Relation
{
// Declaration of members
};
```

Listing 4.10: DataPath as a Child class of Relation

Quantum gates are fundamental operations that manipulate qubits in quantum computing. Each gate has unique properties and operational characteristics that must be accurately represented. A new class called `QuantumGate`, derived from `Relation`, is introduced to represent quantum gates' operations on qubits.

`QuantumGate` is declared as follows:

```
class QuantumGate : public Relation
{
// Declaration of members
};
```

Listing 4.11: Declaration of QuantumGate

Appendix A.6 shows the complete declarations of all the members. Some of the important members of the class are as follows:

Constructors

- `QuantumGate()`
  - **Purpose**: Default constructor for `QuantumGate`.
- `QuantumGate(size_t _gate_size)`
  - **Purpose**: Initializes a `QuantumGate` object with a specified gate size.
  - **Parameters**:
    - `_gate_size`: The number of qubits involved in the quantum gate.

Members

- `void SetGateProperties(std::string _name, double _fidelity, std::string _unitary)`
  - **Purpose**: Sets the properties of the quantum gate.

  - **Parameters**:
    - `_name`: The name of the quantum gate.
    - `_fidelity`: The fidelity of the quantum gate.
    - `_unitary`: The unitary matrix representation of the quantum gate.

- `void SetGateCouplingMap(std::vector<std::vector<Qubit*>> _coupling_mapping)`
  - **Purpose**: Sets the coupling map for the quantum gate.
  - **Parameters**:
    - `_coupling_mapping`: A vector of vectors representing the coupling mapping of qubits.

- `std::vector<std::vector<Qubit*>>coupling_mapping`
  - **Purpose**: Stores the coupling mapping for the quantum gate.

- `std::vector<Component*> qubits`
  - **Purpose**: Stores the qubits involved in the quantum gate.

It is essential to recognize that gate properties can vary depending on the specific qubit pair they act upon. For instance, the fidelity of a CNOT gate operation between qubits 1 and 2 might differ from that of the same CNOT gate operation between qubits 2 and 3. This variability is particularly common in superconducting qubit systems, while systems based on neutral atoms or trapped ions tend to exhibit more consistent properties across all qubit pairs.

The `QuantumGate` class has been meticulously designed to represent both scenarios accurately and efficiently. Figure 4.5 illustrates how different gate types can be visually distinguished by color, especially when gate properties vary between qubit pairs. For example, blue-colored gates might represent 2-qubit CNOT gates, while green-colored gates could represent 1-qubit Identity gates. If the CNOT gate between qubits 1 and 2 has properties different from those of qubits 2 and 3, these gates can be stored separately within the system, each with its own unique ID and name. In this case, the `std::vector<Component*> qubits` will contain the qubits involved in each specific gate operation. This approach is particularly beneficial for handling the intricacies of superconducting qubits.

Conversely, if gate properties remain consistent regardless of the qubits they act on, the `std::vector<Component*> qubits` can remain empty, indicating no specific relation exists between the qubits concerning that quantum gate. This scenario applies more to systems based on neutral atoms or trapped ions, where uniform gate properties are common.

In either situation, the `QuantumBackend` class can store and provide all possible gates based on their size (i.e., the number of qubits involved) and type (e.g., CNOT, RZ).

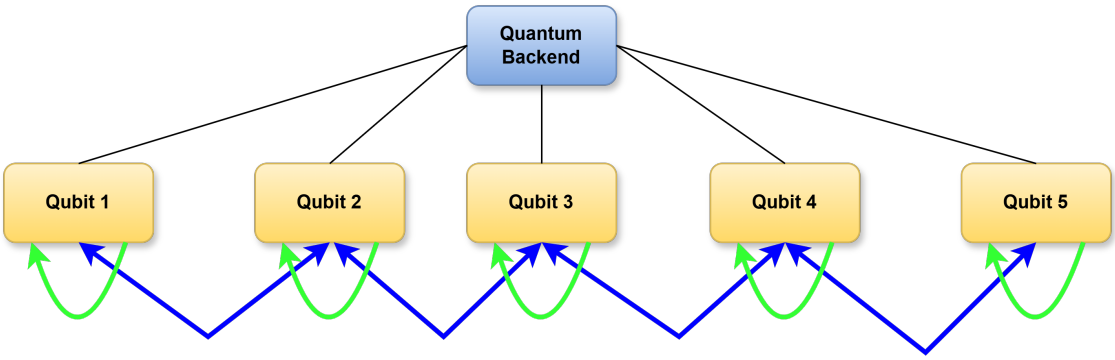

Figure 4.5.: Gates of Different Types in Different Colours

4.2.5. QdmiParser Interface

The `QdmiParser` interface in sys-sage plays a crucial role in facilitating the interaction between users and the QDMI library to create and maintain the topology of a quantum backend. This interface ensures that the internal representation of the quantum system is both accurate and current. The steps involved in this interaction are outlined below:

Topology Creation and Updating Process

1. **Topology Creation Request**:
   - **Step 1**: **User Initiates Topology Creation**:
     - The user begins the process by invoking the `QdmiParser` to create a new quantum backend topology. This is the initial step where the user requests the formation of a structured representation of the quantum backend.
   - **Step 2**: **QdmiParser Communicates with QDMI Implementation**:
     - The `QdmiParser` then communicates with the QDMI implementation to gather necessary information about the quantum devices, qubits, and their connections. This involves setting up a session and invoking QDMI functions to collect data on available hardware components and their configurations.
   - **Step 3**: **Provision of Created Topology**:
     - After successfully gathering and organizing the static information, the `QdmiParser` constructs the topology and provides this information to the user. The static information includes details such as the number of qubits, their initial states, and the available gates.
2. **Updating Dynamic Information**:
   - **Step 4**: **User Requests Updated Information**:

  - At any later point, the user may need to update certain dynamic properties of the quantum backend, such as gate fidelities or qubit coherence times. The user makes a request to `QdmiParser` for the latest values of these properties.

- **Step 5**: **Request for Dynamic Information via QdmiParser**:
  - sys-sage utilizes the `QdmiParser` to send a request to the QDMI library for the required dynamic information. This involves querying the current state of specific properties of the quantum backend components.
- **Step 6**: **QdmiParser Retrieves Updated Information**:
  - The `QdmiParser` communicates with the QDMI library to obtain the latest data on the requested properties. This ensures that the most recent and accurate information is retrieved from the QDMI.
- **Step 7**: **Updating the Internal Representation**:
  - The retrieved dynamic information is used by `QdmiParser` to update the internal representation of the topology within sys-sage. This step ensures that the system's internal model reflects the most current state of the quantum backend.
- **Step 8**: **User Accesses Updated Information**:
  - The user can now access the updated information from sys-sage's internal representation. This allows the user to make decisions or perform operations based on the latest state of the quantum backend, ensuring high accuracy and reliability in quantum computing tasks.

Figure 4.6 summarizes the above steps.

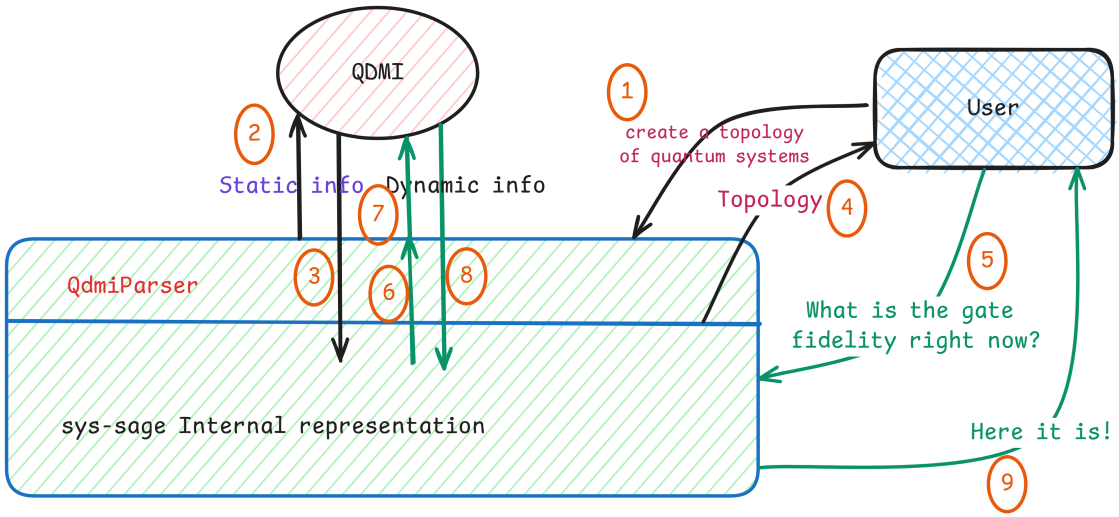

Figure 4.6.: Flow of Information from QDMI to QdmiParser and the creation of topology

The members' complete declaration is as shown in Appendix A.7. Some of the important members of the class are as follows:

Constructor

- `QdmiParser::QdmiParser()`
  - **Purpose**: Initializes the `QdmiParser` object and initiates a QDMI session.

Methods

- **Session Initiation:**
  - `int QdmiParser::initiateSession()`
    - **Purpose**: Initiates a session with the QDMI library.
    - **Returns**: An integer representing the success or failure of the session initiation.
- **Retrieve Available Backends:**
  - `std::vector<QDMI_Device> QdmiParser::get_available_backends()`
    - **Purpose**: Retrieves a list of all available QDMI devices.
    - **Returns**: A vector of `QDMI_Device` objects representing the available devices.
- **Get Qubit Properties:**
  - `void QdmiParser::getQubitProperties(QDMI_Device dev, QDMI_Qubit qubit)`
    - **Purpose**: Retrieves various properties of a specific qubit.
    - **Parameters**:
      - `QDMI_Device dev`: The device containing the qubit.
      - `QDMI_Qubit qubit`: The qubit for which properties are retrieved.
- **Create All Quantum Computer Topologies:**
  - `void QdmiParser::createAllQcTopo(Topology *topo)`
    - **Purpose**: Creates the topology for all available quantum computers.
    - **Parameters**:
      - `Topology *topo`: The topology object to be populated.
  - `Topology QdmiParser::createAllQcTopo()`
    - **Purpose**: Creates and returns the topology for all available quantum computers.
    - **Returns**: A `Topology` object representing all available quantum computers.

- **Create Quantum Computer Topology:**
  - `void QdmiParser::createQcTopo(QuantumBackend *backend, QDMI_Device dev)`
    - **Purpose**: Creates the topology for a specific quantum computer.
    - **Parameters**:
      - `QuantumBackend *backend`: The backend for which the topology is created.
      - `QDMI_Device dev`: The device containing the quantum computer.
  - `QuantumBackend QdmiParser::createQcTopo(QDMI_Device dev, int device_index, std::string device_name)`
    - **Purpose**: Creates and returns the topology for a specific quantum computer.
    - **Parameters**:
      - `QDMI_Device dev`: The device containing the quantum computer.
      - `int device_index`: The index of the device.
      - `std::string device_name`: The name of the device.
    - **Returns**: A `QuantumBackend` object representing the quantum computer.

4.3. Results

This section delves into several key sys-sage API calls that are instrumental in creating and accessing the topology of a Quantum Backend within the sys-sage. Subsection 4.3.1 shows some of these API calls which enable users to establish the initial configuration of the quantum backend and dynamically update its properties as needed. By leveraging these API calls, users can gain detailed insights into the structure and operational parameters of their quantum systems.

Furthermore, the command-line outputs generated by these API calls are showcased in Section 4.3.2. These outputs provide a tangible demonstration of how the APIs function, illustrating the retrieved information and the overall topology configuration process. The examples include the creation of the quantum backend topology, querying qubit properties, and retrieving supported quantum gate types, thereby offering a comprehensive overview of sys-sage's capabilities in managing quantum system topologies.

Finally, 4.3.3 discusses some of the performance and timing analysis of creating and retrieving the topology of qunatum backends using sys-sage.

4.3.1. Example Usage

Using the `QdmiParser`, the topology for all quantum backends can be created. This process involves initializing a `Topology` object and invoking the `createAllQcTopo` method

to populate it with the configuration of the quantum backends.

```
// Create an instance of the interface
QdmiParser qdmi;

// Use QdmiParser to create the topology of either all the backends or one of the
    backends

// Method 1: Topology of all the available Quantum Backends
Topology* qc_topo = new Topology();
qdmi.createAllQcTopo(qc_topo); // or Topology createAllQcTopo();

// Method 2: Query the total available backends and create the topology of one of them
auto quantum_backends = qdmi.get_available_backends();
QuantumBackend* qc = new QuantumBackend(0, "IBM");
qdmi.createQcTopo(qc, quantum_backends[0]); // or QuantumBackend createQcTopo(QDMI_Device
    dev, int device_index = 0, std::string device_name="");
```

Listing 4.12: Creating Quantum Backend Topology

Subsequently, the complete configuration of the backend can be printed, showcasing the hierarchical structure of the quantum system:

```
qc_topo->PrintSubtree();
```

Listing 4.13: Printing the generated topology

When the topology is ready, various properties can be retrieved or queried. For example, the coupling mappings of all the qubits can be retrieved as follows:

```
QuantumBackend* qc = dynamic_cast<QuantumBackend*>(qc_topo->GetChild(0));
int total_qubits = qc->CountAllSubcomponents();
for (int i = 0; i < total_qubits; i++)
{
    Qubit* q = dynamic_cast<Qubit*>(qc->GetChild(i));
    std::cout << "Qubit " << i << " has coupling map { ";
    auto coupling_map = q->GetCouplingMapping();
    for (long unsigned j = 0; j < coupling_map.size(); j++)
    {
        std::cout << coupling_map[j] << " ";
    }
    std::cout << "}\n";
}
```

Listing 4.14: Retrieving the Coupling Mappings of all the qubits

Similarly, qubit properties can be retrieved as such:

```
for (int i = 0; i < total_qubits; i++)
{
    Qubit* q = dynamic_cast<Qubit*>(qc->GetChild(i));
    std::cout << "Qubit " << i << " has following properties: \n";
    std::cout << " T1: " << q->GetT1() << "\n";
    std::cout << " T2: " << q->GetT2() << "\n";
    std::cout << " Readout Error: " << q->GetReadoutError() << "\n";
    std::cout << " Readout Length: " << q->GetReadoutLength() << "\n";
```

```
}
```

Listing 4.15: Retrieving properties of the qubits

Finally, any information related to the supported gate types can also be retrieved as follows:

```
auto _1q_gates = qc->GetGatesBySize(SYS_SAGE_1Q_QUANTUM_GATE);
auto _2q_gates = qc->GetGatesBySize(SYS_SAGE_2Q_QUANTUM_GATE);

if(_1q_gates.size())
{
    int size = _1q_gates.size();
    std::cout << "Total " << size << " 1-Qubit gate(s)\n";
    for (int i = 0; i < size; ++i)
    {
        std::cout << "Gate name:" << _1q_gates[i]->GetName() << ", ";
        std::cout << "Gate size:" << _1q_gates[i]->GetGateSize() << ", ";
        std::cout << "Gate fidelity:" << _1q_gates[i]->GetFidelity() << "\n";
    }
}

// Similarly for other gate sizes...
```

Listing 4.16: Retrieving properties of the supported gate types

4.3.2. Outputs

The retrieved topology of the IBM quantum backend is demonstrated in Listing 4.17. sys-sage successfully interfaces with the QDMI library to display the configuration of the IBM Backend. The topology is presented hierarchically, where the Quantum Backend is depicted as the parent with five qubits as its children. Additionally, the listing showcases the retrieved coupling maps, qubit properties, and gate properties, offering a comprehensive view of the backend's structure and capabilities.

```
[Backend].............Initializing IBM via QDMI
[sys-sage]..............Initiated QDMI session
[sys-sage]..............QDMI_core_device_count returned 1.
[Backend].............Returning available qubits
[sys-sage]..............Found 6 supported gates.
---------- Printing the configuration of IBM Backend ----------
Topology (name sys-sage Topology) id 0 - children: 1 level: 0
  Quantum Backend (name QuantumBackend) id 0 - children: 5 level: 1
    Qubit (name Qubit) id 0 - children: 0 level: 2
    Qubit (name Qubit) id 1 - children: 0 level: 2
    Qubit (name Qubit) id 2 - children: 0 level: 2
    Qubit (name Qubit) id 3 - children: 0 level: 2
    Qubit (name Qubit) id 4 - children: 0 level: 2
---------- Printing Qubit Coupling Mappings for IBM Backend ----------
Qubit 0 has coupling map { 1 }
Qubit 1 has coupling map { 0 2 }
Qubit 2 has coupling map { 1 3 }
```

```
Qubit 3 has coupling map { 2 4 }
Qubit 4 has coupling map { 3 }
---------- Printing Supported Gate Types for IBM Backend ----------
Total 4 1-Qubit gate(s)
Gate name:id, Gate size:1, Gate fidelity:1
Gate name:rz, Gate size:1, Gate fidelity:1
Gate name:sx, Gate size:1, Gate fidelity:1
Gate name:x, Gate size:1, Gate fidelity:1
Total 1 2-Qubit gate(s)
  Gate name:cx, Gate size:2, Gate fidelity:1
Total 1 gate(s) with no type
  Gate name:reset, Gate size:0, Gate fidelity:1
---------- Printing Qubit Properties for IBM Backend ----------
Qubit 0 has following properties:
      T1: 63.4878302170839
      T2: 112.232455355998
      Readout Error: 0.0101
      Readout Length: 3022.22222222222
Qubit 1 has following properties:
      T1: 73.093518301544
      T2: 126.83382141649
      Readout Error: 0.0117
      Readout Length: 3022.22222222222
Qubit 2 has following properties:
      T1: 32.91818023388
      T2: 59.8064388345423
      Readout Error: 0.0243
      Readout Length: 3022.22222222222
Qubit 3 has following properties:
      T1: 49.5108456296743
      T2: 20.5513082711555
      Readout Error: 0.0153
      Readout Length: 3022.22222222222
Qubit 4 has following properties:
      T1: 111.440953419233
      T2: 168.762538283892
      Readout Error: 0.0219
      Readout Length: 3022.22222222222
```

Listing 4.17: Visualizing Topology of IBM's Quantum Backend

4.3.3. Performance Analysis of Topology Generation in sys-Sage

In this section, we analyze the time required to generate the topology of various quantum backends using the sys-sage library. Two of these backends are IBM's Athens (5 qubits)[6] and Cairo (27 qubits)[7]. The remaining backends are hypothetical, created by replicating

[6] https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.fake_provider.FakeAthens

[7] https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.fake_provider.FakeCairo

the configurations of Athens and Cairo to simulate systems with larger numbers of qubits. In total, we examine 7 backends, with qubit counts ranging from 5 to 10,000.

Hotspot analysis and profiling of the application revealed that the majority of the time spent creating the topology is due to retrieving information from QDMI, with sys-sage itself contributing minimal overhead. Three key functions have been identified for performance investigation:

1. **Fetching the information about all qubits of a backend (`setQubits`):** This function retrieves the necessary qubit data from QDMI to facilitate topology creation.

2. **Fetching the information about all gatesets of a backend (`setGateSets`):** This function consolidates all the information retrieved from QDMI to construct the complete topology.

3. **Creating the topology (`createAllqctopo`):** This getter function returns the coupling maps of all qubits. It is a critical function for performance monitoring as its overhead is expected to grow with the number of qubits, making it a significant focus in the analysis.

To accurately measure the relative performance and timing of the functions involved in this study, the `std::chrono` library was employed for high-resolution timing. Listing 4.18 shows the code snippet illustrating the approach used for `createAllQcTopo`.

```
auto start = std::chrono::high_resolution_clock::now();
qdmi.createAllQcTopo(qc_topo);
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> duration = end - start;
std::cout << "Time taken: " << duration.count() << " milliseconds" << std::endl;
```

Listing 4.18: Measuring Runtimes of Critical Functions

Here, `std::chrono::high_resolution_clock` precisely measures the time elapsed during the execution of the function `createAllQcTopo()`. The process involves recording the start time immediately before the function call and the end time immediately after. The difference between these timestamps, calculated using `std::chrono::duration`, represents the duration of the function execution in milliseconds.

To ensure the reliability of the timing data, each function was tested 20 times across different backends. This repeated measurement helps account for variations in execution time due to factors like system load or resource availability. The average, minimum, and maximum timings from these tests are presented in Table 4.1.

The results indicate that the time required to generate the topology (`createAllQcTopo`) increases with the number of qubits in the backend. Topology creation is significantly faster for backends with fewer qubits compared to those with a more significant number of qubits. This increase is expected due to the greater complexity of managing more qubits and their interactions.

`setQubits` function makes multiple QDMI calls to retrieve all necessary qubit information. The timing difference between `setQubits` and `createAllQcTopo` indicates that sys-sage adds only minimal overhead to QDMI during topology generation. This overhead becomes negligible as the number of qubits increases, particularly beyond 500 qubits.

No. of Qubits	createAllQctopo (Runtime in ms.)			setQubits (Runtime in ms.)			GetAllCouplingMaps (Runtime in ms.)		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
5 qubits	0.0956	**0.141**	0.2207	0.0281	**0.048**	0.0817	0.0027	**0.00437**	0.0077
27 qubits	0.2017	**0.4223**	0.7652	0.1082	**0.25**	0.5	0.0097	**0.01829**	0.0356
100 qubits	0.524	**0.725**	1.0856	0.439	**0.60**	0.9109	0.025	**0.039**	0.0785
500 qubits	3.5943	**4.197**	5.0786	3.3698	**3.930**	4.7706	0.1298	**0.1425**	0.1765
1000 qubits	11.0729	**12.379**	14.949	10.7167	**11.953**	14.5571	0.5094	**0.563**	0.7034
5000 qubits	190.01	**196.039**	201.731	188.165	**194.412**	200.043	19.4882	**22.088**	26.9954
10000 qubits	765.443	**836.402**	970.473	760.246	**831.651**	966.85	88.7008	**101.718**	120.75

Table 4.1.: Minimum, Maximum, and Average Runtimes (in ms) for Various Functions on Different Quantum Backends

For retrieving the coupling maps using `GetAllCouplingMaps`, the function takes less than a millisecond to gather information for backends with fewer than 1,000 qubits. However, the retrieval time increases linearly with the number of qubits as the amount of data that needs to be copied and processed grows. Despite this, further optimization could reduce this timing even more.

Figure 4.7 shows the variations in the timings of these functions as line plots. The x-axis represents number of qubits and the y-axis is runtime in milliseconds on log-scale.

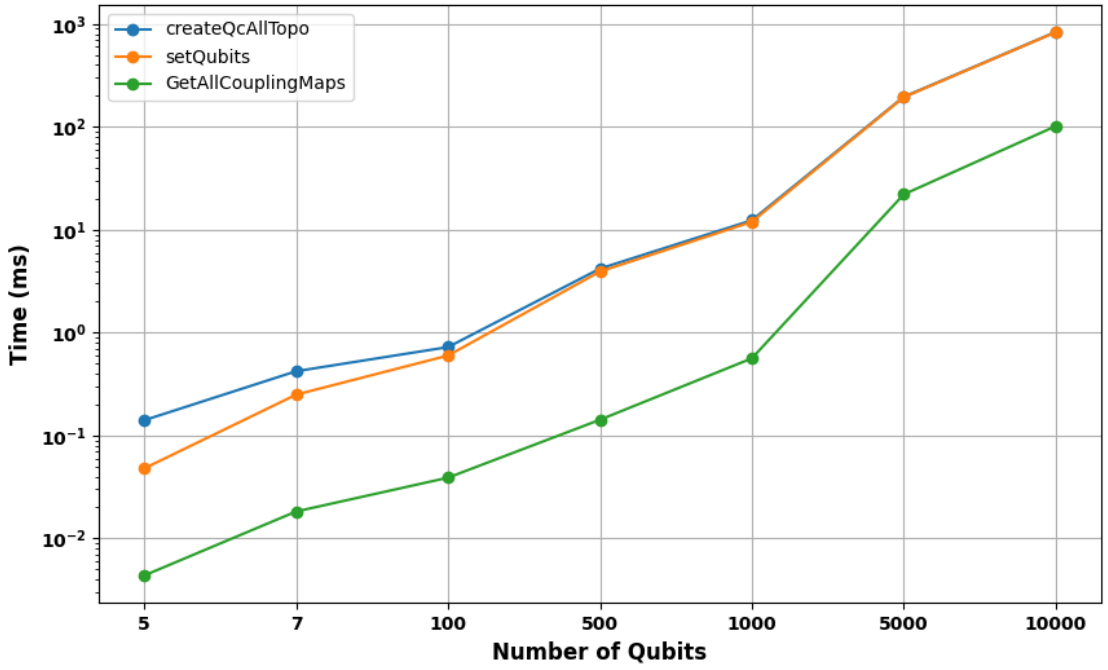

Figure 4.7.: Performance of sys-sage against Number of Qubits

4.4. Representation of Neutral Atoms (NA) and TI systems

Quantum systems based on superconducting qubits differ significantly in their physical characteristics from those based on Neutral Atoms (NA) and Trapped Ions (TI). For superconducting qubit systems, many properties remain static over time, allowing for a stable and fixed representation of qubit connectivity and other parameters. However, several properties can be dynamic in NA and TI systems, changing frequently based on the experimental conditions or specific operations being performed.

In superconducting qubit systems, the concept of coupling mappings is crucial as it defines the physical connectivity between qubits, which is generally fixed. Due to their inherent flexibility, these mappings are less relevant for NA and TI systems. In NA and TI-based quantum systems, qubits are represented by neutral atoms or ions that can be dynamically positioned, trapped, and shuttled according to the requirements of the quantum circuit. This mobility introduces a level of dynamism that is not present in superconducting qubit systems.

To illustrate the flexibility and extensibility of sys-sage, we have developed a new class, `AtomSite`, which is designed to represent quantum backends based on NA and TI technologies. The `AtomSite` class inherits from the `QuantumBackend` class, leveraging the foundational features provided by the parent class while introducing additional properties and mappings specific to NA and TI systems.

```
class AtomSite : public QuantumBackend {
public:
    struct SiteProperties {
    int nRows;
    int nColumns;
    int nAods;
    int nAodIntermediateLevels;
    int nAodCoordinates;
    double interQubitDistance;
    double interactionRadius;
    double blockingFactor;
    } properties;
    std::map<std::string, double> shuttlingTimes;
    std::map<std::string, double> shuttlingAverageFidelities;
};
```

Listing 4.19: Class Definition for AtomSite

Properties Structure

The `SiteProperties` structure encapsulates various characteristics of the atom sites:

- `int nRows`: Number of rows in the atomic array.
- `int nColumns`: Number of columns in the atomic array.
- `int nAods`: Number of acousto-optic deflectors (AODs).

- `int nAodIntermediateLevels`: Number of intermediate levels in AODs.
- `int nAodCoordinates`: Number of coordinates managed by AODs.
- `double interQubitDistance`: Distance between qubits in the array.
- `double interactionRadius`: Radius within which qubits interact.
- `double blockingFactor`: Factor indicating the blocking effect between qubits.

Shuttling Times and Fidelities

In addition to the properties structure, the `AtomSite` class maintains two maps:

- `std::map<std::string, double> shuttlingTimes`: This map stores the times required for shuttling operations between different sites (e.g. moving, aod_moving, etc.).
- `std::map<std::string, double> shuttlingAverageFidelities`: Records the average fidelities of the shuttling operations.

Although the `AtomSite` class has not been tested due to the lack of a corresponding QDMI implementation, it demonstrates that extending sys-sage to support other quantum technologies is a straightforward process. By following the established design patterns, developers can seamlessly integrate additional quantum backend types into the sys-sage, ensuring broad applicability and future-proofing the system for emerging quantum technologies.

5. sys-sage Integration within MQSS

The sys-sage library offers significant advantages in caching data provided by the QDMI library and delivering this data to users via API calls. By acting as an intermediary, sys-sage mitigates the need for libraries to make numerous QDMI calls (typically implemented through C-based functions). Instead, these libraries can utilize a higher-level C++ interface to efficiently fetch all required properties. This approach enhances memory safety, reduces the number of function calls, and results in a cleaner, more maintainable implementation.

In this chapter, we explore three primary use cases of sys-sage within the context of MQSS. These use cases illustrate how sys-sage can be effectively integrated into various components to optimize performance and streamline interactions with quantum backends. The identified use cases are:

1. **Figures of Merits and Constraints (FoMaC)**: The primary objective of the FoMaC is to create an abstraction layer that integrates various quantum technologies. sys-sage is an experimental addition to MQSS that enhances this integration in two ways: as a standalone library within FoMaC, providing a unified interface to manage quantum backend properties, and as a third-party library that streamlines access to quantum hardware information. This dual functionality of sys-sage may improve the efficiency and effectiveness of all the tools that require seamless access to detailed backend properties.

2. **Compiler Passes**: Platform-dependent compiler passes require the physical properties of the quantum backends. Here, sys-sage can efficiently manage and provide quantum backend properties to different compiler passes within MQSS. This integration ensures the compiler can access accurate and up-to-date information, facilitating better optimization and code generation for quantum circuits.

3. **MQT QMAP**: sys-sage is a reliable and consistent aid in integrating with MQT QMAP, offering a reliable and consistent interface for accessing quantum backend properties. This integration supports the mapping of quantum circuits onto physical hardware, ensuring that the mappings are optimized based on the latest backend characteristics.

Each of these use cases will be discussed in detail in the subsequent sections of this chapter. Section 5.1 examines the application of sys-sage within FoMac, focusing on its effectiveness in providing a clean and efficient method for retrieving quantum backend information. Section 5.2 delves into sys-sage use within the passes repository,

highlighting its contributions to enhancing the compilation process. Finally, Section 5.3 explores sys-sage integration with MQT QMAP, showcasing its role in optimizing quantum circuit mappings.

5.1. Figures of Merits and Constraints (FoMaC)

FoMaC retrieves information from quantum devices using the QDMI and computes and caches complex queries to enhance functionality. In this context, sys-sage is an experimental addition to MQSS that can be utilized in two distinct ways.

Firstly, sys-sage can function as a standalone FoMaC library within MQSS. By serving as an independent repository of quantum backend properties, sys-sage can provide a unified interface through which other tools can access and manage detailed information about quantum backends. For instance, this integration may allow the tools to efficiently query and retrieve essential data required for the mapping and scheduling of quantum circuits, thus enhancing the overall efficiency and effectiveness of the mapping process.

Alternatively, sys-sage can be a third-party library that other FoMaC libraries/implementations can leverage. In this capacity, sys-sage facilitates the abstraction of complex backend details, enabling various FoMaC libraries to interact with quantum hardware without dealing directly with the intricacies of QDMI. This approach streamlines the process of obtaining backend properties and performing any post-analysis.

While sys-sage offers significant potential for enhancing FoMaC, it's important to note that its integration may have challenges or limitations. For instance, compatibility issues with existing tools or the need for additional integration resources could be potential hurdles. However, by providing these two flexible modes of integration, sys-sage significantly contributes to the advancement of quantum computing technologies within the MQSS, offering a versatile solution for managing and utilizing quantum backend properties. This experimental work demonstrates the potential of sys-sage to enhance FoMaC's capabilities, whether as a core component or as a supplementary third-party tool.

In this section, we discuss the enhancements made to the 'fomac.cpp' file, part of the FoMaC repository, by replacing traditional QDMI calls with sys-sage calls. This replacement aims to streamline the process of retrieving available quantum devices, making the implementation more efficient and maintainable.

5.1.1. Original Implementation

The original version of the 'fomac.cpp' file uses direct QDMI calls to fetch available quantum devices. The function `FOMAC_available_devices()` is responsible for retrieving these devices and involves multiple steps to interact with the QDMI library, as shown below:

```
1 extern "C" std::vector<QDMI_Device> FOMAC_available_devices()
2 {
3     int count;
```

```
    QDMI_core_device_count(NULL, &count);
    if(count == 0){
        std::cout << "    [FoMaC].............QDMI_core_device_count from QDMI returned
    0." << std::endl;
        return std::vector<QDMI_Device>();
    }

    std::vector<QDMI_Device> registeredDevices;
    QInfo info;
    QInfo_create(&info);

    for(int i = 0; i < count; i++){
        QDMI_Device device;
        QDMI_core_open_device(NULL, i , &info, &device);
        registeredDevices.push_back(device);
    }

    return registeredDevices;
}
```

Listing 5.1: Original Implementation of FOMAC$_{a}$*vailable*$_{d}$*evices*()

In this implementation, the function first retrieves the number of available devices using `QDMI_core_device_count()`. It then iterates through each device, opening it with `QDMI_core_open_device()` and adding it to the `registeredDevices` vector.

5.1.2. Implementation with sys-sage

This version leverages sys-sage to simplify and improve the process of fetching available quantum devices. By using sys-sage, we reduce the number of direct QDMI calls and achieve a cleaner, more maintainable codebase. The updated function is shown below:

```
extern "C" std::vector<QDMI_Device> FOMAC_available_devices()
{
// QdmiParser: sys-sage's interface to qdmi (for retrieving the static topology)
    QdmiParser qdmi;
    const auto quantum_backends = qdmi.get_available_backends(); // Returns a list of all
     the backends
    std::cout << "    [FoMaC].............Total "<< quantum_backends.size() << " devices
    found.\n";

    std::vector<QDMI_Device> registeredDevices;
    for (const auto& device : quantum_backends) {
        registeredDevices.emplace_back(device);
    }

    return registeredDevices;
}
```

Listing 5.2: Modified Implementation with sys-sage

In this improved implementation, the `QdmiParser` class from sys-sage is used to retrieve the list of available quantum backends through the `get_available_backends()`

method. This method returns a list of all available backends, significantly simplifying the process compared to the original version. The retrieved backends are then stored in the `registeredDevices` vector.

5.1.3. Output

`FoMaC_available_devices` is executed within the test suite `FoMacTest` to verify its functionality. The console output is as follows:

```
[Backend].............Initializing IBM via QDMI
[sys-sage]..............Initiated QDMI session
[sys-sage]..............QDMI_core_device_count returned 1.
[FoMaC].............Total 1 devices found.
```

Listing 5.3: Output of FoMaCTest

The console output from running `FoMaC_available_devices` indicates a sequence of steps beginning with the initialization of the IBM backend through the QDMI, as shown by `[Backend].............Initializing IBM via QDMI`. A successful initiation of a QDMI session is confirmed by `[sys-sage]..............Initiated QDMI session`, establishing the necessary connection between the test framework and the QDMI library. The call to `QDMI_core_device_count` function returns a value of 1, as indicated by `[sys-sage]..............QDMI_core_device_count returned 1`, revealing that one quantum device is available. Finally, `[FoMaC].............Total 1 devices found` summarizes the test outcome, confirming that the QDMI library correctly identified one available device, enabling further operations or tests with this device.

5.1.4. Advantages of using sys-sage

Replacing direct QDMI calls with sys-sage offers several advantages:

- **Cleaner Implementation**: The use of a higher-level C++ interface provided by sys-sage results in cleaner and more maintainable code. By abstracting the complexities of QDMI interactions, the code becomes easier to read and manage.
- **Enhanced Memory Safety**: sys-sage, being a C++ library, provides better memory safety compared to direct C-based QDMI calls. This reduces the risk of memory-related issues, such as leaks or buffer overflows.
- **Reduced Number of Function Calls**: By consolidating multiple QDMI calls into a single sys-sage call, the number of function calls is reduced. This can potentially improve performance by minimizing the overhead associated with frequent function invocations.
- **Centralized Data Management**: sys-sage acts as a centralized repository for quantum backend properties, allowing for efficient data caching and retrieval. This centralized approach ensures that all necessary information is readily available

without repeated queries to the QDMI library. Further, the `QdmiParser` object can be used to create the topology of the quantum devices and fetch any other information in a simpler manner.

Overall, the integration of sys-sage into the FoMaC repository demonstrates the potential for more efficient and effective management of quantum backend properties, paving the way for further enhancements and optimizations in the future.

5.2. Compiler Passes

The `passes` repository is dedicated to developing custom LLVM-compliant passes to optimize quantum circuits represented in the Quantum Intermediate Representation (QIR). Utilizing the robust infrastructure provided by LLVM, these passes are specifically designed to enhance the performance and efficiency of quantum computations. The suite of optimization tools within the `passes` repository addresses the unique challenges of QIR-based quantum circuits, enabling more effective quantum processing.

Compiler passes are integral to MQSS, as they enable platform-dependent optimizations by leveraging the physical properties of quantum backends. sys-sage can play a crucial role in this context by efficiently managing and providing quantum backend properties to various compiler passes. By acting as an intermediary and providing streamlined API calls, sys-sage ensures the compiler can access accurate and up-to-date information about the quantum hardware without bothering about the underlying low-level QDMI implementation. This integration contributes to better optimization and improves code generation for quantum circuits, ultimately enhancing the overall performance of quantum applications.

To investigate this integration, we examine the modifications made to `ArchitectureFactory.cpp`, a key component of the `passes` repository.

The `createArchitecture()` method creates an object to the `Architecture` object of the QMAP library. This is further used for the target-specific compilation passes and even mapping the quantum circuits.

5.2.1. Original Implementation

The original version of the code relied heavily on direct QDMI calls to query and manage the properties of quantum backends. Here is the original implementation:

```
namespace mqt {

Architecture createArchitecture(QDMI_Device dev) {
  int num_qubits = 0;

  int err = QDMI_query_qubits_num(dev, &num_qubits);
  if (err != QDMI_SUCCESS)
    throw std::runtime_error("Could not get number of qubits via QDMI");
  if (num_qubits == 0)
```

```
    throw std::runtime_error("Number of qubits cannot be zero");

  QDMI_Qubit qubits;

  // create a coupling map
  err = QDMI_query_all_qubits(dev, &qubits);

  if (err != QDMI_SUCCESS || qubits == NULL)
    throw std::runtime_error("Could not get qubits via QDMI");

  CouplingMap cm{};
  for (int i = 0; i < num_qubits; i++)
  {
    if (qubits[i].coupling_mapping != NULL && qubits[i].size_coupling_mapping)
    {
      for (int j = 0; j < qubits[i].size_coupling_mapping; j++)
        cm.emplace(i, qubits[i].coupling_mapping[j]);
    }
  }

  free(qubits);

  return {static_cast<std::uint16_t>(num_qubits), cm};
}
} // namespace mqt
```

Listing 5.4: Original Version of ArchitectureFactory.cpp

5.2.2. Implementation with sys-sage

In the modified version, sys-sage is used to streamline the retrieval of quantum backend properties, resulting in cleaner and more maintainable code. Here is the revised implementation:

```
namespace mqt {

Architecture createArchitecture(QDMI_Device dev) {

  // QdmiParser: sys-sage's interface to qdmi (for retrieving the static topology)
  QdmiParser qdmi;

  // An instance to QuantumBackend for storing the topology
  QuantumBackend* qc = new QuantumBackend(0, "IBM_Backend");

  // Create the topology
  qdmi.createQcTopo(qc, dev);

  std::uint16_t num_qubits = qc->GetNumberofQubits();
  CouplingMap cm = qc->GetAllCouplingMaps();

  delete qc;
  return {num_qubits, cm};
```

```
19 }
20 }
```

Listing 5.5: Modified Version of ArchitectureFactory.cpp

5.2.3. Advantages of the Replacement

The updated implementation using sys-sage offers several advantages:

- **Enhanced Code Maintainability:** By abstracting the complexity of direct QDMI calls, the new version of the code is more readable and easier to maintain. The use of sys-sage's higher-level C++ interface simplifies the interaction with quantum backend properties.
- **Improved Memory Safety:** The new approach reduces the risk of memory-related errors. The sys-sage library handles the memory management of quantum backend properties, ensuring a more robust implementation.
- **Efficient Data Management:** sys-sage efficiently caches and manages the data provided by QDMI, leading to a more efficient and streamlined data retrieval process.
- **Cleaner Implementation:** By utilizing sys-sage, the code achieves a cleaner and more modular structure. This separation of concerns enhances the overall design and allows for easier future extensions and modifications.
- **Reduced Number of Function Calls**: By consolidating multiple QDMI calls into a single sys-sage call, the number of function calls is reduced. This can potentially improve performance by minimizing the overhead associated with frequent function invocations.

The integration of sys-sage within the `passes` repository exemplifies how leveraging this library can significantly improve the management and retrieval of quantum backend properties, ultimately leading to more effective compiler optimizations.

5.3. QMAP

QMAP is an open-source Munich Quantum Toolkit (MQT) tool that maps quantum circuits to specific quantum computing architectures [WB23]. The primary challenge it addresses is ensuring that the gates in a quantum circuit align with the topology of the target architecture, facilitating effective execution while minimizing additional gate counts and maintaining fidelity. QMAP provides automated and user-friendly methods for addressing the quantum circuit mapping problem, thereby aiding developers in implementing quantum algorithms on actual quantum machines.

Integrating sys-sage with QMAP

To store the architecture properties of a quantum backend, QMAP employs class `Architecture` that encapsulates all architecture-related information. Initially, we considered replacing the `Architecture` class with sys-sage. However, this approach risked making sys-sage overly specific to QMAP, compromising its universality. For instance, QMAP utilizes specific typedefs like `Edge`, `CouplingMap`, and `QubitSubset` tailored to its circuit mappers, which may not be necessary for other tools. Integrating these custom data types into the core structure of sys-sage would be redundant, as not all tools require such specific return types or parameters.

Instead of replacing the `Architecture` class entirely, we opted for a more flexible solution. QMAP supports multiple input methods for importing quantum backend properties, including text files, JSON files, and QDMI. By leveraging sys-sage to import all backend properties, QMAP can reduce the number of required calls, streamlining the process as shown in the example with the passes repository.

We propose creating an `external interface` within sys-sage to handle tool-specific calls. For instance, QMAP can include a header file (for example, `mqt-sys-sage.h`) containing functions that utilize sys-sage's interface to provide properties compatible with QMAP's data structures. This approach allows for seamless extension to other libraries or tools without altering sys-sage's internal representation.

Example Integration Code

Listing A.8 contains the example of functions in `mqt-sys-sage.h` that act as an external interface to QMAP. For example, one of these functions, `GetAllCouplingMaps`, is as shown in Listing 5.6.

```
using Edge        = std::pair<std::uint16_t, std::uint16_t>;
using CouplingMap = std::set<Edge>;
using QubitSubset = std::set<std::uint16_t>;

CouplingMap GetAllCouplingMaps(QuantumBackend backend) {
    CouplingMap result;
    for(auto i = 0; i < num_qubits; ++i) {
        Qubit* q = dynamic_cast<Qubit*>(GetChild(i));
        auto coupling_map = q->GetCouplingMapping();
        for (size_t j = 0; j < coupling_map.size(); ++j) {
            result.emplace(i, coupling_map[j]);
        }
    }
    return result;
}
```

Listing 5.6: External Interface for QMAP

`GetAllCouplingMaps` takes an object of type `QuantumBackend` (which contains the topology of a quantum backend) and returns a `CouplingMap` that can be used by QMAP.

As shown in Listing 5.5, this function can be used to create an object of the `Architecture` class, which internally stores the topology of the quantum backend.

This integration ensures that QMAP can efficiently utilize sys-sage to retrieve backend properties, reducing complexity and enhancing compatibility with other tools and libraries.

6. Conclusion and Future Research Outlook

In this thesis, we have explored and extended the capabilities of sys-sage to enhance its functionality for quantum systems. The primary objective was to create a unified interface that seamlessly integrated various quantum technologies. By providing a higher-level representation of quantum technologies through static and dynamic topologies, our work aims to facilitate better management and interaction with quantum systems.

Additionaly, the core structure of QDMI has been thoroughly discussed, and a significant proposal has been put forward — the QDMI implementation for the IBM backends. This implementation, which has been seamlessly integrated with sys-sage and other MQSS tools, serves the crucial function of retrieving and providing platform-specific properties. Its successful integration with the QRM further underscores its potential impact.

The extensions made to the sys-sage library to accommodate the representation of quantum systems have focused on universality and compatibility with various tools and libraries. The proposed API calls for creating and interacting with these representations were thoroughly detailed, offering insights into how users can leverage sys-sage for various quantum computing tasks, explicitly creating the topology of the systems. Extensions include new C++ classes derived from `Component`, such as `QuantumBackend` and `Qubit`. A newer abstraction for representing the interaction of HPC and QC components, called `Relation`, has been introduced. `DataPath` for HPC systems and `QuantumGate` for QC systems are derived from the `Relation` class. These abstractions and extensions will help efficiently and accurately represent HPC-QC systems.

The extensions have been meticulously tested with dummy IBM backends, and their integration, compatibility, and performance have been investigated. The integration is promising, and it can be concluded that sys-sage can accurately represent quantum computers' topology and platform characteristics.

Further, the performance implications of the topology generation and retrieval have also been investigated. It has been observed that sys-sage adds only minimal overhead to QDMI during topology generation. This overhead becomes negligible as the number of qubits increases, particularly beyond 500 qubits. Also, one of the most important getter functions, GetAllCouplingMaps, takes less than a millisecond to provide the coupling maps of all the qubits for backends with less than 1000 qubits. . Despite this, further optimization could improve the performance of such functions.

One of the key areas of investigation in this thesis has been the potential for integrating sys-sage with other tools within the MQSS ecosystem, including FoMaC, passes, and QMAP. While investigating the possibilities of integration, sys-sage has been able to

reduce the number of native QDMI calls and provide all the information with higher-level C++ based API calls. This integration could significantly enhance quantum computing tools' overall functionality and interoperability, fostering a more cohesive environment for quantum research and development.

6.1. Future Work

Continued efforts could focus on expanding sys-sage integration with additional quantum computing tools and platforms, potentially incorporating more diverse quantum technologies. Specifically, the integration with quantum computers based on neutral atoms and trapped ions systems should be further tested as soon as their QDMI implementations are available.

This thesis represents an experimental work investigating whether sys-sage, a library traditionally centered on HPC, can be effectively used for QC systems. While this investigation has been conducted with existing tools, there is ample opportunity for future efforts to focus on checking its compatibility and finding more use cases with other tools within or outside MQSS. For instance, one promising future opportunity is the integration of sys-sage with the Munich Quantum Portal, a web-based portal for accessing and managing quantum resources.

Additionally, future research could explore optimization techniques to enhance the performance and efficiency of sys-sage in handling complex quantum system representations. As time progresses, the number of physical qubits will surely increase. Ensuring that the internal representation can support retrieving and providing ample information with the best performance will become critical.

Further, sys-sage's current role focuses on retrieving information, storing it internally, and providing it via API calls. sys-sage acts as a caching database, which uses QDMI to store the physical characteristics of backends. Future efforts can also focus on enhanced post-processing, analyses, and correlation of the retrieved information. For example, it could be possible to analyze the available coupling maps and identify the best pair of qubits for a particular operation.

Finally, efforts can be put into testing and investigating sys-sage for unified HPC-QC systems. A user could retrieve the topology of the classical and quantum systems and use that information to schedule tasks accordingly. This potential for unification promises efficient resource usage and opens up new possibilities for the optimal utilization of quantum computing power, instilling hope for the future of quantum computing.

A. Appendix

A.1. QDMIOpenClose Test

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <qdmi.h>
#include "qdmi_internal.h"

int QDMI_session_init(QInfo info, QDMI_Session *session);
int QDMI_session_finalize(QDMI_Session session);
QDMI_Library find_library_by_name(const char *libname);

#define CHECK_ERR(a, b)                          \
    {                                            \
        if (a != QDMI_SUCCESS)                   \
        {                                        \
            printf("\n[Error]: %i at %s", a, b); \
            return 1;                            \
        }                                        \
    }

int main(int argc, char **argv)
{
    // Initialization of Structures
    QInfo info;
    QDMI_Session session = NULL;
    QDMI_Library lib;
    QDMI_Device device;
    int err;

    device = malloc(sizeof(struct QDMI_Device_impl_d));
    if (device == NULL)
    {
        printf("\n[ERROR]: Device could not be created");
        exit(EXIT_FAILURE);
    }

    err = QInfo_create(&info);
    CHECK_ERR(err, "QInfo_create");

    // Session Initialization
    err = QDMI_session_init(info, &session);
```

```
42      CHECK_ERR(err, "QDMI_session_init");
43
44      // Loading IBM Backend Library
45      lib = find_library_by_name("/home/diogenes/bin/lib/libbackend_ibm.so");
46      if (!lib)
47      {
48          printf("\n[ERROR]: Library could not be found");
49          exit(EXIT_FAILURE);
50      }
51      device->library = *lib;
52
53      // Querying Device Information
54      int num_qubits;
55      err = QDMI_query_qubits_num(device, &num_qubits);
56      CHECK_ERR(err, "QDMI_query_qubits_num");
57
58      int status = 0;
59      err = QDMI_device_status(device, device->library.info, &status);
60      CHECK_ERR(err, "QDMI_device_status");
61
62      printf("    [QDMIOpenClose].............Found %d qubits.\n", num_qubits);
63
64      // Session Finalization
65      err = QDMI_session_finalize(session);
66      CHECK_ERR(err, "QDMI_session_finalize");
67      printf("    [QDMIOpenClose].............Closed connection to the device.\n");
68
69      err = QInfo_free(info);
70      CHECK_ERR(err, "QInfo_free");
71
72      // Cleanup
73      free(device);
74
75      printf("\n[DEBUG]: Test Finished\n\n");
76      return 0;
77  }
```

Listing A.1: QDMIOpenClose Test

A.2. QDMI Integrated with Quantum Resource Manager

```
[qresourcemanager_d]..Listening on queue queue_manager
[Backend].............Initializing IBM via QDMI
[Backend].............Fetching config file
[qresourcemanager_d]..Received a QuantumTask
[Generator Runner]....Invoking generator: libgenerator_cutter.so
[Generator]...........Returning generated sub-circuits to the Generator
    Runner

[Scheduler Runner]....Invoking scheduler: libscheduler_round_robin.so
```

```
[FoMaC]..............Available device found: /libbackend_ibm.so
[Scheduler]...........1 available device(s)
[Scheduler]...........Choosing /libbackend_ibm.so as target device for job
    with ID 0
[Backend].............Returning available qubits
[FoMaC]..............Coupling mapping of qubit[0]: { 1 }
[FoMaC]..............Coupling mapping of qubit[1]: { 0 2 }
[FoMaC]..............Coupling mapping of qubit[2]: { 1 3 }
[FoMaC]..............Coupling mapping of qubit[3]: { 2 4 }
[FoMaC]..............Coupling mapping of qubit[4]: { 3 }
[Selector Runner].....Invoking selector: libselector_manual.so
[Selector]............Returning list of passes to the Selector Runner
[Pass Runner].........Flag inserted: "lrz_supports_qir" = true
[Pass Runner].........Applying pass: QirQMapPass
[Backend].............Returning available qubits
[Pass]...............Warning: gate not supported: __quantum__rt__initialize
[Pass]...............Warning: gate not supported:
    __quantum__rt__tuple_record_output
[Pass]...............Warning: gate not supported:
    __quantum__rt__result_record_output
[Pass]...............Warning: gate not supported:
    __quantum__rt__result_record_output
[Pass Runner].........Applying pass: QirDivisionByZeroPass
[Pass Runner].........Applying pass: QirXCnotXReductionPass
[Pass Runner].........Applying pass: QirCommuteCnotRxPass
[Pass Runner].........Applying pass: QirCommuteRxCnotPass
[Pass Runner].........Applying pass: QirCommuteCnotXPass
[Pass Runner].........Applying pass: QirCommuteXCnotPass
[Pass Runner].........Applying pass: QirCommuteCnotZPass
[Pass Runner].........Applying pass: QirCommuteZCnotPass
[Pass Runner].........Applying pass: QirAnnotateUnsupportedGatesPass
[Pass Runner].........Applying pass: QirPlaceIrreversibleGatesInMetadataPass
[Pass]...............Reversible gate found: __quantum__qis__h__body
[Pass]...............Reversible gate found: __quantum__qis__cx__body
[Pass]...............Reversible gate found: __quantum__qis__x__body
[Pass]...............Reversible gate found: __quantum__qis__z__body
[Pass]...............Reversible gate found: __quantum__qis__y__body
[Pass]...............Reversible gate found: __quantum__qis__s__body
[Pass]...............Reversible gate found: __quantum__qis__cz__body
[Pass]...............Reversible gate found: __quantum__qis__mz__body
[Pass Runner].........Applying pass: QirCNotToHCZHDecompositionPass
[Pass Runner].........Applying pass: QirSwapToCnotsDecompositionPass
[Pass Runner].........Applying pass: QirCZToHCnotHDecompositionPass
[Pass Runner].........Applying pass: QirFunctionAnnotatorPass
[Pass Runner].........Applying pass: QirRedundantGatesCancellationPass
[Pass]...............Redundant gate pair found: __quantum__qis__cx__body
[Pass]...............Redundant gate pair found: __quantum__qis__cx__body
[Pass]...............Redundant gate pair found: __quantum__qis__mz__body
[Pass]...............Redundant gate pair found: __quantum__qis__mz__body
[Pass Runner].........Applying pass: QirGroupingPass
[Pass Runner].........Applying pass: QirDeferMeasurementPass
```

```
[Pass Runner].........Applying pass:
    QirRemoveBasicBlocksWithSingleNonConditionalBranchInstsPass
[Pass Runner].........Applying pass: QirQubitRemapPass
[Pass Runner].........Applying pass: QirResourceAnnotationPass
[Pass Runner].........Applying pass: QirHadamardAndXGateSwitchPass
[Pass Runner].........Applying pass: QirHadamardAndYGateSwitchPass
[Pass Runner].........Applying pass: QirHadamardAndZGateSwitchPass
[Pass Runner].........Applying pass: QirXGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirYGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirZGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirSToSDaggerPass
[Pass Runner].........Applying pass: QirSDaggerToSPass
[Pass Runner].........Applying pass: QirReverseCnotPass
[Pass]................Reversing Cnot
[Pass Runner].........Applying pass: QirSwapAndCnotReplacementPass
[Pass Runner].........Applying pass: QirFunctionReplacementPass
[Pass]................Function has 'replaceWith' attribute:
    __quantum__qis__cz__body
[Pass]................Function is a replacement          :
    __quantum__qis__cz_to_hcnoth__body
[Pass Runner].........Applying pass: QirReplaceConstantBranchesPass
[Pass Runner].........Applying pass: QirRemoveNonEntrypointFunctionsPass
[Backend].............QDMI_control_pack_qir
[Backend].............QDMI_control_submit

[Scheduler Runner]....Invoking scheduler: libscheduler_round_robin.so
[FoMaC]...............Available device found: /libbackend_ibm.so
[Scheduler]...........1 available device(s)
[Scheduler]...........Choosing /libbackend_ibm.so as target device for job
    with ID 1
[Backend].............Returning available qubits
[FoMaC]...............Coupling mapping of qubit[0]: { 1 }
[FoMaC]...............Coupling mapping of qubit[1]: { 0 2 }
[FoMaC]...............Coupling mapping of qubit[2]: { 1 3 }
[FoMaC]...............Coupling mapping of qubit[3]: { 2 4 }
[FoMaC]...............Coupling mapping of qubit[4]: { 3 }
[Selector Runner].....Invoking selector: libselector_manual.so
[Selector]............Returning list of passes to the Selector Runner
[Pass Runner].........Flag inserted: "lrz_supports_qir" = true
[Pass Runner].........Applying pass: QirQMapPass
[Backend].............Returning available qubits
[Pass]................Warning: gate not supported: __quantum__rt__initialize
[Pass]................Warning: gate not supported:
    __quantum__rt__tuple_record_output
[Pass]................Warning: gate not supported:
    __quantum__rt__result_record_output
[Pass]................Warning: gate not supported:
    __quantum__rt__result_record_output
[Pass Runner].........Applying pass: QirDivisionByZeroPass
[Pass Runner].........Applying pass: QirXCnotXReductionPass
[Pass Runner].........Applying pass: QirCommuteCnotRxPass
[Pass Runner].........Applying pass: QirCommuteRxCnotPass
```

```
[Pass Runner].........Applying pass: QirCommuteCnotXPass
[Pass Runner].........Applying pass: QirCommuteXCnotPass
[Pass Runner].........Applying pass: QirCommuteCnotZPass
[Pass Runner].........Applying pass: QirCommuteZCnotPass
[Pass Runner].........Applying pass: QirAnnotateUnsupportedGatesPass
[Pass Runner].........Applying pass: QirPlaceIrreversibleGatesInMetadataPass
[Pass]................Reversible gate found: __quantum__qis__h__body
[Pass]................Reversible gate found: __quantum__qis__cx__body
[Pass]................Reversible gate found: __quantum__qis__x__body
[Pass]................Reversible gate found: __quantum__qis__z__body
[Pass]................Reversible gate found: __quantum__qis__y__body
[Pass]................Reversible gate found: __quantum__qis__s__body
[Pass]................Reversible gate found: __quantum__qis__cz__body
[Pass]................Reversible gate found: __quantum__qis__mz__body
[Pass Runner].........Applying pass: QirCNotToHCZHDecompositionPass
[Pass Runner].........Applying pass: QirSwapToCnotsDecompositionPass
[Pass Runner].........Applying pass: QirCZToHCnotHDecompositionPass
[Pass Runner].........Applying pass: QirFunctionAnnotatorPass
[Pass Runner].........Applying pass: QirRedundantGatesCancellationPass
[Pass]................Redundant gate pair found: __quantum__qis__cx__body
[Pass]................Redundant gate pair found: __quantum__qis__cx__body
[Pass]................Redundant gate pair found: __quantum__qis__mz__body
[Pass]................Redundant gate pair found: __quantum__qis__mz__body
[Pass Runner].........Applying pass: QirGroupingPass
[Pass Runner].........Applying pass: QirDeferMeasurementPass
[Pass Runner].........Applying pass:
    QirRemoveBasicBlocksWithSingleNonConditionalBranchInstsPass
[Pass Runner].........Applying pass: QirQubitRemapPass
[Pass Runner].........Applying pass: QirResourceAnnotationPass
[Pass Runner].........Applying pass: QirHadamardAndXGateSwitchPass
[Pass Runner].........Applying pass: QirHadamardAndYGateSwitchPass
[Pass Runner].........Applying pass: QirHadamardAndZGateSwitchPass
[Pass Runner].........Applying pass: QirXGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirYGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirZGateAndHadamardSwitchPass
[Pass Runner].........Applying pass: QirSToSDaggerPass
[Pass Runner].........Applying pass: QirSDaggerToSPass
[Pass Runner].........Applying pass: QirReverseCnotPass
[Pass]................Reversing Cnot
[Pass Runner].........Applying pass: QirSwapAndCnotReplacementPass
[Pass Runner].........Applying pass: QirFunctionReplacementPass
[Pass]................Function has 'replaceWith' attribute:
    __quantum__qis__cz__body
[Pass]................Function is a replacement          :
    __quantum__qis__cz_to_hcnoth__body
[Pass Runner].........Applying pass: QirReplaceConstantBranchesPass
[Pass Runner].........Applying pass: QirRemoveNonEntrypointFunctionsPass
[Backend].............QDMI_control_pack_qir
[Backend].............QDMI_control_submit
[qresourcemanager_d]..Adapted QIR sent to the Quantum Daemon
```

Listing A.2: QDMI Integrated with Quantum Resource Manager

A.3. Sample Topology of a HPC system

```
-- Parsing Hwloc output from file ./example_data/skylake_hwloc.xml
-- End parseHwlocOutput
Total num HW threads: 24
------------------ Printing the whole tree ------------------
Topology (name sys-sage Topology) id 0 - children: 1 level: 0
  Node (name Node) id 1 - children: 2 level: 1
    Chip (name socket) id 0 - children: 1 level: 2
      Cache (name Cache) id 7 - children: 2 level: 3
        NUMA (name Numa) id 0 - children: 6 level: 4
          Cache (name Cache) id 6 - children: 1 level: 5
            Cache (name Cache) id 5 - children: 1 level: 6
              Core (name Core) id 0 - children: 1 level: 7
                HW_thread (name HW_thread) id 0 - children: 0 level: 8
          Cache (name Cache) id 11 - children: 1 level: 5
            Cache (name Cache) id 10 - children: 1 level: 6
              Core (name Core) id 1 - children: 1 level: 7
                HW_thread (name HW_thread) id 1 - children: 0 level: 8
          Cache (name Cache) id 15 - children: 1 level: 5
            Cache (name Cache) id 14 - children: 1 level: 6
              Core (name Core) id 2 - children: 1 level: 7
                HW_thread (name HW_thread) id 2 - children: 0 level: 8
          Cache (name Cache) id 19 - children: 1 level: 5
            Cache (name Cache) id 18 - children: 1 level: 6
              Core (name Core) id 8 - children: 1 level: 7
                HW_thread (name HW_thread) id 3 - children: 0 level: 8
          Cache (name Cache) id 23 - children: 1 level: 5
            Cache (name Cache) id 22 - children: 1 level: 6
              Core (name Core) id 9 - children: 1 level: 7
                HW_thread (name HW_thread) id 4 - children: 0 level: 8
          Cache (name Cache) id 27 - children: 1 level: 5
            Cache (name Cache) id 26 - children: 1 level: 6
              Core (name Core) id 10 - children: 1 level: 7
                HW_thread (name HW_thread) id 5 - children: 0 level: 8
        NUMA (name Numa) id 1 - children: 6 level: 4
          Cache (name Cache) id 31 - children: 1 level: 5
            Cache (name Cache) id 30 - children: 1 level: 6
              Core (name Core) id 3 - children: 1 level: 7
                HW_thread (name HW_thread) id 6 - children: 0 level: 8
          Cache (name Cache) id 35 - children: 1 level: 5
            Cache (name Cache) id 34 - children: 1 level: 6
              Core (name Core) id 4 - children: 1 level: 7
                HW_thread (name HW_thread) id 7 - children: 0 level: 8
          Cache (name Cache) id 39 - children: 1 level: 5
            Cache (name Cache) id 38 - children: 1 level: 6
              Core (name Core) id 5 - children: 1 level: 7
                HW_thread (name HW_thread) id 8 - children: 0 level: 8
          Cache (name Cache) id 43 - children: 1 level: 5
            Cache (name Cache) id 42 - children: 1 level: 6
              Core (name Core) id 11 - children: 1 level: 7
```

```
              HW_thread (name HW_thread) id 9 – children: 0 level: 8
        Cache (name Cache) id 47 – children: 1 level: 5
          Cache (name Cache) id 46 – children: 1 level: 6
            Core (name Core) id 12 – children: 1 level: 7
              HW_thread (name HW_thread) id 10 – children: 0 level: 8
        Cache (name Cache) id 51 – children: 1 level: 5
          Cache (name Cache) id 50 – children: 1 level: 6
            Core (name Core) id 13 – children: 1 level: 7
              HW_thread (name HW_thread) id 11 – children: 0 level: 8
  Chip (name socket) id 1 – children: 1 level: 2
    Cache (name Cache) id 57 – children: 2 level: 3
      NUMA (name Numa) id 2 – children: 6 level: 4
        Cache (name Cache) id 56 – children: 1 level: 5
          Cache (name Cache) id 55 – children: 1 level: 6
            Core (name Core) id 0 – children: 1 level: 7
              HW_thread (name HW_thread) id 12 – children: 0 level: 8
        Cache (name Cache) id 61 – children: 1 level: 5
          Cache (name Cache) id 60 – children: 1 level: 6
            Core (name Core) id 1 – children: 1 level: 7
              HW_thread (name HW_thread) id 13 – children: 0 level: 8
        Cache (name Cache) id 65 – children: 1 level: 5
          Cache (name Cache) id 64 – children: 1 level: 6
            Core (name Core) id 2 – children: 1 level: 7
              HW_thread (name HW_thread) id 14 – children: 0 level: 8
        Cache (name Cache) id 69 – children: 1 level: 5
          Cache (name Cache) id 68 – children: 1 level: 6
            Core (name Core) id 8 – children: 1 level: 7
              HW_thread (name HW_thread) id 15 – children: 0 level: 8
        Cache (name Cache) id 73 – children: 1 level: 5
          Cache (name Cache) id 72 – children: 1 level: 6
            Core (name Core) id 9 – children: 1 level: 7
              HW_thread (name HW_thread) id 16 – children: 0 level: 8
        Cache (name Cache) id 77 – children: 1 level: 5
          Cache (name Cache) id 76 – children: 1 level: 6
            Core (name Core) id 10 – children: 1 level: 7
              HW_thread (name HW_thread) id 17 – children: 0 level: 8
      NUMA (name Numa) id 3 – children: 6 level: 4
        Cache (name Cache) id 81 – children: 1 level: 5
          Cache (name Cache) id 80 – children: 1 level: 6
            Core (name Core) id 3 – children: 1 level: 7
              HW_thread (name HW_thread) id 18 – children: 0 level: 8
        Cache (name Cache) id 85 – children: 1 level: 5
          Cache (name Cache) id 84 – children: 1 level: 6
            Core (name Core) id 4 – children: 1 level: 7
              HW_thread (name HW_thread) id 19 – children: 0 level: 8
        Cache (name Cache) id 89 – children: 1 level: 5
          Cache (name Cache) id 88 – children: 1 level: 6
            Core (name Core) id 5 – children: 1 level: 7
              HW_thread (name HW_thread) id 20 – children: 0 level: 8
        Cache (name Cache) id 93 – children: 1 level: 5
          Cache (name Cache) id 92 – children: 1 level: 6
            Core (name Core) id 11 – children: 1 level: 7
```

```
                HW_thread (name HW_thread) id 21 - children: 0 level: 8
          Cache (name Cache) id 97 - children: 1 level: 5
            Cache (name Cache) id 96 - children: 1 level: 6
              Core (name Core) id 12 - children: 1 level: 7
                HW_thread (name HW_thread) id 22 - children: 0 level: 8
          Cache (name Cache) id 101 - children: 1 level: 5
            Cache (name Cache) id 100 - children: 1 level: 6
              Core (name Core) id 13 - children: 1 level: 7
                HW_thread (name HW_thread) id 23 - children: 0 level: 8
-----------------                       -----------------
```

Listing A.3: Sample Topology of a HPC system

A.4. class QuantumBackend

A.4.1. Declaration

```
class QuantumBackend : public Component {
public:
    QuantumBackend(int _id = 0, string _name = "QuantumBackend");
    QuantumBackend(Component * parent, int _id = 0, string _name = "QuantumBackend");
    void SetNumberofQubits(int _num_qubits);
    void SetQDMIDevice(QDMI_Device dev);
    QDMI_Device GetQDMIDevice();
    int GetNumberofQubits () const;
    void addGate(QuantumGate *gate);
    std::vector<QuantumGate*> GetGatesBySize(size_t _gate_size) const;
    std::vector<QuantumGate*> GetGatesByType(size_t _gate_type) const;
    std::vector<QuantumGate*> GetAllGateTypes() const;
    int GetNumberofGates() const;
    std::vector<Qubit *> GetAllQubits();
    std::set<std::pair<std::uint16_t, std::uint16_t> > GetAllCouplingMaps();
    void RefreshTopology(std::set<int> qubit_indices); // qubit_indices: indices of the
    qubits that need to be refreshed

    ~QuantumBackend() override = default;

private:
    int num_qubits;
    int num_gates;
    QDMI_Device device; // For refreshing the topology
    std::vector <QuantumGate*> gate_types;
};
```

Listing A.4: class QuantumBackend

A.4.2. Constructors

- `QuantumBackend(Component *parent, int _id = 0, string _name = "QuantumBackend")`

  - **Purpose**: Initializes a `QuantumBackend` object and inserts it into the Component Tree as a child of the specified parent component.
  - **Parameters**:
    * `parent`: The parent `Component` in the Component Tree.
    * `_id`: The unique identifier for the `QuantumBackend`, with a default value of 0.
    * `_name`: The name of the `QuantumBackend`, with a default value of "QuantumBackend".
  - **Sets**:
    * `componentType` to `SYS_SAGE_COMPONENT_QUANTUM_BACKEND`.

- `QuantumBackend(int _id = 0, string _name = "QuantumBackend")`
  - **Purpose**: Initializes a `QuantumBackend` object without automatically inserting it into the Component Tree.
  - **Parameters**:
    * `_id`: The unique identifier for the `QuantumBackend`, with a default value of 0.
    * `_name`: The name of the `QuantumBackend`, with a default value of "QuantumBackend".
  - **Sets**:
    * `componentType` to `SYS_SAGE_COMPONENT_QUANTUM_BACKEND`.

A.4.3. Public Members

- `void SetNumberofQubits(int _num_qubits)`
  - **Purpose**: Sets the number of qubits in the quantum backend.
  - **Parameters**:
    * `_num_qubits`: The number of qubits.

- `int GetNumberofQubits() const`
  - **Purpose**: Retrieves the number of qubits in the quantum backend.
  - **Returns**: The number of qubits as an integer.

- `void addGate(QuantumGate *gate)`
  - **Purpose**: Adds a `QuantumGate` to the quantum backend.
  - **Parameters**:
    * `gate`: A pointer to the `QuantumGate` to be added.

- `std::vector<QuantumGate*> GetGatesBySize(size_t _gate_size) const`
  - **Purpose**: Retrieves all gates of a specific size.
  - **Parameters**:
    - `_gate_size`: The size of the gates to be retrieved.
  - **Returns**: A vector of pointers to `QuantumGate` objects of the specified size.
- `std::vector<QuantumGate*> GetGatesByType(size_t _gate_type) const`
  - **Purpose**: Retrieves all gates of a specific type.
  - **Parameters**:
    - `_gate_type`: The type of gate to be retrieved.
  - **Returns**: A vector of pointers to `QuantumGate` objects of the specified type.
- `std::vector<QuantumGate*> GetAllGateTypes() const`
  - **Purpose**: Retrieves all gates in the quantum backend.
  - **Returns**: A vector of pointers to all `QuantumGate` objects.
- `int GetNumberofGates() const`
  - **Purpose**: Retrieves the total number of gates in the quantum backend.
  - **Returns**: The number of gates as an integer.
- `std::vector<Qubit *> GetAllQubits()`
  - **Purpose**: Retrieves all qubits associated with the quantum backend.
  - **Returns**: A vector of pointers to `Qubit` objects.
  - **Details**: This function fetches all child components of type `SYS_SAGE_COMPONENT_QUBIT`, dynamically casts them to `Qubit` pointers, and returns them in a vector.
- `std::set<std::pair<std::uint16_t, std::uint16_t» GetAllCouplingMaps()`
  - **Purpose**: Retrieves the coupling map for all qubits in the quantum backend.
  - **Returns**: A set of pairs, each representing a coupling between two qubits.
  - **Details**: This function iterates over all qubits, retrieves their coupling maps, and stores the couplings as pairs in a set to ensure uniqueness.
- `void SetQDMIDevice(QDMI_Device dev)`
  - **Purpose**: Sets the `QDMI_Device` associated with the quantum backend.
  - **Parameters**:
    - `dev`: The `QDMI_Device` to be associated with the quantum backend.
- `QDMI_Device GetQDMIDevice()`
  - **Purpose**: Retrieves the `QDMI_Device` associated with the quantum backend.

  - **Returns**: The `QDMI_Device` object.

- `void RefreshTopology()`
  - **Purpose**: Refreshes the topology of the quantum backend, updating the internal representation of the qubits and gates. This function calls the `RefreshProperties()` method of every child `Qubit` and updates the properties. This is meant to update the dynamic properties when the user requests them.

A.4.4. Private Members

- `int num_qubits`
  - **Purpose**: This private member stores the number of qubits in the quantum backend.

- `int num_gates`
  - **Purpose**: This private member stores the number of gates in the quantum backend.

- `std::vector<QuantumGate*> gate_types`
  - **Purpose**: This private member stores a vector of pointers to `QuantumGate` objects, representing the different gate types in the quantum backend.

- `QDMI_Device device`
  - **Purpose**: This private member stores the handle to the corresponding `QDMI_Device` which the topology actually corresponds to. This member is meant to refresh the topology and all the dynamic information.

A.5. class Qubit

A.5.1. Declaration

```
class Qubit : public Component {
public:
    Qubit(int _id = 0, string _name = "Qubit");
    Qubit(Component * parent, int _id = 0, string _name = "Qubit");
    void SetCouplingMapping( const std::vector <int> &coupling_mapping, const int &
    size_coupling_mapping);
    void SetProperties(double t1, double t2, double readout_error, double readout_length)
    ;
    const std::vector <int> &GetCouplingMapping() const;
    const double GetT1() const;
    const double GetT2() const;
    const double GetReadoutError() const;
    const double GetReadoutLength() const;
```

```
12      const double GetFrequency() const;
13      void RefreshProperties();
14      ~Qubit() override = default;
15
16  private:
17      std::vector <int> _coupling_mapping;
18      int _size_coupling_mapping;
19      double fidelity;
20      double _t1;
21      double _t2;
22      double _readout_error;
23      double _readout_length;
24      double _fequency;
25      std::string _calibration_time;
26  };
```

Listing A.5: class Qubit

A.5.2. Constructors

- `Qubit(Component *parent, int _id = 0, string _name = "Qubit")`
  - **Purpose**: Initializes a `Qubit` object and inserts it into the Component Tree as a child of the specified parent component.
  - **Parameters**:
    - `parent`: The parent `Component` in the Component Tree.
    - `_id`: The unique identifier for the `Qubit`, with a default value of 0.
    - `_name`: The name of the `Qubit`, with a default value of "Qubit".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUBIT`.
- `Qubit(int _id = 0, string _name = "Qubit")`
  - **Purpose**: Initializes a `Qubit` object without automatically inserting it into the Component Tree.
  - **Parameters**:
    - `_id`: The unique identifier for the `Qubit`, with a default value of 0.
    - `_name`: The name of the `Qubit`, with a default value of "Qubit".
  - **Sets**:
    - `componentType` to `SYS_SAGE_COMPONENT_QUBIT`.

A.5.3. Public Members

- `void SetCouplingMapping(const std::vector<int> \&coupling_mapping, const int \&size_coupling_mapping)`

  - **Purpose**: Sets the coupling mapping for the qubit.
  - **Parameters**:
    * `coupling_mapping`: A vector representing the coupling mapping.
    * `size_coupling_mapping`: The size of the coupling mapping.

- `void SetProperties(double t1, double t2, double readout_error, double readout_length)`
  - **Purpose**: Sets the physical properties of the qubit.
  - **Parameters**:
    * `t1`: The relaxation time of the qubit.
    * `t2`: The dephasing time of the qubit.
    * `readout_error`: The readout error rate of the qubit.
    * `readout_length`: The readout length of the qubit.

- `const std::vector<int>\& GetCouplingMapping() const`
  - **Purpose**: Retrieves the coupling mapping of the qubit.
  - **Returns**: A constant reference to a vector representing the coupling mapping.

- `const double GetT1() const`
  - **Purpose**: Retrieves the relaxation time of the qubit.
  - **Returns**: The relaxation time as a double.

- `const double GetT2() const`
  - **Purpose**: Retrieves the dephasing time of the qubit.
  - **Returns**: The dephasing time as a double.

- `const double GetReadoutError() const`
  - **Purpose**: Retrieves the readout error rate of the qubit.
  - **Returns**: The readout error rate as a double.

- `const double GetReadoutLength() const`
  - **Purpose**: Retrieves the readout length of the qubit.
  - **Returns**: The readout length as a double.

- `const double GetFrequency() const`
  - **Purpose**: Retrieves the operational frequency of the qubit.
  - **Returns**: The frequency as a double.

- `void RefreshProperties()`
  - **Purpose**: Refreshes the properties of the qubit, updating the internal representation of the qubit's properties. This function is meant to update the dynamic properties when the user requests them.

A.5.4. Private Members

- `std::vector<int> _coupling_mapping`
  - **Purpose**: Stores the coupling mapping for the qubit.
- `int _size_coupling_mapping`
  - **Purpose**: Stores the size of the coupling mapping.
- `double _t1`
  - **Purpose**: Stores the relaxation time of the qubit.
- `double _t2`
  - **Purpose**: Stores the dephasing time of the qubit.
- `double _readout_error`
  - **Purpose**: Stores the readout error rate of the qubit.
- `double _readout_length`
  - **Purpose**: Stores the readout length of the qubit.
- `double _frequency`
  - **Purpose**: Stores the operational frequency of the qubit.
- `std::string _calibration_time`
  - **Purpose**: Stores the last calibration time of the qubit.

A.6. class QuantumGate

A.6.1. Declaration

```
class QuantumGate : public Relation {

public:
    QuantumGate();
    QuantumGate(size_t _gate_size);
    QuantumGate(size_t _gate_size, std::string _name, double _fidelity, std::string
    _unitary);
    QuantumGate(size_t _gate_size, const std::vector<Qubit *> & _qubits);
    QuantumGate(size_t _gate_size, const std::vector<Qubit *> & _qubits, std::string
    _name, double _fidelity, std::string _unitary);
    void SetGateProperties(std::string _name, double _fidelity, std::string _unitary);
    void SetGateCouplingMap(std::vector<std::vector<Qubit*>> _coupling_mapping);
    void SetAdditionalProperties();
    double GetFidelity() const;
    size_t GetGateSize() const;
    std::string GetUnitary() const;
```

```
15      void Print() override;
16      void DeleteRelation() override;
17
18  private:
19
20      size_t gate_size;
21      int gate_length;
22      std::string unitary;
23      double fidelity;
24      std::vector<std::vector<Qubit*>> coupling_mapping;
25      std::vector<Qubit*> qubits; // Remove from here.
26      std::map<std::string, double> additional_properties;
27  };
```

Listing A.6: class QuantumGate

A.6.2. Constructors

- `QuantumGate()`
  - **Purpose**: Default constructor for `QuantumGate`.
- `QuantumGate(size_t _gate_size)`
  - **Purpose**: Initializes a `QuantumGate` object with a specified gate size.
  - **Parameters**:
    - `_gate_size`: The number of qubits involved in the quantum gate.

A.6.3. Public Members

- `void SetGateProperties(std::string _name, double _fidelity, std::string _unitary)`
  - **Purpose**: Sets the properties of the quantum gate.
  - **Parameters**:
    - `_name`: The name of the quantum gate.
    - `_fidelity`: The fidelity of the quantum gate.
    - `_unitary`: The unitary matrix representation of the quantum gate.
- `void SetGateCouplingMap(std::vector<std::vector<Qubit*» _coupling_mapping)`
  - **Purpose**: Sets the coupling map for the quantum gate.
  - **Parameters**:
    - `_coupling_mapping`: A vector of vectors representing the coupling mapping of qubits.
- `void SetAdditionalProperties()`

  - **Purpose**: Sets additional properties for the quantum gate.

- `void SetGateType()`
  - **Purpose**: Sets the type of the quantum gate.
- `int GetGateType() const`
  - **Purpose**: Retrieves the type of the quantum gate.
  - **Returns**: The type of the quantum gate as an integer.
- `double GetFidelity() const`
  - **Purpose**: Retrieves the fidelity of the quantum gate.
  - **Returns**: The fidelity of the quantum gate as a double.
- `size_t GetGateSize() const`
  - **Purpose**: Retrieves the size of the quantum gate.
  - **Returns**: The number of qubits involved in the quantum gate.
- `std::string GetUnitary() const`
  - **Purpose**: Retrieves the unitary matrix representation of the quantum gate.
  - **Returns**: The unitary matrix as a string.
- `std::string GetName() const`
  - **Purpose**: Retrieves the name of the quantum gate.
  - **Returns**: The name of the quantum gate as a string.

A.6.4. Private Members

- `int type`
  - **Purpose**: Denotes the type of the quantum gate, helping to distinguish between different gates (e.g., ID, RZ, etc.).
- `std::string name`
  - **Purpose**: Stores the name of the quantum gate.
- `size_t gate_size`
  - **Purpose**: Stores the number of qubits involved in the quantum gate.
- `int gate_length`
  - **Purpose**: Stores the time needed to execute the gate operation.
- `std::string unitary`
  - **Purpose**: Stores the unitary matrix representation of the quantum gate.

- `double fidelity`
  - **Purpose**: Stores the fidelity of the quantum gate.
- `std::vector<std::vector<Qubit*» coupling_mapping`
  - **Purpose**: Stores the coupling mapping for the quantum gate.
- `std::vector<Component*> qubits`
  - **Purpose**: Stores the qubits involved in the quantum gate.
- `std::map<std::string, double> additional_properties`
  - **Purpose**: Stores additional properties of the quantum gate.

A.7. QdmiParser Interface

A.7.1. Declaration

```
class QdmiParser
{
public:
    QdmiParser();
    std::vector<QDMI_Device>get_available_backends();
    int get_num_qubits(QDMI_Device dev);
    void createAllQcTopo(Topology *topo);
    Topology createAllQcTopo();

    void createQcTopo(QuantumBackend *backend,QDMI_Device dev);
    QuantumBackend createQcTopo(QDMI_Device dev,int device_index = 0, std::
    stringdevice_name="");

    static void refreshQubitProperties(QDMI_Devicedev, Qubit *q)
    { //Refresh Qubit properties
    QDMI_Qubit qubits;
    int err
    err = QDMI_query_all_qubits(dev, &qubits)
    if (err != QDMI_SUCCESS || qubits == NULL){
        std::cout << "   [sys-sage].............Could not obtain available qubits via
    QDMI\n"
        return;
    }

    int qubit_index = q->GetId()
    // Refreshing Qubit coupling map
    int coupling_map_size;
    std::vector<int> coupling_mapping;
    coupling_map_size =(&qubits[qubit_index])>size_coupling_mapping;
    coupling_mapping.resize(coupling_map_size);
    std::copy(qubits[qubit_index].coupling_mappng, qubits[qubit_index].coupling_mapping +
    coupling_map_size,coupling_mapping.begin());
```

```
30      q->SetCouplingMapping(coupling_mapping,coupling_map_size); // Set all the qubit
        properties
31      int scope;
32      // Declare prop as a vector
33      std::vector<int> prop{QDMI_T1_TIME,QDMI_T2_TIME, QDMI_READOUT_ERROR,
        QDMI_READOUT_LENGTH};
34      std::array<std::string, 4> properties{"T1","T2", "readout_error", "readout_length"};
35
36      double value;
37      for (size_t i = 0; i < 4; ++i)
38      {
39          // QDMI_Qubit_property prop_index;
40          QDMI_Qubit_property prop_index = new(QDMI_Qubit_property_impl_t);
41
42          prop_index->name = prop[i];
43          int err =QDMI_query_qubit_property_exists(dev,&qubits[qubit_index], prop_index,&
        scope);
44          if(err)
45          {
46              std::cout << "   [syssage].............Queried property doesn't exist: "
47                  << i<<"\n";
48              continue;
49          }
50          err =QDMI_query_qubit_property_type(dev,&qubits[qubit_index], prop_index);
51          if(prop_index->type == QDMI_DOUBLE){
52              err =QDMI_query_qubit_property_d(dev,&qubits[qubit_index],
53                  prop_index,&value);
54              if(err){
55              std::cout << "   [syssage].............Unable to query property: "
56                  << i <<"\n";
57              continue;
58              }
59              delete prop_index;
60
61              q->SetProperties(qubits[qubit_index].t1,
62                  qubits[qubit_index].t2,
63                  qubits[qubit_index].readout_error,
64                  qubits[qubit_index].readout_length);
65      }
66  private:
67      int initiateSession();
68      static QInfo info;
69      static QDMI_Session session;
70      void getCouplingMapping(QDMI_Device dev,QDMI_Qubit qubit, std::vector<int>&
        coupling_mapping, int &coupling_map_size);
71      void getQubitProperties(QDMI_Device dev,QDMI_Qubit qubit);
72      void setQubits(QuantumBackend *backend,QDMI_Device dev);
73      void setGateSets(QuantumBackend *backend,QDMI_Device dev);
74
75  };
```

Listing A.7: QdmiParser

A.7.2. Constructor

- `QdmiParser::QdmiParser()`
  - **Purpose**: Initializes the `QdmiParser` object and initiates a QDMI session.

A.7.3. Methods

- **Session Initiation:**
  - `int QdmiParser::initiateSession()`
    - **Purpose**: Initiates a session with the QDMI library.
    - **Returns**: An integer representing the success or failure of the session initiation.
- **Retrieve Available Backends:**
  - `std::vector<QDMI_Device> QdmiParser::get_available_backends()`
    - **Purpose**: Retrieves a list of all available QDMI devices.
    - **Returns**: A vector of `QDMI_Device` objects representing the available devices.
- **Get Number of Qubits:**
  - `int QdmiParser::get_num_qubits(QDMI_Device dev)`
    - **Purpose**: Retrieves the number of qubits for a given QDMI device.
    - **Parameters**:
      - `QDMI_Device dev`: The device from which to retrieve the number of qubits.
    - **Returns**: An integer representing the number of qubits.
- **Get Coupling Mapping:**
  - `void QdmiParser::getCouplingMapping(QDMI_Device dev, QDMI_Qubit qubit, std::vector<int> &coupling_mapping, int &coupling_map_size)`
    - **Purpose**: Retrieves the coupling mapping for a specific qubit.
    - **Parameters**:
      - `QDMI_Device dev`: The device containing the qubit.
      - `QDMI_Qubit qubit`: The qubit for which the coupling mapping is retrieved.
      - `std::vector<int> \&coupling_mapping`: The vector to store the coupling mapping.
      - `int \&coupling_map_size`: The size of the coupling mapping.

- **Get Qubit Properties:**
  - `void QdmiParser::getQubitProperties(QDMI_Device dev, QDMI_Qubit qubit)`
    - **Purpose**: Retrieves various properties of a specific qubit.
    - **Parameters**:
      - `QDMI_Device dev`: The device containing the qubit.
      - `QDMI_Qubit qubit`: The qubit for which properties are retrieved.
- **Set Qubits:**
  - `void QdmiParser::setQubits(QuantumBackend *backend, QDMI_Device dev)`
    - **Purpose**: Sets the qubits for a specific quantum backend.
    - **Parameters**:
      - `QuantumBackend *backend`: The backend to which the qubits are set.
      - `QDMI_Device dev`: The device containing the qubits.
- **Set Gate Sets:**
  - `void QdmiParser::setGateSets(QuantumBackend *backend, QDMI_Device dev)`
    - **Purpose**: Sets the gate sets for a specific quantum backend.
    - **Parameters**:
      - `QuantumBackend *backend`: The backend to which the gate sets are set.
      - `QDMI_Device dev`: The device containing the gate sets.
- **Create All Quantum Computer Topologies:**
  - `void QdmiParser::createAllQcTopo(Topology *topo)`
    - **Purpose**: Creates the topology for all available quantum computers.
    - **Parameters**:
      - `Topology *topo`: The topology object to be populated.
  - `Topology QdmiParser::createAllQcTopo()`
    - **Purpose**: Creates and returns the topology for all available quantum computers.
    - **Returns**: A `Topology` object representing all available quantum computers.
- **Create Quantum Computer Topology:**
  - `void QdmiParser::createQcTopo(QuantumBackend *backend, QDMI_Device dev)`
    - **Purpose**: Creates the topology for a specific quantum computer.

* **Parameters**:
  - `QuantumBackend *backend`: The backend for which the topology is created.
  - `QDMI_Device dev`: The device containing the quantum computer.

– `QuantumBackend QdmiParser::createQcTopo(QDMI_Device dev, int device_index, std::string device_name)`

* **Purpose**: Creates and returns the topology for a specific quantum computer.
* **Parameters**:
  - `QDMI_Device dev`: The device containing the quantum computer.
  - `int device_index`: The index of the device.
  - `std::string device_name`: The name of the device.
* **Returns**: A `QuantumBackend` object representing the quantum computer.

A.8. QMAP Integration

```
#include "sys-sage.hpp"
#include <mqt-qmap>
#include <set>

using Edge        = std::pair<std::uint16_t, std::uint16_t>;
using CouplingMap = std::set<Edge>;
using QubitSubset = std::set<std::uint16_t>;

CouplingMap GetAllCouplingMaps(QuantumBackend backend) {
    CouplingMap result;
    for(auto i = 0; i < num_qubits; ++i) {
        Qubit* q = dynamic_cast<Qubit*>(GetChild(i));
        auto coupling_map = q->GetCouplingMapping();
        for (size_t j = 0; j < coupling_map.size(); ++j) {
            result.emplace(i, coupling_map[j]);
        }
    }
    return result;
}

CouplingMap GetFullyConnectedMap(const std::uint16_t nQubits) {
    CouplingMap result{};
    for (std::uint16_t q = 0; q < nQubits; ++q) {
        for (std::uint16_t p = q + 1; p < nQubits; ++p) {
            result.emplace(q, p);
            result.emplace(p, q);
        }
    }
```

```
29      return result;
30  }
31
32  void GetReducedCouplingMap(const QubitSubset& qubitChoice, CouplingMap& reducedMap) {
33      reducedMap.clear();
34      auto couplingMap = GetAllCouplingMaps();
35      for (const auto& [q0, q1] : couplingMap) {
36          if (qubitChoice.find(q0) != qubitChoice.end() && qubitChoice.find(q1) !=
        qubitChoice.end()) {
37              reducedMap.emplace(q0, q1);
38          }
39      }
40  }
41
42  QubitSubset GetQubitSet() {
43      QubitSubset qubitSet;
44      for (Component* qubit : children) {
45          if (Qubit* q = dynamic_cast<Qubit*>(qubit)) {
46              qubitSet.insert(q->GetId());
47          }
48      }
49      return qubitSet;
50  }
```

Listing A.8: External Interface for QMAP

List of Figures

List of Tables

Listings

Bibliography

[Bar+95] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. "Elementary gates for quantum computation." en. In: *Physical Review A* 52.5 (Nov. 1995), pp. 3457–3467. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.52.3457. URL: https://link.aps.org/doi/10.1103/PhysRevA.52.3457 (visited on 05/22/2024).

[Bro+10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications." In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Pisa: IEEE, Feb. 2010, pp. 180–186. ISBN: 978-1-4244-5672-7. DOI: 10.1109/PDP.2010.67. URL: http://ieeexplore.ieee.org/document/5452445/ (visited on 06/26/2024).

[Els+23] A. Elsharkawy, X.-T. M. To, P. Seitz, Y. Chen, Y. Stade, M. Geiger, Q. Huang, X. Guo, M. A. Ansari, C. B. Mendl, D. Kranzlmüller, and M. Schulz. "Integration of Quantum Accelerators with High Performance Computing – A Review of Quantum Programming Tools." In: (2023). Publisher: [object Object] Version Number: 2. DOI: 10.48550/ARXIV.2309.06167. URL: https://arxiv.org/abs/2309.06167 (visited on 05/17/2024).

[Ezr23] O. Ezratty. "Perspective on superconducting qubit quantum computing." en. In: *The European Physical Journal A* 59.5 (May 2023), p. 94. ISSN: 1434-601X. DOI: 10.1140/epja/s10050-023-01006-7. URL: https://link.springer.com/10.1140/epja/s10050-023-01006-7 (visited on 05/17/2024).

[HRB08] H. Haffner, C. Roos, and R. Blatt. "Quantum computing with trapped ions." en. In: *Physics Reports* 469.4 (Dec. 2008), pp. 155–203. ISSN: 03701573. DOI: 10.1016/j.physrep.2008.09.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0370157308003463 (visited on 05/22/2024).

[Hum+21] T. S. Humble, A. McCaskey, D. I. Lyakh, M. Gowrishankar, A. Frisch, and T. Monz. "Quantum Computers for High-Performance Computing." In: *IEEE Micro* 41.5 (Sept. 2021), pp. 15–23. ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2021.3099140. URL: https://ieeexplore.ieee.org/document/9537178/ (visited on 05/17/2024).

[Kja+20] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver. "Superconducting Qubits: Current State of Play." en. In: *Annual Review of Condensed Matter Physics* 11.1 (Mar. 2020), pp. 369–395. ISSN: 1947-5454, 1947-5462. DOI: 10.1146/annurev-conmatphys-031119-050605. URL: https://www.annualreviews.org/doi/10.1146/annurev-conmatphys-031119-050605 (visited on 05/22/2024).

[Lad+10] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien. "Quantum computers." en. In: *Nature* 464.7285 (Mar. 2010), pp. 45–53. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature08812. URL: https://www.nature.com/articles/nature08812 (visited on 05/22/2024).

[LP17] V. Lahtinen and J. Pachos. "A Short Introduction to Topological Quantum Computation." In: *SciPost Physics* 3.3 (Sept. 2017), p. 021. ISSN: 2542-4653. DOI: 10.21468/SciPostPhys.3.3.021. URL: https://scipost.org/10.21468/SciPostPhys.3.3.021 (visited on 05/22/2024).

[McK+18] D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, J. Gomez, M. Hush, A. Javadi-Abhari, D. Moreda, P. Nation, B. Paulovicks, E. Winston, C. J. Wood, J. Wootton, and J. M. Gambetta. "Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments." In: (2018). Publisher: [object Object] Version Number: 1. DOI: 10.48550/ARXIV.1809.03452. URL: https://arxiv.org/abs/1809.03452 (visited on 05/17/2024).

[NC12] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 1st ed. Cambridge University Press, June 2012. ISBN: 978-1-107-00217-3 978-0-511-97666-7. DOI: 10.1017/CBO9780511976667. URL: https://www.cambridge.org/core/product/identifier/9780511976667/type/book (visited on 05/22/2024).

[Pre18] J. Preskill. "Quantum Computing in the NISQ era and beyond." en. In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://quantum-journal.org/papers/q-2018-08-06-79/ (visited on 05/22/2024).

[Sch+23] M. Schulz, L. Schulz, M. Ruefenacht, and R. Wille. "Towards the Munich Quantum Software Stack: Enabling Efficient Access and Tool Support for Quantum Computers." In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Bellevue, WA, USA: IEEE, Sept. 2023, pp. 399–400. ISBN: 9798350343236. DOI: 10.1109/QCE57702.2023.10301. URL: https://ieeexplore.ieee.org/document/10313717/ (visited on 05/17/2024).

[Sei+23] P. Seitz, A. Elsharkawy, X.-T. M. To, and M. Schulz. "Toward a Unified Hybrid HPCQC Toolchain." In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Bellevue, WA, USA: IEEE, Sept. 2023, pp. 96–102. ISBN: 9798350343236. DOI: 10.1109/QCE57702.2023.10191.

URL: https://ieeexplore.ieee.org/document/10313648/ (visited on 05/17/2024).

[VS24] S. Vanecek and M. Schulz. "sys-sage: A Unified Representation of Dynamic Topologies & Attributes on HPC Systems." en. In: *Proceedings of the 38th ACM International Conference on Supercomputing*. Kyoto Japan: ACM, May 2024, pp. 363–375. ISBN: 9798400706103. DOI: 10.1145/3650200.3656627. URL: https://dl.acm.org/doi/10.1145/3650200.3656627 (visited on 06/26/2024).

[WB23] R. Wille and L. Burgholzer. "MQT QMAP: Efficient Quantum Circuit Mapping." en. In: *Proceedings of the 2023 International Symposium on Physical Design*. Virtual Event USA: ACM, Mar. 2023, pp. 198–204. ISBN: 978-1-4503-9978-4. DOI: 10.1145/3569052.3578928. URL: https://dl.acm.org/doi/10.1145/3569052.3578928 (visited on 05/17/2024).

[Win+23] K. Wintersperger, F. Dommert, T. Ehmer, A. Hoursanov, J. Klepsch, W. Mauerer, G. Reuber, T. Strohm, M. Yin, and S. Luber. "Neutral atom quantum computing hardware: performance and end-user perspective." en. In: *EPJ Quantum Technology* 10.1 (Dec. 2023), p. 32. ISSN: 2662-4400, 2196-0763. DOI: 10.1140/epjqt/s40507-023-00190-1. URL: https://epjquantumtechnology.springeropen.com/articles/10.1140/epjqt/s40507-023-00190-1 (visited on 05/17/2024).