A Conceptual Design: The Verification of QIR's Generic Quantum Optimization Passes

by

Paria Naghavi

B.ESc., Western University, Canada 2012 and

B.A., Western University, Canada 2012

A project Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Summary

This report outlines the design of a verification system for QIR quantum optimization passes using Vellvm interpreter, along with QWIRE semantics and memory model. The primary objective is to establish a foundation for future advancements in quantum optimization and verification techniques. The system focuses on a small circuit in a one-qubit system but can be scaled up for more complex analytics and optimization passes in QAT. The design includes three components: the QIR input, Vellvm parser, and verification tools of QWIRE. The input consists of the QIR quantum blocks before and after the optimization pass. The parser utilizes QWIRE semantics to map the QIR code to Coq objects and types, generating a Coq AST for analysis. The verification is performed by Coq's proof assistant, checking for well-formedness and equivalence of density matrices from the QIR code blocks corresponding to pre and post quantum optimization pass application. The design can be extended to other generic or targeted quantum transformations in QAT, including target-specific gate merging. The overall goal is to provide an agnostic and hybrid-compatible verification system that can improve the reliability of quantum computations on diverse hardware platforms.

Table of Contents

1. Introduction

The formal validity of quantum programs is an undeniable need of the current surge in quantum development frameworks and compilers. If the circuit intended for a quantum target is not passing the laws of quantum mechanics, the program will crash, and resources maybe wasted [1]. The laws governing quantum computation and consequent information processing approaches are very different from common classical computation. For example, classically we firmly rely on extracting and copying a value from one variable to another, with respect to the memory model defined by the programming language syntax. However, copying the state of a qubit is forbidden due to the No-Cloning Theorem. The act of measuring the quantum state, causes it to collapse to a classical bit. The No-Cloning Theorem is proven using techniques such as the linearity of quantum mechanics and the properties of matrices representing gates [2].

Building development frameworks, let alone a programming language to compute with emerging quantum devices, led to development of sizable projects. Type-safe programming languages such as LIQUi|> and QWIRE utilize proof-assistance to model and check specifications governing their target systems [1]. In such languages, the correctness of density matrices, representing the quantum state of the system, are verified to be well-typed. Likewise, gate's properties must satisfy the unital type, meaning that quantum circuits are reversible, and states are normalized. Such conditions aim to satisfy the statistical and quantum devices' requirements, with respect to laws of quantum mechanics.

Current quantum computers, known as Noisy Intermediate-Scale Quantum (NISQ) devices, have limited qubit coherence times and high error rates. These limitations make it challenging to perform lengthy and complex computations purely on quantum hardware [2]. Therefore, in recent years, classical – quantum hybrid computations have gained more popularity. They require the utilization of both classical and quantum resources to complete the computational model [3]. This approach allows for leveraging

the strengths of the classical paradigms to solve quantum problems and the other way around.

Classical preprocessing and post-processing are used to prepare inputs, analyze outputs, or interface with classical systems [4]. For example, to have a fault-tolerant computation, the error correction techniques on classical computers are applied after quantum computation is completed to present the result. Hybrid models can help reduce the resources needed for error correction and improve the overall reliability of quantum computations [5]. The quantum Random Access Machine (QRAM) framework allows for classical processors to interact with quantum registers [6]. This model was pioneered by Knill in 1986 and has been used ever since to demonstrate the ability of quantum computers to outperform classical computers in certain computational tasks. This ability is called quantum advantage and is due to the unique properties of quantum systems, such as superposition and entanglement. Achieving quantum advantage has the potential applications in fields like cryptography, optimization, and drug discovery. However, it requires addressing challenges in error mitigation, qubit coherence, scalability, and algorithm development.

Prior to execution, the code must be lowered to a format that is acceptable by the target machine, which concludes in the process of compilation. With the need for hybrid execution, hardware-agnostic compilers are required to bridge the abstraction gap between high-level development framework and the diverse hardware platforms [7]. Quantum devices have different native gate sets. These sets are the gates that have physical attributes on the hardware. In the majority of development frameworks, more gates and operations are available for the algorithm designer during the design and implementation of their algorithms than what the target machine offers [7]. Quantum computers come in various architectures, each with its own set of target profile and optimization techniques. Agnostic compilers help abstract away the hardware-specific details and provide a unified programming interface that facilitates interoperability. Quantum agnostic and hybrid compatible compilers are designed to work from multiple

development frameworks to target devices. An input program written in a preferred framework can be designed to target several quantum devices, as their target specifications are applied at lower level. This allows the researcher to abstract out the lower-level requirements and generate a hybrid algorithm without detailed considerations of each target.

At the target specific level of compilation, agnostic compilers optimize and map the quantum circuits to the target hardware. They analyze the structure of the quantum algorithm and match it with the capabilities of the hardware. This allows them to determine the most efficient mapping of operations onto the available qubits and gates. Moreover, the circuits can be optimized prior to the target-specific phase. For example, generic quantum circuit decomposition methods can reduce the number of gates during higher level intermediate phases and improve compilation time. Formal verification methods can be used to check for the fidelity of quantum transformation, ensuring that the programs execute correctly and effectively on the given hardware platform [1]. Additionally, agnostic compilers can offer future proving. As hardware is rapidly evolving, and new technologies and platforms are constantly being developed. Hybrid and agnostic compilers provide layers of abstraction that shield the programmers, from underlying changes in hardware architectures [8].

The Quantum Intermediate Representation (QIR) ecosystem provides an agnostic and hybrid compatible compilation supports for Pythonic and QIR-enabled development frameworks. QIR employs LLVM, an open-source compiler toolchain that utilizes a platform-independent representation of program code. This representation acts as a link between the front-end, which is specific to the source language, and the back end, which is specific to the target architecture, within the compiler. The LLVM based data types and tools allow for compiler native classical control at the IR level, whereas most quantum compilers add classical support to a purely quantum compiler [8]. The QIR ecosystem facilitates interoperability between several development frameworks and targets. Thus, transformations, analyzes, and optimizations can be shared among source

code frameworks and targets at the IR level. QIR's design allows for a broad spectrum of quantum and classical data types. Each target profile utilizes a subset of QIR specifications. The ecosystem is new and evolving and currently quantum optimization and transformation verification passes are not developed [21]. In this work, I design a verification process for QIR generic circuit decomposition optimization passes using Verified LLVM (Vellvm), an LLVM interpreter, and QWIRE.

Vellvm is a framework that uses formal verification techniques in Coq, to ensure the correctness of LLVM IR code. The adoption of Interaction Tree (ITree) and refinement theory establishes program equivalences through induction and elementary rewriting [9]. This model provides a foundation for defining "correct program transformations" that can be verified at various levels of abstraction, using the modular nature of the semantics. While Vellvm itself is not specifically designed for verifying QIR, it can be extended to support the verification of QIR code. This involves aligning Vellvm with QWIRE's formal semantics, for defining the rules for type systems and well-formedness to establish the correctness of quantum transformations and optimizations. This report presents a design prototype that leverages the Vellvm interpreter, together with the QWIRE language and proof tools, to verify the efficacy of elementary quantum optimization passes in the QIR.

2. Quantum Computation

The quantum circuit model, initially proposed by Richard Feynman in the 1980s, modernized the field of quantum computing by providing a new framework for harnessing the power of quantum mechanics [10]. Prior to the introduction of this model, quantum mechanical operations were simulated on classical machines, which had exponential costs and limitations. However, the quantum circuit model shifted the paradigm by enabling the direct implementation of particle behaviors and transformations in a "Quantum Computer" [11].

The advantages offered by the quantum circuit model have driven the progress in the development of large-scale, general-purpose quantum computers. By leveraging the

unique properties of quantum systems, such as superposition and entanglement; quantum computation offers the potential for exponential speedup in solving certain problems. While quantum simulation remains a significant area of research in quantum computing, the applications of quantum computers have expanded to various domains, including optimization problems and machine learning. Furthermore, the advancements in quantum computing have opened new possibilities and sparked innovation in the field. Researchers are actively exploring the potential of quantum algorithms and their applications in solving complex computational tasks more efficiently [12].

In quantum computing, the primary focus revolves around understanding and manipulating the transformations and behaviors exhibited by particles within a quantum system [13]. It is crucial to note that when the behavior of a particle is observed or measured, a fundamental principle of quantum mechanics comes into play: the irreversible destruction of the particle's quantum properties. This occurs because the act of measurement projects the quantum state of the particle onto one of its basis vectors, resulting in a classical outcome [2]. The irreversible nature of measurement stems from the no-cloning theorem, which asserts the impossibility of generating an identical copy of an arbitrary quantum state. Consequently, measurement transformations are irreversible processes that always yield classical data [14].

This inherent irreversibility of quantum measurements has significant implications for quantum computing. It highlights the fundamental distinction between classical and quantum information processing, where classical information can be freely copied and duplicated, while quantum information cannot be perfectly replicated due to the no-cloning theorem. As a result, quantum algorithms and computations must carefully navigate the balance between utilizing quantum superposition and entanglement to perform complex calculations, while respecting the limitations imposed by the irreversible nature of measurements. This understanding forms the bedrock of quantum information theory and serves as a guiding principle for designing quantum algorithms and verification systems [2].

The field of quantum computing relies on the notion of quantum gates, which are valid transformations applied to qubits. This approach is known as the Quantum Circuit Model, which draws conceptual inspiration from the classical Logical Circuit Model. In this model, quantum gates manipulate the quantum state of qubits, and users interact with quantum devices through programming languages that operate at different levels of abstraction. These languages provide varying degrees of functionality and mathematical frameworks. Like most classical programming languages that allow the direct manipulation of the binary state of a bit, languages based on the quantum circuit model provide users with the ability to access the quantum state of qubits.

For verifying the correctness of classical compilers, there are two common approaches: semantic preservation and equivalence checking [15]. Semantic preservation aims to demonstrate that the behavior of the target code aligns with the intentions of the source code. Equivalence checking first proves that the components of the source language are equivalent to each other, and then checking for equivalence between the source and IR or target code [15]. Equivalence can be observed or contextually checked, and it plays an important role in modular compilation. By adopting these verification techniques, quantum programming languages can ensure the reliability and correctness of quantum computations.

Proof engineering for quantum compilers face many challenges during the NISQ era, as the use of quantum computer hardware is expensive with limited numbers of qubits. Therefore, the current gate-model quantum computers do not support operations requiring large number of qubits. Quantum compilers employ distinct formal methods compared to classical compilers due to the unique challenges they encounter. The verification of classical compilers is concerned with proves that are irrelevant to their quantum counterparts, such as memory safety. While classical compiler verification methods are backed by years of research, the quantum compiler correctness is in its infancy [17].

3. QIR Ecosystem

QIR is an LLVM IR that adheres to specifications defined by the QIR Alliance. These specifications outline the standards and guidelines for representing quantum programs within the LLVM framework. The QIR ecosystem aims to provide interoperability for hybrid computation in the quantum computing field. It operates as a compiler system with front-end, middle, and back-end abstraction layers to transform source code into target-machine code. Figure 1 shows abstraction layers in QIR and their corresponding functionalities.

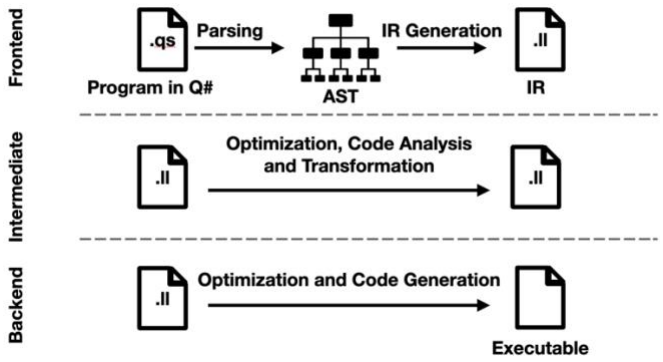

Figure 1. QIR Abstraction Layers

The front-end of QIR is responsible for parsing and translating source code from a compatible quantum development framework such as Qiskit, Q#, Cirq, QuTiP, etc., into the LLVM IR. The conversion of Pythonic source code to QIR can be achieved using the PyQIR APIs, acting as a bridge between the underlying Python quantum libraries of source and QIR. PyQIR is going to be discussed in more details in section 3.2. Additionally, other development frameworks like Q# integrate the capability of QIR generation into their compiler systems.

The middle-end performs various optimizations on the LLVM IR, such as dead code elimination, loop optimizations and inlining. The optimization of quantum gates can be applied in this phase. Finally, the QIR back end generates machine code for a specific target architecture, taking into account the characteristics of the underlying hardware, using profiles. The Quantum Adaptor Tool (QAT) is a component of the QIR ecosystem

that is responsible for lowering the generic QIR to the target specific one, while allowing for target specific optimizations.

The QIR ecosystem leverages the LLVM technologies and multi-level IR (MLIR) approach to facilitate compilation for different high-level languages and low-level targets. MLIR compilers take advantage of introducing new layers of abstraction in the middle phase, dissolving traditional compiler phases [8]. Tools and components that lower the instruction with MLIR approach can be reused for different source languages and target machines. It was initially introduced for in machine learning workflows to facilitate compilation for co-processing. MLIR spans the abstraction levels from after front end to machine code generation, creating a space for compiler extensibility and reusability with different high-level languages and low-level targets. It is an open-source project developed within LLVM that provides an infrastructure for building and optimizing compilers. The goal of MLIR is to enable the development of various compiler tools that can operate on programs at various levels of IR abstraction. It is designed to enable the representation of tools for high-level constructs like loops and functions down to low-level constructs like instruction generation [8][19].

The QIR specifications provide a wide range of descriptions for the intermediate representation of hybrid computation. It aims to establish a standardized format that can express programs from different quantum front ends and convert them into code for diverse quantum-classical architectures. The specifications define LLVM-based representations for classical and quantum data types and callables, enabling the construction of compiled hybrid programs from QIR compatible source compiler. Led by the QIR Alliance, the specification is designed to represent quantum programs within the LLVM MLIR framework. The Figure 2 summarizes the QIR ecosystem, and its components, relevant to this report.

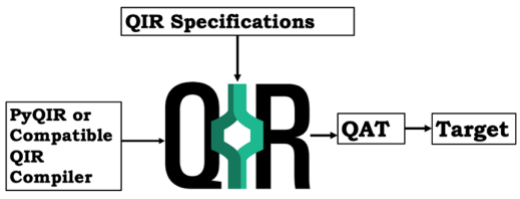

Figure 2. QIR Ecosystem

The motivation behind QIR is to enable language developers to write a single compiler and quantum computer developers to create a single code generator, allowing various quantum development frameworks to run on different hardware. The QIR Alliance seeks to foster the development of shared libraries for both quantum application development and quantum compiler development, providing a space for code reusability and collaboration.

3.1. QIR Specifications

QIR-specs provides a wide range of specifications for an intermediate representation of hybrid computation. Its purpose is to offer a common format that can be used to express programs from different quantum languages and convert them into code for various quantum-classical architectures. It defines a large set of LLVM-based representations for both classical and quantum data types, which can be used to build compiled quantum programs from a compatible source compiler. A subset of these specifications is used to define each target profiles, as QIR offers a broad spectrum of capabilities with respect to the current diversity of target machine architectures [21].

The representations are intended to be versatile enough to support the various quantum development platforms, and flexible enough to be converted into executable code for any quantum computer. The intermediate representation by QIR specification allows for code transformers and analyzers to operate at the IR level before the final target-specific code generation. Data type specifications are types of values and variables allowed by source compilers, during compilation process. Besides simple data types such as Int, Double, Bool, Pauli, and Range, other data types in QIR are represented as pointers to opaque LLVM structures. Declaring the data types as opaque pointers allows the compiler to create a placeholder for the type without needing to know its complete definition at that point. In the next section we examine a few of these data types provided by QIR-specs, such as %Qubit, %Result and %Array.

3.1.1. Qubit (%Qubit)

Qubits are denoted as references to an opaque LLVM structure type, %Qubit, using pointer representation. Qubits can be managed either statically or dynamically. If a value is known at compile time, the corresponding memory allocation for that value is static. This means that the memory is allocated and determined before the program is executed. On the other hand, if a value is not known until runtime, the memory allocation for that value is dynamic. Static qubits have target-specific identifiers known at compile time. They can be managed using inttoptr instruction. On the other hand, dynamic qubits allocation and tracking are managed during quantum runtime. During dynamic allocations, the instructions take an integer value as its input and returns a pointer value. The integer value is treated as an address that points to a memory location. The pointer value can then be used to access the contents of the memory location. Qubit values are integer identifiers that have been cast into a special type, making them distinguishable from normal integers. The only operation allowed on qubit values is to pass them to a function. The quantum execution functions provide the necessary mechanisms to allocate and release qubits [21].

3.1.2. Measurement (%Result)

Measurement outcomes are indicated through pointers to an opaque LLVM structure type, %Result. In hybrid computation, the result value can determine the next quantum or classical operations. Hybrid computation control flow requires the value result to dictate the following operations or termination. Currently, they are many quantum computing measurement approaches that vendors take advantage of. The forward declaration of measurement values allows for flexibility to serve a wide range of measurement approaches. Further, QIR's classical runtime utility functions provide tools to compare and presents null, zero, one and negative values of measurement results [21].

3.1.3. Arrays (%Array)

In QIR, arrays are passed as a pointer to an opaque LLVM structure called %Array. The representation of array data is decided by the runtime. Any array manipulations needed to access an item are done through corresponding runtime functions. Access to array elements is provided through byte pointers that the calling code needs to bitcast to the appropriate type.

Immutable arrays are supported, and modified copies can be created. If the existing array is not used after the generation of the modified copy, it is possible to avoid the copy and modify the existing array in place instead. Array slicing and projection can be used to construct new arrays that allow similar optimizations [21].

3.2. Front-End: PyQIR

PyQIR is a set of APIs designed to lower the pythonic source code such as QuTiP and Qiskit to QIR. It is not intended to be used for algorithm development, but rather by compiler front-end developers. By leveraging PyQIR, algorithm developers can write quantum programs in their favorite Pythonic quantum development framework, then utilize PyQIR to generate QIR, without having to directly interact with the low-level underlying LLVM implementation [22].

The current version of PyQIR module has functionalities such as pyqir.generator, pyqir.parser and pyqir.evaluator to lower and assess the Pythonic source code to QIR. With pyqir.generator.types, a wide range of data types and callables can be defined. To combine the benefits of dynamic execution and static compilation, PyQIR is supported by just-in-time (JIT) compilation infrastructure for evaluation and optimization purposes. Non-Pythonic development frameworks such as Q# implement the generation of QIR into their compiler systems.

3.3. Middle Phase: QIR

Whether the QIR is generated from PyQIR or a compatible compiler, the code is separated to into classical and quantum blocks. For example, Figure 3 shows the quantum teleportation algorithm has an if condition close to the end of its operation. The result from the condition determines the next quantum operation. Since the quantum device has no way of dealing with this simple control flow step, it separates the condition from the quantum circuit. The control flow graph (CFG) is constructed with different blocks for classical and quantum instructions during the IR phase. The Figure 3 blow shows the teleportation code in Qiskit on the left and the corresponding CFG on the right.

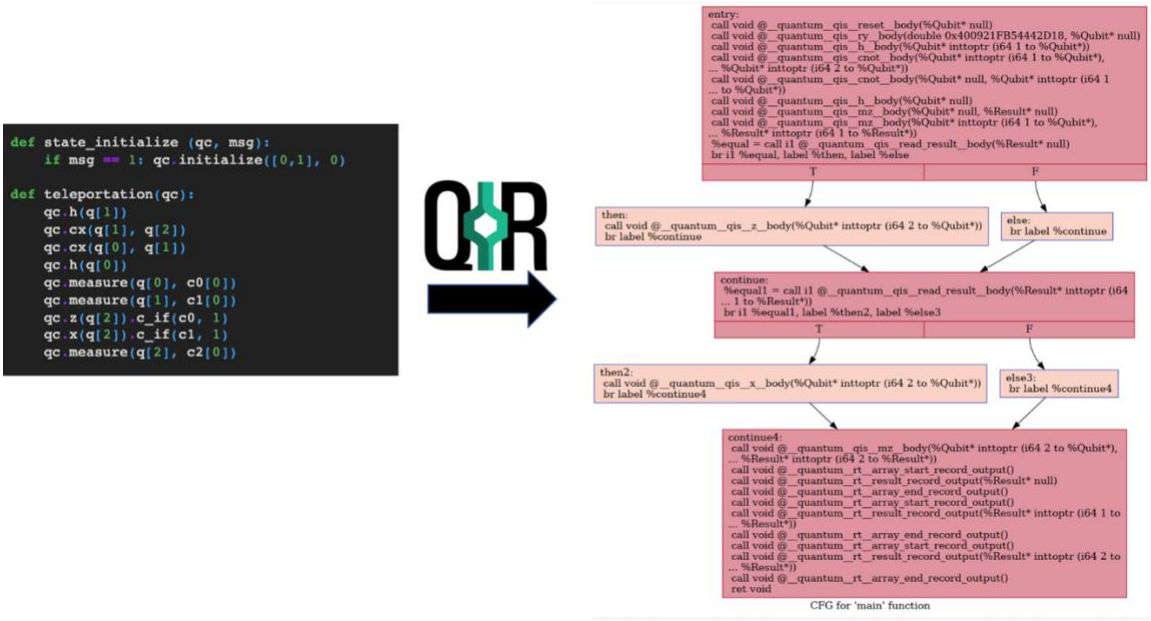

Figure3. Control Flow Graph of Teleportation in QIR

Hybrid and some quantum algorithms such as Variational Quantum Eigensolver (VQE) and Shor's algorithm require dynamic allocations of qubits. For example, in VQE, the classical optimization loop generates new quantum circuits that need to be executed on additional qubits. In the case of Shor's algorithm, the number of qubits required for the algorithm is related to the number of digits in the number being factored, which means that the allocation needs to be dynamic as the size of the input changes [2]. In these

algorithms, the number of qubits needed may not be known in advance, and the algorithm may require allocating qubits at runtime as it proceeds through the computation.

One of the key features of MLIR is its ability to capture both the high-level semantics of a program and the low-level details of the target hardware. This allows for more fine-grained optimizations and code generation tailored to specific hardware architectures. MLIR also supports incremental and compositional compilation, where optimizations and transformations can be applied at different stages of the compilation process [19].

3.4. Back-end: Profiles and Targets

Prior to profiling, QIR remains in a generic state. The specifications provided by qir-specs are general, allowing for a wide range of possibilities for quantum programs and hybrid features. A QIR profile describes a subset of the generic QIR functionality and conventions. A target profile describes the QIR specifications for a specific target [21]. Vendors provide requirements for their hardware and map the instruction to the target machines. A profile, along with the quantum instruction set (QIS), makes up a target that represents the architecture's capabilities. Various quantum platforms may impose distinct limitations on sequential gates, subroutines, branching, and measurement. These conditions are not reflected in the generic QIR and can be expressed by using profiles [21].

3.5. Quantum Adaptor Tool (QAT)

QAT is specifically designed to convert a generic QIR into a target-specific QIR that aligns with the requirements of a particular backend. This conversion process ensures compliance with the specified profile, while still enabling both generic and target-specific optimizations. As mentioned in the previous section, implementations of the QIR specification for a target machine will only utilize a subset of its features. These subsets may have additional constraints, such as dynamic allocation or handling qubits. Such a constrained subset is referred to as a "target"[23]. For example, early versions of

quantum backends may have limited classical instructions available. In this case, the vendor or user of the backend would define a profile that includes only a specified subset of capabilities.

QAT's architecture consists of three main tasks: loading the QIR, creating a generator and validator, and transforming and validating the QIR [23]. The generator performs transformations on the generic QIR to make it compliant with the selected profile, using a configuration with LLVM passes. The validator checks the compliance of a specialized QIR with the profile. The process involves passing the QIR through various components, which can install and execute different LLVM passes based on the configuration. QAT aims to preserve the modularity and extensibility of LLVM's opt tool while, adding the required capabilities for profile transformations and validation. The transformation component is highly configurable and performs replacements in the QIR using a custom replacer function [23].

The generic QIR supports dynamic qubit allocation as the default method for allocating qubits. However, there is a fundamental difference in how simulators and hardware backends handle qubit allocation. Simulators can dynamically allocate qubits as needed, while hardware backends have a limited number of qubits and lack the necessary logic for dynamic management. As a result, it becomes necessary to convert dynamic qubit management into static allocation to ensure compatibility with backends such as simulators.

QAT is a command line tool that allows for LLVM passes and lowering the code according to a profile. The target configurations can be defined entirely using the command line; however, it is recommended to be formatted in a .yaml file. The configuration has three sections. First, there is a high-level description of the adaptor pipeline, whether the target needs adaption or validation. Second, the specific adaptor for each component is identified. Lastly, the final QAT result undergoes validation against

the specified target. Validating the QIR tasks are checking that the QIR is compliant with the profile specification and that the QIR only uses the specified QIS [23].

Adaptors offer various transformation of the QIR according to specific requirements. The adaptor configuration settings include flags for applying adaptors, validating, and emitting human-readable LLVM IR. The validation process for the QIR encompasses two main tasks: target profile validation and target QIS validation. Target profile validation ensures compliance with classical behavior by verifying internal calls, handling of undefined values, proper usage of opcodes, correct handling of external calls, and adherence to pointer types. Conversely, target QIS validation is concerned with checking the presence of required qubits and results within the QIR, while also specifying the allowed QIS elements [23]

The adaptors in QAT serve different purposes in the transformation process, such as the option of optimization of the QIR using various LLVM passes. QAT's target mapping in the adaptor transforms the QIR to a targeted version based on the QIS, including mapping a dynamic qubit to a static allocation. The target profile mapping adaptor can perform additional optimizations after the final generic transformation. For example, the replacement linking adaptor conditionally replaces function calls based on the presence of replacement functions in the QIR [23].

3.5.1. Replacement Linking

Replacement linking is a process where one function call in QIR is replaced by another. For example, suppose there is a function that generates a sequence of H (Hadamard), X (Pauli-X), and H gates. This function can be replaced by another function that utilizes only a Z (Pauli-Z) gate. This replacement aims to optimize the circuit by finding an equivalent representation that achieves the desired outcome while potentially reducing the number of gates and resource requirements [24]. This provides a mechanism to address hardware compatibility issues as well general circuit decomposition optimizations. It enables the use of software implementations for unsupported gates and

ensures that code written at the frontend level can be executed on targets with different native gate sets, without the need for extensive modifications. Moreover, it allows for reducing the size of the circuit, which leads to efficient compilation and runtime. This report is intended to use replacement linking as a tool for quantum circuit decomposition optimization pass.

4. Formalization

Hoare logic can be used in a formal system for verifying the correctness of programs using pre- and post-conditions. It provides a framework for reasoning about program statements and their effects on program state. ITrees and monads support explicit handling of effects, modular reasoning, and state management, which can be an addition to the formal reasoning provided by Hoare logic for program verification. Different from Hoare logics, ITrees and monads offer a compositional and modular approach to verification. They provide a way to structure computations and handle effects explicitly, which can be useful for reasoning about complex program behaviors and interactions. While Hoare logic focuses on before and after-conditions to reason about program correctness, ITree and monads provide a compositional and modular approach to program verification.

4.1. Monads

Monad represents the computation as a step-by-step process to achieve a desired result, like a recipe. A sequence of steps that can be composed together to reason about computation. It encapsulates effects in programming language. A monad consists of a type constructor and two operations: "return", which takes a pure value and wraps it in the monad, and "bind", which takes a monad and a function that returns a new monad, and sequentially composes them [26]. Examples of monads used in Haskell include the Maybe monad for dealing with optional values, the List monad for collections, and the State monad for stateful computations. The Free monad refers to a particular way of defining a monad that allows for the composition of multiple effects. This means that you

can combine different operations such as reading of a file, writing to a database, or doing calculations into a single computation [27].

In functional programming, a monad provides a way to structure computations with side effects in a composable and controlled way. It ensures predictable and reliable execution of programs by encapsulating side effects within a monadic context. Monad's type constructors define the monadic type that represents the computation's context. For example, in the case of the Maybe monad, the type constructor represents the possibility of having a value or nothing. Monad's unit function takes a value and wraps it inside the monad, lifting it into the monadic context. This establishes the initial state of the computation. Lastly, bind operation allows for chain computations. It takes a monadic value, applies a function that produces a new monadic value, and returns the result. It enables sequential composition and facilitates passing values between computations. Further, monads provide a structured approach to handling side effects, such as state manipulation, I/O operations, or error handling. They ensure the purity of functional code by isolating side effects within the monadic context and enabling controlled execution [27].

A monad can be designed to incorporate the environment by including operations that interact with the environment. For example, a state monad is a specific kind of monad that manages and encapsulates state. It provides operations to read and update the state, based on the model of the environment variables. By using the state monad or similar monadic structures, we can have computations depending on the environment, while ensuring that the environment is managed and isolated within the monadic context. Majority of quantum programs contains circuits that manipulate quantum states and perform measurements on them. In this report, the state monad is considered to be employed to manage the state of the quantum system during the verification process. It provides operations for initializing, transforming, and measuring quantum states, ensuring proper state handling throughout the verification. By utilizing the state monad, we can reason about the behavior of quantum states and their transformations accurately.

Monads provide a powerful abstraction for structuring computations and capturing effects. In the context of QIR verification, monads allow explicit handling of computational effects, such as state changes and nondeterminism in dynamic allocations of qubits, and external interactions such as control flow. Quantum operations and measurements exhibit non-deterministic behavior, making it crucial to model and control these effects.

Quantum computations involve operations and transformations on quantum states, which can introduce various effects such as entanglement, superposition, measurement, and probabilistic outcomes. These effects need to be properly accounted for and managed to ensure the integrity of the quantum program's behavior. By utilizing monads, we can define specific monadic structures that encapsulate the effects of quantum computations. These monadic structures provide a mechanism to compose quantum operations, handle probabilistic outcomes, manage state changes, and propagate errors and exceptions that may occur during the computation. The use of monads in quantum programming languages or frameworks allows for a clear separation of the effects and behaviors of quantum computations, making the code more structured, readable, and maintainable.

4.2. ITree

ITrees are a formalism used in programming language semantics to capture the dynamic behavior of a program. They represent a computation as a tree of interactions between the program and its environment, where each node in the tree corresponds to a single interaction. Interactions refer to the exchanges that take place between a program and its environment during execution. In the context of ITrees, an interaction represents a single step in the computation, where the program performs some actions, and the environment responds with some effects [28].

These interactions can be internal or external, including function calls, callbacks, event handling, and inter-component communication. External interactions involve

input/output operations, communication with external devices, or interaction with the outside world. Each node represents a specific computation or effectful operation and holds associated state and metadata. Edges in an ITree represent the transition between different nodes. They take the control flow of the computation from one state to another. An edge connects nodes and represents the result of an operation that leads to the next state. In other words, nodes capture the individual states, while the edges connect these states and define the flow of control [28].

By leveraging ITrees and monads, specifically the identity monad or the maybe monad, we can capture program actions that do not yield effect responses. By utilizing these monads, program actions without effect responses can be represented within the computational structure. Quantum computations often involve complex interactions and operations where certain actions may not have an immediate observable effect or may produce uncertain outcomes. By incorporating monads into the computational structure, we can model and handle these scenarios, ensuring that the behavior of quantum programs is represented and accounted for.

ITrees is a data structure used in Coq proof assistance to formalize memory management. Unlike "inductive" definitions that build up a structure incrementally, coinductive definitions start with a potentially infinite structure and describe its properties in terms of the behavior of its substructures with infinite chains. One application of coinductive reasoning in the context of quantum computing is the definition of quantum states. Instead of defining a quantum state through sequential application of quantum gates, a coinductive approach represents the quantum state as an infinite superposition of basis states. For example, a qubit state can be defined as a superposition of the basis states $|0\rangle$ and $|1\rangle$, where the coefficients represent the probability amplitudes. Coinductive reasoning allows us to define properties of quantum states based on the behavior of their substructures, such as checking for entanglement between quantum states.

ITrees have the capability to be transformed into executable programs, enabling the execution of interaction models in real-world environments. This feature allows for the translation of an ITree representation of an interaction into a program that can perform the desired actions. Additionally, ITrees provide a means to define domains for constructing denotational semantics of programming languages within Coq. This allows for modularization of the effects of the semantics while still preserving executability. In the context of this design paper, the use of ITrees and monads revolves around QWIRE and Vellvm, which demonstrate the benefits of modularizing the effects of semantics while maintaining executability. QWIRE utilizes denotational semantics for managing its mathematical aspects, while Vellvm employs ITrees for the verification of its comprehensive memory model. The modularity of Vellvm facilitates the integration of QWIRE's semantics and QRAM memory model into the Vellvm interpreter.

4.3. Vellvm

Vellvm is a project focused on providing a formal semantics for a substantial subset of the LLVM IR. QIR, on the other hand, is an LLVM IR that adheres to the specifications of QIR. Formally, an ITree is a coinductive variant of the free monad, parameterized by a set of events E of kind Type → Type and a return type R. ITrees construct impure computations with using three operations for returning a pure value of type R (Ret r), representing a silent step (Tau t), and representing a visible event e followed by a continuation k (Vis e k) [28]. ITrees in Vellvm are a way to represent complex computations in a structured and organized way, using a combination of trees and monads. They are particularly useful when dealing with potentially infinite or non-terminating computations. ITrees support a rich equational theory of equivalences up-to-tau, i.e., up-to the weak bisimulation that observes the uninterpreted events performed by the computations, the pure values they returned, and their potential divergence. This notion of weak equivalence is central to the verification of correctness of program transformations [9].

In the context of QIR verification, Vellvm's ITree facilitates the modular specification and verification of quantum programs. It allows breaking down the verification process into smaller, more manageable components, each represented as an ITree node. These nodes can represent different aspects of QIR, such as gate operations, circuit transformations, and measurement outcomes, enabling a structured and modular verification approach.

4.3.1. Vellvm Interpreter

The significant advantage of using ITree's handlers is the ability to apply multiple handlers to the same events, enabling easy experimentation with different semantic features such as alternative memory models. This aspect makes it possible to define both the complete Verified IR (VIR) semantic model and an executable VIR interpreter, such as in Vellvm. The model accounts for nondeterminism by interpreting some events prepositionally while the executable interpreter specifies the nondeterminism, making it suitable for testing and debugging. The two semantics support most of the interpretation levels, facilitating the proof of the implementation's refinement of the model [9].

Vellvm's propositional semantics involve evaluating the meaning of logical propositions expressed in Vellvm programs. They refer to statements or conditions within the program that can be true or false. By using propositional semantics, we can reason about the behavior of Vellvm programs and verify their correctness. We can analyze the truth values assigned to propositions, trace them in CFG, and make assertions about their expected behavior [9].

Vellvm introduces a novel approach to formal semantics by using a monadic interpretation of ITrees. This allows for a more compositionality and modularity of language semantics, while still enabling the use of an executable interpreter. The formal semantics of Vellvm is in Coq proof assistant, and it handles many non-trivial language features of LLVM IR. The semantics is defined modularly in terms of event handlers,

including the handles for nondeterminism. The goal of Vellvm is to provide a platform for verified LLVM optimizations and compilers [9].

4.3.2. VIR

VIR provides a compositional, modular, and executable formal semantics for a realistic sequential subset of LLVM IR. Specifically, VIR's syntax is represented as ITrees with respect to different effects: local environment, stack, global identifiers, memory model, nondeterminism, external function calls, etc. The use of ITrees allows VIR to define a modular and compositional semantics for LLVM IR, where different aspects of the language semantics can be treated independently and combined together in a structured manner. The design of VIR incorporates the insights and advancements of modern proof engineering techniques, particularly ITree-based monadic semantics [9].

The syntax of VIR is represented as ITrees, with effects implemented using independent event handlers, based on algebraic effects. The semantic model of VIR is defined in terms of a fully propositional specification, capturing the nondeterministic aspects of the language. Additionally, an executable interpreter was implemented, sharing most of its code with the propositional semantics. This allows for the validation of the semantics by comparing it against the expected behaviors of LLVM IR over a set of test cases, simulated by Coq in Vellvm. The executable interpreter, Vellvm, also enables properties-based testing using tools like QuickChick [9].

The relational proof method is conducted by establishing relationships between different program states or executions and use formal logic to reason about these relations. This focuses on how different states or executions are related to each other. In terms of metatheory, the compositional semantics of VIR gives rise to a relational proof method that allows nondeterminism of the programs without the need for explicit simulation diagrams or coinduction. This proof infrastructure enables the definition and verification of correct program transformations at different levels of abstraction [9].

VIR is validated in two ways. First by a verified compiler targeting VIR such as Helix project. This demonstrates the utility of VIR and its metatheory for checking the compiler correctness. Second, the use of ITrees in VIR allows for the extraction of an executable interpreter that can be used to cross-validate against other LLVM IR implementations and run a larger end-to-end test [9].

4.4. Vellvm for Verification of QIR

Similar to VIR, verification for QIR can be implemented using formal and executable semantics for a subset of QIR in the Coq proof assistant. This allows us to reason and verify the behavior and correctness of QIR programs by defining the QIR related syntax and support for Vellvm intrinsics. To achieve this, we need to add constructs for qubits, measurement, gates, and hybrid control flow.

The core aspect of VIR is its use of ITrees and monadic semantics. ITrees are used as the core specification to define the behavior and effects of QIR computations. They provide a structured representation of interactions and effects in a program, such as the local environment, memory model, nondeterminism, and external function calls. Monads can be used to handle the effects and state transformations within QIR. The use of monads allows for explicit management of effects, such as quantum state manipulation, measurement outcomes, and probabilistic behavior. It enables precise control and reasoning about the behavior of QIR programs. By providing a formal semantics in Coq, Vellvm enables the verification of QIR programs using theorem proving and formal methods. It allows for the specification of properties and invariants that QIR programs are required to satisfy, and the use of proof techniques to establish the correctness of those properties.

The formalization of VIR in Coq also facilitates the development of a verification infrastructure for QIR. It enables the formulation and proof of correctness properties, termination-sensitive refinements, and program transformations. It allows for the

establishment of program correctness at different levels of abstraction, benefiting from the modularity and compositional nature of Vellvm.

5. QWIRE

QWIRE is a programming language that focuses on the separation of classical computation from quantum circuit descriptions. By syntactically distinguishing quantum data inside circuits from classical data, QWIRE enables developers to reason about quantum algorithms and circuits in a type-safe manner. This separation is supported by concepts from linear/nonlinear logic and the QRAM model for hybrid quantum computing. Type theory plays the main role in defining well-formed matrices, ensuring that quantum states, circuits, and gates adhere to the conditions dictated by quantum mechanical rules. With its emphasis on separation, type safety, and well-formedness, QWIRE provides a reliable and modular framework for quantum computing. The axiomatic approach of QWIRE, along with its focus on linearity, guarantees the soundness and consistency of the language. With its modular design and strong foundations, QWIRE offers a promising platform for developing and verifying quantum computing applications.

5.1. Type theory

Type theory is concerned with classifying expressions and values into different types and defining rules for how these types are allowed to interact with each other. Before we discuss type theory, let's consider its predecessor, set theory. In set theory, a set is a collection of distinct elements. It deals with concepts like membership, union, intersection, and cardinality. Due to several paradoxes without any resolutions, type theory was evolved by Bertrand Russell between 1902 and 1908. Type theory focuses on classifying objects based on their types and properties and defining rules on the interaction of types. It is more concerned with the structure and behavior of objects within a system which is described as its type, rather than its membership to a set or groups of sets [30][16].

In QWIRE, the logical framework is developed with valid representation of quantum states, circuits, gates according to the limitations dictated by quantum mechanical rules. Therefore, it utilizes type theory in defining well-formed matrices. For example, quantum gates are well-formed if they satisfy the unitary condition. A quantum gate is unitary if the matrix multiplication of it with its conjugate transpose equals the identity matrix. For example, for an arbitrary unitary gate U, must hold the below condition:

$$UU^{\dagger} = U^{\dagger}U = I \rightarrow U^{\dagger} = U^{-1}$$

This means that the gate is reversible and can be undone by applying its conjugate transpose. This property is essential for ensuring the reversibility of quantum computations, which is a fundamental requirement in quantum computing.

In QWIRE, quantum circuits are represented using a dedicated circuit language, and their validity is determined by a typing judgment. This judgment comprises the circuit itself, a context that includes the names and types of the input wires, and the output type of the circuit. By applying the typing judgment, we can determine whether the circuit and its gates are well-formed according to the specified rules. Additionally, type theory in QWIRE provides a formal framework for reasoning about the structure and behavior of quantum circuits, allowing for precise definition of types and ensuring their well-formedness [16].

5.2. Well-formedness

In formal methods and logic, well-formedness refers to the validity and consistency of statements and values within the described logical framework. Using type theory, the well-formed matrix types were defined for density matrices, qubit states, and gates. The density matrix represents the state of the quantum system. For example, the nature of quantum gates must be unitary. Meaning that, the arbitrary U gate has to satisfy the following condition: $U^{\dagger}=U^{-1}$. Otherwise, the proposed operation cannot be applied on a qubit. Further, the quantum states have a condition to be normalized. This refers to summation of squared probabilities of each state must equal to 1. This condition can also be incorporated into well-formed matrix type of quantum density matrices. QWIRE

uses type theory and defined well-formed types for matrices presenting gates and qubits' states to set such requirements. This involves checking the syntax for adherence to the rules of the logical system, for example the unitary condition for gates. By ensuring well-formedness, QWIRE maintains the integrity and correctness of quantum computations [16].

5.3. Axioms and Linearity

In logic, axioms are principal statements that used as starting points for deductive reasoning. They are assumed to be true without requiring proof. Axioms define the basic rules and properties of a logical system and provide a foundation for deriving theorems and reasoning. The relationship between axioms and linearity lies in the fact that linearity can be expressed and defined using axioms. Linear systems follow the principle of superposition, meaning that the output of the system is a linear combination of its inputs. In QWIRE language, the quantum circuit is separated from the classical instruction according to linearity. This will allow for quantum circuits to be described using linear types, while classical computations can benefit from using non-linear types [1][19].

The no-cloning theorem is a fundamental phenomenon in quantum mechanics that states it is impossible to create an identical copy of an arbitrary quantum state. In other words, the theory states that there is no universal quantum cloning machine that can perfectly replicate an arbitrary quantum state. Compared to classical computation, in which duplicating a value of a variable in the memory is almost free, this limitation is required to be enforced in quantum programming language. QWIRE uses linear type and static single assignment for quantum wires, which represent qubits. This ensures that qubits are used exactly once and prevents unintended duplication of the quantum state. This allows for the interaction of quantum and classical resources while maintaining linearity. However, any communication or interaction between quantum and classical resources must comply with the linear/non-linear constraints imposed by the type system [16].

The axiomatic approach to QWIRE ensures that the quantum circuit language is independent of the classical host language. This means that the behavior and properties of QWIRE circuits are defined in a way that is not influenced by the specifics of the classical host language used for implementation. This independency of QWIRE ensures that well-formed circuits in QWIRE will not encounter errors or unexpected behavior, with respect to no-cloning theorem at runtime. This property of QWIRE ensures its soundness, as it maintains consistency with its formal semantics and adheres to the fundamental principles of quantum mechanics. This design approach in QWIRE language made it adaptable to verification systems.

5.4. QWIRE Monads

In QWIRE monads are bounded with dynamic lifting protocols to verify the circuits' types [16]. Monads are used to model higher-order functions to construct typed proofs. The type wire refers to a qubit or a quantum bit or product of the two types [1]. Wires cannot be constructed outside the monads, which solidify the logical framework for type-dependent bonding of qubits, quantum state and gates [1].

QWIRE's Monad.v combines the type-dependent design of the language with the isolated Coq's Kernel to ensure the correctness of types. Although the no-cloning theorem and wire qubit types were integrated in the monads of QWIRE, the quantum mechanical domain specifications of the language are declared in the quantum module, found in QuantumLib, a dependency of QWIRE package. The semantics of QWIRE relevant to the circuit decomposition of QIR optimization passes are found in QuantumLib project, which supports QWIRE [31].

5.5. Incorporation of QWIRE in Vellvm Parser and Verification Tools

Type theory forms the foundation of QWIRE, allowing for precise classification of quantum circuits based on their types and properties. This ensures well-formedness and adherence to quantum mechanical rules. QWIRE utilizes type theory to define matrix

types for quantum states, gates, and density matrices, contributing to the reliability and correctness of hybrid quantum computing programs. QWIRE also employs axioms and linearity, separating quantum circuits from classical instructions and providing a language for quantum circuit verification. By incorporating QWIRE into Vellvm, the verification system can check quantum circuits before and after optimization passes, advancing quantum optimization techniques and verification methods. This conceptual design report focuses on integrating the two projects, leveraging their executability and formal methods techniques for the verification of QIR optimization passes.

6. Design

In this section, we explore the design of a verification system that utilizes the QIR quantum blocks and modified Vellvm parser based on QWIRE semantics to generate Coq Abstract Syntax Tree (AST) for equivalence checking. The primary design goal is to lay the groundwork for future advancements in quantum optimization techniques and verification methods. This design is focused on a small circuit in a one qubit system; However, it can be expanded to more complex analytics, and optimization passes in QAT. The following sections outline the different components of the design and the required pipeline. Figure 4 summarizes the design of this process.

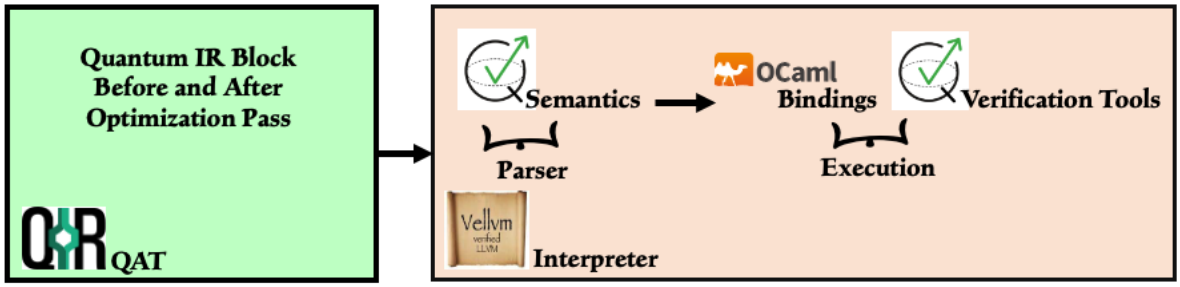

Figure 4– Design Overview: Verification on QAT passes using Vellvm and QWIRE

6.1. Input: QIR Blocks Before and After QAT Optimization Passes

The selection of representative QIR optimization passes within the scope of this study is a matter of feasibility, as the complexity of circuit decomposition escalates significantly with increasing number of qubits in the system. The QIR ecosystem is constantly

updating with some limited publicly available features. The current state of QAT does not offer any quantum optimization passes, but they are under development. In this project in addition to the equivalence checking system, I provided a design prototype for a future quantum optimization pass in QAT usually written in C++. This QAT pass will operate on the quantum block of QIR that are matching its requirements. The QIR quantum code blocks before and after the application of the pass are to be sent to the next step into Vellvm platform. Before the description of the prototypes, let us consider QAT's classical and quantum passes, their differences, and the class of quantum passes that this work focuses on.

QAT extensibility provides the capability to design and create optimization passes for quantum circuits. More specifically, QAT provides the ability to develop optimization passes for quantum circuits, while offering classical optimization and control flow using LLVM native passes such as O1, O2 and O3. These 3 optimization levels in LLVM determine the extent and aggressiveness of optimizations applied during compilation. O1 includes basic optimizations for improved code performance. O2 adds more transformations, including loop optimizations and function inlining. O3 applies the most aggressive optimizations, aiming for maximum code performance. An example of a classical optimization pass for QAT is the inline pass implemented by the QIR community [32].

Unlike the abundance of classical optimization passes in LLVM backed by decades of research and development, the order of application and the presence of quantum optimization passes in the QIR compilation process are still under discussion in the community. The main debate is whether these passes should be closer to the native gate set required by the hardware profile or closer to the front end of QIR, where general circuit decomposition optimization passes can be applied to reduce the number of gates and compilation time.

This work focuses on the latter approach and presents examples of generic circuit decomposition of one-qubit systems, such as replacing three gates with one. The replacement of these gates is the matter of matrix multiplication. For example, the sequence of Hadamard, Pauli-X and Hadamard gates (HXH) equals a Pauli-Z gate. The matrix representation of these gates are $H = 1/\sqrt{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$, $X=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ and $Z=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$. Their multiplication is as followed: $HXH= 1/2\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$. This relation is employed as a circuit decomposition technique, where HXH=Z. Circuit decomposition techniques aim to reduce the number of gates and resources needed for circuit execution, improving efficiency and resource utilization. Below are a few of such circuit decomposition examples:

HZH = X

HYH = -Y

XY = -YZ = iZ

YZ = -ZY = iY

ZX = -ZX =iY

XYZ = iI

During one of the QAT demos, the developers presented the replacement of a Z gate with an X gate sandwiched between two H gates [24]. The original description of the pass in C++ is not yet publicly available. I provided a prototype for this circuit decomposition optimization pass for QIR, which can be found in the appendices. In Appendix 1 the pseudocode is described, and Appendix 2 provides a sample C++ code for HXH optimization. The code scans the quantum code blocks of QIR to identify instances of the HXH sequence and replaces it with a Z gate.

The input of the verification system is the QIR code from before and after this optimization pass is applied. The before code in LLVM can be found in Appendix 3 and the after code in Appendix 4. These inputs yield the same density matrix when verified using QWIRE's verification tools. This equivalence is established by demonstrating the

well-formedness of the matrices and verifying their equality on both sides of the equation. The verification process ensures that the gate matrices are well-typed and that their values are consistent, confirming the equivalence between the two sides.

6.2. Parser: QWIRE Semantics

The purpose of the Vellvm parser is to lower the IR code to an ITree for the interpretation step. Currently, Vellvm does not offer any quantum formal semantics to map the QIR quantum code to an ITree. QWIRE semantics can close this gap. Incorporating of QWIRE semantics in Vellvm parser will allow to map the QIR code to the corresponding Coq objects and types. Specifically, by including QWIRE's datatypes and memory model in the parser in the current dev branch of Vellvm, we can generate bitcode that matches the QIR code to the QWIRE's verification tools requirements in the execution step of the designed verification system.

The Vellvm interpreter provides command line tools to print the Coq domination tree as an .ast file from IR inputs. The tree can be used as a design and development tool to check the accordance of the ITree, generated by Vellvm parser from the QIR code blocks. In other words, the AST can be used analytically to ensure that appropriate Coq objects have been assigned during the parsing process. The incorporation of QWIRE semantics and memory model is a fascinating task, but out of the scope of the design and more relevant to the development of the pipeline.

6.3. Verification

To perform equivalence checking, the Vellvm interpreter produces an abstract representation of the QIR quantum blocks, which can be printed as a Coq AST. OCaml's bindings in Vellvm traverse the ITree generated by the parser, performing the appropriate action for each event. Once executed, Coq provides tools for proofs and reasoning about the state of the code, density matrices and type-safety. Via QWIRE verification tools such Equations.v, the Coq proof assistant is used to check for well-

formedness and equivalence of density matrix from QIR code blocks corresponding to pre- and post-quantum optimization pass application [33].

QWIRE uses its type checking and inference mechanism to verify that the formal representations of both programs are well-formed and satisfies the typing rules of the language. Coq's type checker infers the types of expressions and checks that the type constraints are satisfied. Further, there is room for extensibility for new QWIRE tactics and other language constructs to guide the proof process. See below for expected Coq code that checks equivalence between before and after HXH optimization is applied.

```
Definition HXH : Box Qubit Qubit :=
  box_ q ⇒ _H $ _X $ _H $ q.

Lemma HXH_Z : HXH ≡ _Z.
Proof.
  matrix_denote.
  intros.
  Msimpl.
  restore_dims.
  rewrite σz_sa, σx_sa, hadamard_sa.
  repeat rewrite <- Mmult_assoc.
  rewrite (Mmult_assoc _ _ hadamard).
  rewrite (Mmult_assoc _ hadamard).
  rewrite <- (Mmult_assoc hadamard).
  replace (hadamard × σx × hadamard) with σz. easy.
  crunch_matrix.
Qed.
```

This code demonstrates a formal proof of the equivalence between the HXH circuit and the Pauli-Z gate in QWIRE. It was adapted from a similar proof in QWIRE's

Equation.v. It leverages reasoning and matrix manipulation tactics to establish the equivalence. The HXH circuit is constructed with the box_q lambda function that takes a qubit variable named "q" and applies the circuit. Next, the lemma, named HXH_Z, asserts that the HXH circuit is equivalent to the Pauli-Z (_Z) gate. The proof of this lemma started with the matrix_denote tactic, which initiates the manipulation of matrix representations of quantum states to obtain the corresponding density matrix for each side of the lemma. Several other tactics such as matrix simplification, restoring dimensions, and rewriting of matrix expressions are used to manipulate and rewrite the matrix representations of the gates and density matrix of the circuit. This concludes the proof of equivalence of before and after of circuit decomposition optimization of QIR code, using Vellvm interpreter and QWIRE language.

7. Conclusion

This report showcases the application of formal methods in quantum compilers, specifically through the use of QWIRE. The utilization of linear types ensures that quantum resources cannot be duplicated or cloned, in alignment with the non-cloning theorem in quantum mechanics. This formalization approach proves valuable in detecting potential vulnerabilities during the execution of QIR programs, emphasizing the significance of formal methods in ensuring the integrity of quantum computations.

The application of this prototype can be extended to other generic or targeted quantum transformations in QAT. For example, in target specific gate merging, the program state matrix arithmetic will allow for merging gates for purposes such as lowering to the native gate set of a specific hardware. Before and after gate merging's QIR code blocks can be inputted in the designed equivalence checking system to utilize the modified Vellvm and QWIRE language to prove that the source code intentions are preserved.

The verification process must be established for each optimization applied to the circuit. It is essential to assign the appropriate proofs for each transformation using a pipeline that connects QIR in Vellvm to QWIRE tools. This pipeline ensures that the correctness and integrity of the circuit are maintained throughout the optimization process. By

utilizing this systematic approach, we can effectively validate the transformations and ensure the preservation of the intended behavior of the circuit.

The systematic utilization of verification methods allows for the comprehensive assessment of the performance of the compiler and runtime systems. By defining appropriate evaluation metrics, such as compilation time, memory usage, gate count reduction, and circuit correctness, we can effectively measure the efficacy and effectiveness of the QAT passes. These quantitative metrics provide valuable insights for users in making informed decisions regarding the choice of front-end and back-end options. Through this systematic evaluation process, we can present an informative platform that facilitates decision-making and enables the optimization of compiler performance.

Glossary

CFG

A Control Flow Graph (CFG) is a graphical representation that illustrates the flow of control within a program. It consists of nodes representing program statements and edges denoting the possible transitions between these statements. CFGs are useful for analyzing and optimizing programs, as they provide insights into the program's control flow structure.

IR

Intermediate Representation (IR) serves as an intermediate step between the high-level source code and the machine code or executable output. The IR abstraction opens up the possibility of analysis, optimization and code generation, an abstraction level away from execution. LLVM IR is the form used to represent code in the QIR compiler projects such as QAT.

LLVM

LLVM is not an acronym. It is an open-source compiler infrastructure project. It provides a collection of modular and reusable compiler and tool chain technologies for various target and front ends. Its design allows for flexibility in performing various optimization and analysis techniques at the IR level. LLVM representation is in Static Single Assignment (SSA) form, by which variables are assigned exactly once throughout the program. It serves as a common representation for different development frameworks and target architectures, offering both general and target specific optimization, profiling, and analysis tools. The LLVM project is supported by decades of classical and heterogeneous compilation research. For this reason, the interoperable and modular design of QIR was based in LLVM and MLIR.

MLIR

MLIR stands for Multi-Level Intermediate Representation. It is an open-source project developed within LLVM that provides an infrastructure for building and optimizing compilers. The goal of MLIR is to enable the development of a wide range of compiler tools that can operate on programs at various levels of IR abstraction. It is designed to enable the representation of a wide range of tools for high-level constructs like loops and functions down to low-level constructs like instruction generation.
The motivation for developing MLIR arose from the observation that modern machine learning frameworks were constructed using different compilers, graph technologies, and runtime systems without a common infrastructure. This resulted in duplicated abstraction layers, tools and difficulties in generalizing the layers to support new hardware. There was a need for sharing frontend lowering infrastructure with common design principles. The MLIR project aimed to provide a mutual compilers infrastructure, allowing for analysis and optimization of computations as graphs. It uses the Static Single Assignment (SSA) form in the IR with the advantages of making dataflow analysis simple and with more efficient algorithms.

MLIR allows developers to write code in multiple levels of abstraction through the use of dialects. Each dialect defines its own set of high-level abstractions and optimizations that are specific to that domain. To optimize code, MLIR provides a set of generic optimization passes that can be applied to any dialect. These passes operate at a higher level of abstraction and are not specific to any particular dialect. For example, a common optimization pass is loop optimization, which looks for loops in the code and applies optimizations to make them faster. In contrast, dialect-specific optimization passes operate specifically for the dialect they are designed for. They are tailored to the specific abstractions and semantics of a given dialect, and thus can provide specific optimizations. The ability to represent multiple levels of abstraction for one IR incentivizes passes that operate across these levels.

For QIR, MLIR provides a modular framework that can be customized to target different hardware platforms. This allows QIR to be compiled to a wide range of quantum target machines, without having to write device-specific code. MLIR backed optimizations in QIR operate at multiple levels of abstraction, including dialect-specific optimizations, which can be tailored to the needs of hybrid computations.

QIS

A Quantum Instruction Set (QIS) refers to a collection of operations that can be performed on a quantum device. It consists of a set of quantum gates, runtime functions that manipulate quantum states and qubits and enable the execution of quantum computations. These instructions can include operations such as single-qubit gates (e.g., Pauli-X, Pauli-Y, Pauli-Z), multi-qubit gates (e.g., CNOT, SWAP), measurement operations, and other specialized operations used in quantum algorithms.

References

[1] R. Rand, J. Paykin, and S. Zdancewic, "QWIRE Practice: Formal Verification of Quantum Circuits in Coq," Electronic Proceedings in Theoretical Computer Science, vol. 266, pp. 119–132, Feb. 2018, doi: https://doi.org/10.4204/eptcs.266.8.

[2] M. A. Nielsen and I. L. Chuang, Quantum computation and quantum information. Cambridge: Cambridge University Press, 2000.

[3] A. Callison and N. Chancellor, "Hybrid quantum-classical algorithms in the noisy intermediate-scale quantum era and beyond," Physical Review A, vol. 106, no. 1, p. 010101, Jul. 2022, doi: https://doi.org/10.1103/PhysRevA.106.010101.

[4] R. LaRose et al., "Mitiq: A software package for error mitigation on noisy quantum computers," Quantum, vol. 6, p. 774, Aug. 2022, doi: https://doi.org/10.22331/q-2022-08-11-774.

[5] A. Paetznick and K. M. Svore, "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries," Quantum Information and Computation, vol. 14, no. 15 & 16, pp. 1277–1301, Nov. 2014, doi: https://doi.org/10.26421/qic14.15-16-2.

[6] E. Knill, "Quantum computing," Nature, vol. 463, no. 7280, pp. 441–443, Jan. 2010, doi: https://doi.org/10.1038/463441a.

[7] Mccaskey, Alexander, et al. "Extending C++ for Heterogeneous Quantum-Classical Computing." ACM Transactions on Quantum Computing, vol. 2, no. 2, July 2021, pp. 1–36, https://doi.org/10.1145/3462670. Accessed 8 Oct. 2021.

[8] Kaiser, Sarah, and Paria Naghavi. "What Is QIR?" QIR Book, Nov. 2022, github.com/qir-alliance/qir-book. Accessed 4 May 2023.

[9] Zakowski, Yannick, et al. “Modular, Compositional, and Executable Formal Semantics for LLVM IR.” Proceedings of the ACM on Programming Languages, vol. 5, no. ICFP, 19 Aug. 2021, pp. 1–30, https://doi.org/10.1145/3473572. Accessed 5 May 2023.

[10] Feynman, Richard P. “Simulating Physics with Computers.” International Journal of Theoretical Physics, vol. 21, no. 6-7, June 1982, pp. 467–488, https://doi.org/10.1007/bf02650179. Accessed 27 Aug. 2019.

[11] Feynman, Richard P. “Quantum Mechanical Computers.” Optics News, vol. 11, no. 2, 1 Feb. 1985, p. 11, https://doi.org/10.1364/on.11.2.000011. Accessed 12 Mar. 2020.

[12] Schuld, Maria, and Nathan Killoran. “Is Quantum Advantage the Right Goal for Quantum Machine Learning?” PRX Quantum, vol. 3, no. 3, 14 July 2022, https://doi.org/10.1103/prxquantum.3.030101. Accessed 17 July 2022.

[13] MARGOLUS, NORMAN. “Quantum Computation.” Annals of the New York Academy of Sciences, vol. 480, no. 1 New Technique, Dec. 1986, pp. 487–497, https://doi.org/10.1111/j.1749-6632.1986.tb12451.x. Accessed 20 Jan. 2021.

[14] Knill, E. “Conventions for Quantum Pseudocode.” Www.osti.gov, 1 June 1996, www.osti.gov/biblio/366453. Accessed 5 May 2023.

[15] Leroy, Xavier. “Formal Verification of a Realistic Compiler.” Communications of the ACM, vol. 52, no. 7, July 2009, pp. 107–115, https://doi.org/10.1145/1538788.1538814. Accessed 5 May 2023.

[16] Paykin, Jennifer, et al. “QWIRE: A Core Language for Quantum Circuits.” ACM SIGPLAN Notices, vol. 52, no. 1, Jan. 2017, pp. 846–858, https://doi.org/10.1145/3093333.3009894. Accessed 5 May 2023.

[17] Amy, Matthew. *Formal Methods in Quantum Circuit Design*. 2019. Available: https://uwspace.uwaterloo.ca/bitstream/handle/10012/14480/Amy_Matthew.pdf?sequence=5&isAllowed=y

[18] T. Nguyen, D. Lyakh, R. C. Pooser, T. S. Humble, T. Proctor, and M. Sarovar, “Quantum Circuit Transformations with a Multi-Level Intermediate Representation Compiler,” arXiv:2112.10677 [quant-ph], vol. 1, Dec. 2021, Accessed: May 05, 2023. [Online]. Available: https://arxiv.org/abs/2112.10677

[19] “MLIR,” mlir.llvm.org. https://mlir.llvm.org/ (accessed May 05, 2023).

[20] “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” llvm.org. https://llvm.org/pubs/2004-01-30-CGO-LLVM.html (accessed May 05, 2023).

[21] “qir-spec/specification/v0.1 at main · qir-alliance/qir-spec,” GitHub. https://github.com/qir-alliance/qir-spec/tree/main/specification/v0.1 (accessed May 05, 2023).

[22] “PyQIR documentation,” www.qir-alliance.org. https://www.qir-alliance.org/pyqir/ (accessed May 05, 2023).

[23] “QAT Documentation,” www.qir-alliance.org. https://www.qir-alliance.org/qat/ (accessed May 05, 2023).

[24] T. F. Rønnow, “qat/qir/demos/ReplaceLinking at main · qir-alliance/qat,” GitHub, May 22, 2022. https://github.com/qir-alliance/qat/tree/main/qir/demos/ReplaceLinking (accessed May 05, 2023).

[25] N. Grimm et al., “A Monadic Framework for Relational Verification Applied to Information Security, Program Equivalence, and Optimizations.” Accessed: May 05, 2023. [Online]. Available: https://arxiv.org/pdf/1703.00055.pdf

[26] J. Paykin and S. Zdancewic, “The linearity Monad,” Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Sep. 2017, doi: https://doi.org/10.1145/3122955.3122965.

[27] “Haskell/Understanding monads - Wikibooks, open books for an open world,” en.wikibooks.org. https://en.wikibooks.org/wiki/Haskell/Understanding_monads (accessed May 05, 2023).

[28] L. Xia et al., “Interaction trees: representing recursive and impure programs in Coq,” Proceedings of the ACM on Programming Languages, vol. 4, no. POPL, pp. 1–32, Dec. 2019, doi: https://doi.org/10.1145/3371119.

[29] G. Plotkin and J. Power, “Algebraic Operations and Generic Effects,” Applied Categorical Structures, vol. 11, no. 1, pp. 69–94, 2003, doi: https://doi.org/10.1023/a:1023064908962.
[30] Per Martin-Löf and G. Sambin, Intuitionistic Type Theory. 1984.

[31] The INQWIRE Developers, “INQWIRE QuantumLib,” GitHub, Jan. 01, 2022. https://github.com/inQWIRE/QuantumLib (accessed May 06, 2023).

[32] T. F. Rønnow, “QIR Adaptor Tool,” GitHub, Apr. 28, 2023. https://github.com/qir-

alliance/qat/blob/main/AdaptorExamples/InlinePassAdaptor/main.cpp (accessed May 08, 2023).

[33] K. Hietala, "QWIRE," GitHub, Mar. 27, 2023. https://github.com/inQWIRE/QWIRE/blob/master/Equations.v (accessed May 09, 2023).

Appendices

Appendix 1: Pseudocode for HXH=Z quantum circuit decomposition

```
// Iterate over the QIR instructions
for (each instruction in QIR) {
  // Check if the instruction is a sequence of H, X, and H gates
  if (instruction is HXH sequence) {
    // Get the qubit on which the HXH sequence operates
    qubit = get qubit from the HXH sequence

    // Remove the HXH gate sequence
    remove the HXH sequence instructions

    // Add a new Z gate instruction with the qubit
    add Z gate instruction with the qubit
  }
}
```

Appendix 2: Code in C++ for optimization pass to replace HXH gates with a Z gate

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/IRBuilder.h"

using namespace llvm;

struct HXHReplacementPass : public FunctionPass {
  static char ID;

  HXHReplacementPass() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    for (auto &BB : F) {
      for (auto &I : BB) {
        // Check if the instruction is a sequence of H, X, and H gates
        if (isHXHSequence(&I)) {
          // Get the qubit on which the HXH sequence operates
          Value *qubit = getQubitFromSequence(&I);

          // Remove the HXH sequence
          I.eraseFromParent();

          // Insert a new Z gate instruction with the qubit
          insertZGate(qubit);
        }
      }
    }
```

```
    return true;
  }

private:
  bool isHXHSequence(Instruction *I) {
    // Check if the instruction is an H gate
    if (isa<HInstruction>(I)) {
      // Check if the next instruction is an X gate
      Instruction *next = I->getNextNode();
      if (next && isa<XInstruction>(next)) {
        // Check if the next instruction after X is an H gate
        Instruction *nextNext = next->getNextNode();
        if (nextNext && isa<HInstruction>(nextNext)) {
          return true;
        }
      }
    }

    return false;
  }

  Value* getQubitFromSequence(Instruction *I) {
    // Assuming the qubit parameter is always the first operand
    return I->getOperand(0);
  }

  void insertZGate(Value *qubit) {
    // Create an IRBuilder and insert the Z gate instruction
    IRBuilder<> builder(qubit->getContext());
    builder.SetInsertPoint(qubit->getNextNode());
```

```
    builder.CreateCall(/* Z gate function */, { qubit });
  }
};

char HXHReplacementPass::ID = 0;

static RegisterPass<HXHReplacementPass> X("hxh-replacement", "HXH Replacement
Pass", false, false);
```

Appendix 3: The QIR code for circuit including H,X and H gate

```
; ModuleID = 'qir-circuit'
source_filename = "qir-circuit"

%Qubit = type opaque

declare void @__quantum__qis__h__body(%Qubit*)
declare void @__quantum__qis__x__body(%Qubit*)

define void @example() {
entry:
  %q = call %Qubit* @__quantum__rt__qubit_allocate()
  call void @__quantum__qis__h__body(%Qubit* %q)
  call void @__quantum__qis__x__body(%Qubit* %q)
  call void @__quantum__qis__h__body(%Qubit* %q)
  call void @__quantum__rt__qubit_release(%Qubit* %q)
  ret void
}
```

Appendix 4: After the Z gate replaced H, X and H

```
; ModuleID = 'library'
source_filename = "library"

%Qubit = type opaque

define void @soft_z(%Qubit* %q) {
entry:
  call void @__quantum__qis__z__body(%Qubit* %q)
  ret void
}

declare void @__quantum__qis__z__body(%Qubit*)
```