Rule-based hardware-configurable static analysis for quantum programs

Yi-Ting Chen*‡, Lauren Capelluto*, Ryan Shaffer*, Jeffrey Heckey†

*AWS Quantum Technologies, New York City, USA
†AWS Quantum Technologies, Seattle, USA
‡yitchen@amazon.com

***Abstract*—The rapid evolution of quantum hardware necessitates an adaptable static analysis framework for validating quantum programs. In this work, we introduce SHARP, a rule-based static analysis framework designed for OpenQASM that decouples hardware-specific constraints from the validation engine. By employing a rule-based approach, SHARP allows quantum computing services to validate programs against evolving instruction set architectures (ISAs) without having to modify the core analysis engine. SHARP achieves this by ingesting ISA descriptions as standalone data, making the validation engine resilient to hardware updates. Furthermore, the framework employs an analysis-driven validation mechanism that enables multi-stage validation, simplifying rule construction and enhancing extensibility. We demonstrate how SHARP supports hardware-specific validation tasks and discuss its impact on quantum software development, including language decoupling, program-level constraint validation, and feature access control. Our findings suggest that SHARP provides a path toward a scalable and maintainable programming interface for quantum computing.**

***Index Terms*—quantum computing service, static analysis, hardware-configurable validation, rule-based system, quantum software**

I. Introduction

Quantum computing hardware has evolved rapidly. Various architectures are actively being developed, including superconducting qubits [1]–[4], trapped ions [5]–[7], neutral atoms [8], [9], and photonic chips [10], each with unique operational characteristics and challenges [11], [12]. Furthermore, some devices expose low-level control of quantum hardware through pulse-programming [13], [14] which provides fine-grained control over the qubit state [15], [16]. Additionally, more advanced features such as mid-circuit measurement [17], classical feed-forward control [18], [19], error detection, and error mitigation are gaining support on more devices.

To innovate quantum applications, it is common for quantum program developers to directly program instructions specific to a particular device, using only the operations natively supported on that device, a type of quantum programming known as verbatim programming or native programming. Taking the native operations as the device's instruction set architecture (ISA), we can draw a comparison to classical computing. Unlike classical computing, where ISAs such as x86, ARM, and RISC-V have matured and stabilized [20]–[22], those for quantum hardware are undergoing rapid evolution. Each new generation may introduce novel instructions or even new data types, modify existing ones, or deprecate outdated operations, making it challenging to maintain a consistent static analysis framework. Furthermore, a quantum computing cloud service like Amazon Braket [23] can host access to diverse hardware. Compatibility issues may arise when software must support multiple ISAs simultaneously [24], [25].

In this work, we propose a framework that is stable and resilient against the rapid evolution of quantum ISAs. The framework performs the validation step of static analysis. We call this framework *Static Hardware-configurable Analysis with a Rule-based Protocol* or SHARP. Although the idea behind SHARP applies to any program representation, we focus on OpenQASM [26], [27] in this work. As shown in Fig. 1a, SHARP decouples the ISA description from the static analysis engine. The engine itself has no knowledge of specific hardware; instead, it consumes and processes the hardware description as a standalone piece of data at the time of validation, alongside the program being validated. When a device's existing ISA is modified or new features are supported, no changes are required in the static analysis engine. The onboarding of the feature is done by updating the standalone ISA description. SHARP simplifies the process of onboarding new features, not only reducing the overhead and effort required for implementation, but also making the static analysis layer of the compiler more stable.

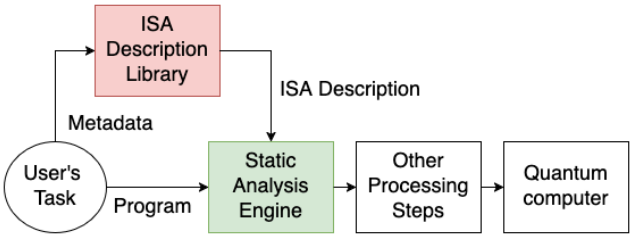

Fig. 1. Workflow of a service that uses SHARP. When a user submits a quantum task, which includes the quantum program and metadata such as the hardware they want to execute the program on, SHARP finds the ISA description of the hardware and feeds it to the static analysis engine. When adding a new feature, only the ISA description needs to be updated (colored red), while the engine (colored green) does not need to change with each feature addition.

The main contributions of this work are as follows:

- SHARP is a framework for static analysis of quantum programs that gains resiliency to changes in the quantum ISA by separating the ISA description from the core mechanism of static analysis.
- An ISA description is expressed by a rule-based system. The rules are evaluated by an inference engine against program elements when performing static analysis.
- The design decouples the language of the quantum program from the core logic of static analysis and the rule-based system in SHARP. An adapter plugin can be used to support new quantum programming languages, without re-implementing the framework.

II. Background

A. Static analysis against hardware constraints

Static analysis plays a crucial role in quantum computing services by ensuring program correctness before execution on quantum hardware [28], [29]. Given the scarcity and limited availability of quantum processors, early detection of program errors through static analysis enhances the user experience by preventing runtime failures [30]. Developers receive immediate feedback, reducing debugging time and improving efficiency [31], [32]. Moreover, static analysis serves as a safeguard, preventing unreasonable usage that could pose security risks [33], [34] or violate hardware constraints [35].

In the context of quantum computing, users can compose an abstract program and use a compiler to convert the program to a hardware ISA. At times, users program specifically to an ISA to have better control over how their programs are run. Therefore, for quantum programs, static analysis must validate not only the syntax and semantics of the input program, but also the hardware-specific constraints. For example,

```
rx(angle) qubit;
```

is an OpenQASM program instruction. The gate, `rx`, takes an angle as input and applies to a qubit. Given the definition of the valid data types for `angle` and `qubit`, validations for `rx` fit perfectly into grammar-based syntax checkers. However, a hardware ISA may impose additional constraints, such as the acceptable range of angles for each qubit, which is not normally captured by grammars. Therefore, a static analysis framework needs to be hardware-aware, by either hard-coding hardware restrictions or consuming them as input. The latter option makes the framework more flexible.

While quantum compilers like the Qiskit transpiler rely on predefined target data, such as qubit connectivity, it does not handle novel operations beyond its predefined scope [36], [37]. Similarly, Q# provides a "target profile" for compiling quantum programs to Quantum Intermediate Representation (QIR) [38], but its predefined selection of profiles limits flexibility, preventing developers from specifying arbitrary hardware constraints. For quantum static analysis, the target configurability needs to be generalized beyond a few fixed families of ISAs to accommodate the dynamic and evolving capabilities of quantum hardware.

B. Rule-based software systems

Rule-based systems have long been recognized as a powerful approach in software design [39], [40], particularly for tasks involving decision-making [41], validation [42], and expert knowledge representation [43]–[45]. The transparency and explainability of rule-based systems make them particularly suitable for applications where clear reasoning and consistent validation against the rules are required [41], [45].

In the context of program validation, a rule-based approach offers several advantages. By encoding the legal instructions for various ISAs as rules, the system can effectively validate user programs against these predefined criteria [46]. This design allows for extensibility by simply adding or modifying rules [43], making it suitable for supporting iterations of quantum hardware innovation.

Modern rule-based systems in classical programming further highlight the practicality of this paradigm. One prominent example is ESLint [47], a static code analysis tool for JavaScript. ESLint enables developers to define custom linting rules for enforcing code conventions and detecting issues, and its rules can be bundled into configurations and interpreted dynamically. Other systems such as Clang-Tidy for C++ [48] and Error Prone for Java [49] also utilize rule-based mechanisms for static analysis. These tools analyze abstract syntax trees (ASTs) and apply rules that flag violations or potential issues in source code. The success of these rule-based static analyzers in classical domains reinforces the viability of a similar design in quantum programming.

III. Rule-based static analysis

To address the challenges of constantly evolving hardware ISAs, SHARP performs ISA-aware static analysis with a rule-based system. Rules define constraints on program elements under an ISA, and the rules are targeted by a pass sequencing mechanism. The mechanism performs analysis passes before the main validation passes. Pass sequencing allows for the extensible and modular passes. It has been demonstrated to improve performance in code generation [50] and code optimization [51], [52]. In SHARP, pass sequencing enables multi-stage validation and allows for expressing complex validation logic with simple rules.

In the following, we use an example to introduce the working principles of SHARP. We conceptualize a quantum computer "QC1" whose ISA includes a single instruction that is complex enough to demonstrate the expressiveness of SHARP. Real-world hardware that inspires QC1 includes quantum systems from Rigetti and IBM. In this example, we want to validate the OpenQASM snippet below against the ISA of QC1.

```
// Example OpenQASM program
OPENQASM 3.0;
valid_gate1 $1;
raw_measure(R1) $1;
```

In this example program, we have one gate `valid_gate1` which is already supported by the system. The instruction, `raw_measure`, is a novel instruction. We will explore how SHARP performs validation on `raw_measure` in two steps:

1) An analysis pass traverses the AST of the program and annotates a node if it corresponds to a `raw_measure` instruction.
2) A validation pass enforces the ISA constraints for those nodes annotated as `raw_measure` while traversing the AST.

In the subsections below, we explain how to construct the constraints from the hardware as rules and then we explain how to sequence the rules and their corresponding actions to validate the `raw_measure` instruction.

A. Hardware constraints as SHARP rules

The raw measurement instruction returns the raw in-phase/quadrature (I/Q) readout values [53] on a qubit instead of the binary readout classification result of `0` or `1`. Assume QC1 has the following specification for the `raw_measure` instruction:

1) Constraint on qubits: It takes a qubit identifier as input. Only qubits `$1`, `$2`, and `$3` on QC1 support this instruction.
2) Constraint on readout resonators: It takes a resonator identifier as input. The two resonators on the device are labeled `R1` and `R2`. While `$1` physically connects to `R1` and `$2` to `R2`, `$3` connects to both `R1` and `R2`.
3) Return value type: It returns two real numbers for the I/Q readout values.

We will create a rule to represent this instruction specification. First, we define the `Rule` object, which is constructed with "attribute" and "kernel" arguments. We use the following pseudo-code syntax to express a `Rule` object. The same pseudo-code syntax is used throughout this paper.

```
rule = Rule(
    attribute=attribute,
    kernel=kernel
)
```

The "attribute" argument is an abstract object that defines the properties of a program element, such as the value of an input to a gate, to check against. The "kernel" argument defines how to check the attribute. For example, it may check whether an attribute is equal to a target value or whether each element of the attribute has specific types.

More complex constraints can be constructed by stacking `Rule` objects. Common ways to stack rules include using `ALL`, which returns true if all rules are true; `ANY`, which returns true if one of the rules is true; and `CONDITIONAL`, which evaluates the condition and then the rules if the condition is evaluated to be true. For brevity, we will refer to a `Rule` object and a stack of rules both as a "rule." Below, we construct a rule for each of the three constraints.

1) Constraint on qubits: The constraint on the qubits can be expressed as the following rule:

```
qubit_rule = Rule(
    attribute=Instruction.qubits,
    kernel=Kernel.in([$1, $2, $3])
)
```

2) Constraint on readout resonators: The constraint for the resonator is more complex, as it is a constraint between the two input arguments. It can be expressed as follows:

```
resonator_rule = ANY(
    AND(
        Rule(
            attribute=Instruction.qubits,
            kernel=Kernel.equal($1)
        ),
        Rule(
            attribute=Instruction.inputs,
            kernel=Kernel.equal(R1)
        )
    ),
    AND(
        Rule(
            attribute=Instruction.qubits,
            kernel=Kernel.equal($2)
        ),
        Rule(
            attribute=Instruction.inputs,
            kernel=Kernel.equal(R2)
        )
    ),
    AND(
        Rule(
            attribute=Instruction.qubits,
            kernel=Kernel.equal($3)
        ),
        Rule(
            attribute=Instruction.inputs,
            kernel=Kernel.in([R1, R2])
        )
    ),
)
```

3) Return value type: The hardware specification tells us what return type to expect. We can also represent this as a constraint to assert that developers are handling the return type correctly:

```
return_rule = ALL(
    Rule(
        attribute=Instruction.returns.length,
        kernel=Kernel.equal(2)
    ),
    Rule(
        attribute=Instruction.returns.type,
        kernel=Kernel.all_equal(FloatType)
    )
)
```

The overall specification for the `raw_measure` feature can be constructed by stacking the three rules that we have created, denoted as `val_rule` below.

```
raw_measure_rule = ALL(
    qubit_val_rule,
    return_val_rule,
    resonator_val_rule
)
```

B. Analysis-driven validation

Now we have a rule that can validate the `raw_measure` instruction. However, it is not the only rule for the program. Let's assume each instruction of QC1, e.g., `valid_gate1`, is subject to a rule. These rules should conditionally apply to specific types of instructions. The conditions for applying validation rules are also defined as rules. We call these conditions "analysis rules" to distinguish them from validation rules. Validation rules and analysis rules trigger different actions: a validation rule throws an error if the rule fails, while an analysis rule produces information about whether to apply a validation. A `Pass` is a construct used to encapsulate the rule-action pair, where the action only takes effect if the rules in the pass are evaluated to True.

```
pass = Pass(
    rule = rule,
    action = action,
)
```

In order to chain an analysis pass to a validation pass, the result of the analysis is stored in the AST nodes. The action of an analysis pass reads from a node, performs analysis using its rules, and then caches the analysis results back to the node. The information in the node determines the behavior of subsequent validation or analysis passes. On the other hand, a validation pass checks its rule against an AST node and throws an error if the node represents an invalid program element. A node contains a collection of key-value pairs, taking the form of:

```
ast_node_data = {
    key1: value1,
    key2: value2,
    ...
    keyK: valueK
}
```

Below, we construct the analysis and validation passes to validate `raw_measure` instructions.

1) Define the analysis rule: The validation rule we constructed in Section III-A should only target the `raw_measure` instructions. To construct the analysis pass that implements rule targeting, we first create the analysis rule which evaluates to True only for AST nodes that represent `raw_measure` instructions.

```
analysis_rule = ALL(
    Rule(
        attribute=Instruction.name,
        kernel=Kernel.equal("raw_measure")
    )
)
```

2) Define the analysis pass: The analysis pass action sets the `is_raw_measure` field in each AST node to True if the node satisfies the analysis rule.

```
analysis_pass = Pass(
    rule=ana_rule,
    action=set(
        ast_node_data,
        key="is_raw_measure",
        value=True
    )
)
```

3) Define the validation pass: We construct the validation pass conditioned on the truth values of `is_raw_measure` and `raw_measure_rule`. The first check is expressed as a rule:

```
condition_rule = Rule(
    attribute=ast_node_data.get("is_raw_measure"),
    kernel=Kernel.equal(True)
)
```

Together, the validation pass is

```
validation_pass = Pass(
    rule=CONDITIONAL(
        condition=condition_rule,
        rule=raw_measure_rule
    ),
    else_action=throw_exception
)
```

When an error is thrown, the error message from the triggered validation rule (e.g., optionally defined as a part of `qubit_val_rule`, `return_val_rule` and `resonator_val_rule`) can be included as part of the error, providing users with actionable information to modify their programs.

C. Apply passes to programs

A `PassSequence` is a construct that consists of passes in the form of:

```
pass_sequence = PassSequence(
    passes=[analysis_pass, validation_pass]
)
```

The `pass_sequence` defined above is a piece of data representing the validation of `raw_measure`. When applying `pass_sequence`, the inference engine in SHARP traverses the program AST and applies all the passes in the pass sequence sequentially to each visited node. Upon evaluating the rules in a pass sequence, the `attribute` and `kernel` arguments of the rules allow the inference engine to validate a specific constraint. The inference engine gathers information from the AST based on the `attribute` and applies checks on the attribute values based on the `kernel`. The results of the rules are then combined through the rule stacking structure into a final outcome of True or False. The outcome of the rules then triggers actions in the passes to perform the validation.

D. Adding and updating the feature

To add support for `raw_measure` on QC1 to the quantum computing service, we add `pass_sequence` as data to the SHARP ISA description of QC1.

Consider a case where, months later, the manufacturer of QC1 updates the hardware system and removes the resonator input variable (e.g., defaults `$3` to use `R1` and disallows user selection). In such a case, the SHARP framework only requires that the existing rule object `raw_measure_rule` is replaced with the object `raw_measure_rule_v2`, defined below.

```
raw_measure_rule_v2 = ALL(
    qubit_val_rule,
    return_val_rule
)
```

IV. Discussion

A. Language decoupling

In order to apply the kernel of a rule against an attribute of a program element, SHARP first needs to extract the attributes from the program element. This requires parsing the program and retrieving values from the properties of AST nodes. As a consequence, the overall workflow of SHARP must be dependent on the particular language in which the program is written.

In SHARP, the language dependency is decoupled by an attribute extraction layer, as illustrated in Fig. 2. All the dependencies on the language, such as the parser and the AST implementation, are confined to the attribute extraction layer. The extraction layer has a list of members, each representing the attributes we may want to validate. Each of them is associated with an extractor. When an extractor is applied, the extracted information is sent back to the rule, which then triggers the kernel to evaluate.

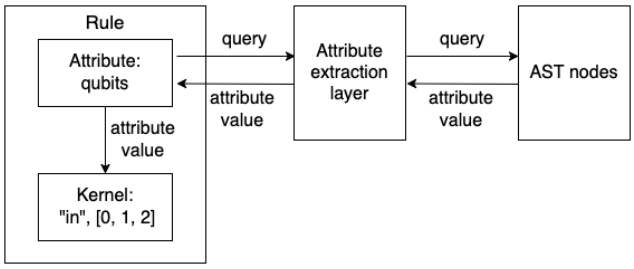

Fig. 2. Decoupling from the language and AST implementation.

For example, OpenQASM [27] supports a Python implementation of AST. The AST node for quantum gate calls is

```
@dataclass
class QuantumGate(QuantumStatement):
    modifiers: List[QuantumGateModifier]
    name: Identifier
    arguments: List[Expression]
    qubits: List[
        Union[IndexedIdentifier, Identifier]
    ]
    duration: Optional[Expression] = None
```

OpenQASM and QIR [38] may also implement parsers and ASTs in C++ which may have different names and types for the node and its attributes. Each attribute extraction layer maps attributes from a specific AST implementation to fields in SHARP based on their semantic meaning, such as mapping the `arguments` attribute in the above AST node implementation to `Instruction.inputs` in SHARP.

The part of the framework behind the extraction layer (i.e., rules, rule stacks, and passes) is language-agnostic. When targeting a different language or replacing the AST or the parser, we only need to implement a new attribute extraction layer. The rest of the framework can remain unchanged.

B. Program-level constraints

In Section III, we showed an example of applying SHARP to support a `raw_measure` instruction. The constraints for `raw_measure` are at the instruction level, and the validation is based on information from a single program element. In fact, SHARP can support constraints beyond the instruction level. In order to use program-level information for validation, SHARP stores program-level results of analysis passes. In Section III-B, the AST node data is introduced to contain information that is only accessible when visiting the node. In SHARP, there are two more data storage objects, differing in their life cycle and scope: (1) a global key-value table that is accessible by all passes and persists throughout the validation process, and (2) a pass key-value table that is only accessible to one pass and persists from the start to the end of the pass.

Below, we describe two examples of program-level constraints and explain how SHARP can support them.

1) Instruction count: In quantum hardware, a `measure` instruction typically takes a significantly longer time to execute compared to quantum gate instructions [54], [55]. If there are too many mid-circuit `measure` instructions in a program, qubits will decohere, and there will be no meaningful results. Therefore, a quantum computer can set a limit on the maximum number of `measure` instructions. Assume this limit is `M`.

This constraint can be supported by a pass sequence with two passes.

- An analysis pass that checks if an AST node corresponds to a `measure` instruction. If so, it increments the value in the `measurement_count` field of the pass key-value table by 1.
- A validation pass that checks if the value in the `measurement_count` field is greater than `M`. If so, it throws an error.

2) Order between instructions: Some instructions may only be used when another instruction already exists. For example, an instruction `classical_control` may only be valid if the control predicate is evaluated by a mid-circuit measurement, `mcm`. In such an example, a `mcm` instruction must precede a `classical_control` instruction.

This constraint can be supported by a pass sequence with two passes.

- An analysis pass that checks if an AST node corresponding to a `mcm` instruction with a specific predicate has been seen by the pass. If so, it sets the value in the `seen_mcm` field of the pass key-value table to True.
- A validation pass that checks if an AST node corresponds to a `classical_control` with the specific predicate. If it does, and if `seen_mcm` is False, it throws an error.

C. Feature access control

A quantum computing service can support feature access control at the account level, ensuring that access to specific devices or features is granted based on user roles. For example, experimental features can be restricted to users who opt

into beta programs. These users gain early access to new functionalities with the understanding that the features may be unstable, helping collect valuable feedback and enabling rapid iteration. In the framework of SHARP, because ISA descriptions are stored as standalone data, the management of feature access control is the management of data. When new access levels are added, they do not require adding logic or managing code paths in the underlying service components. SHARP enables a data-driven mechanism to regulate feature availability.

D. Performance and scalability

Quantum hardware typically supports a small set of native gates. For example, Rigetti Ankaa-3 supports three native gate types (Rx, Rz, and ISWAP), IQM Garnet supports two (PRx and CZ), and IonQ Aria-1 supports three (GPi, GPi2 and MS). Hardware constraints are generally tied to these native gates and a constant number of additional rules for control flow or device-level constraints. Thus, the number of rules grows linearly with the number of native gates, N_g. Given a quantum program with N_L lines of code, the inference engine in SHARP applies rule stacks to each relevant AST node. The resulting validation complexity is $\mathcal{O}(N_g N_L)$, as each line may be associated with one or more rules, depending on its instruction type.

To evaluate performance in practice, we benchmarked a Python implementation of SHARP on a MacBook M1. We used a rule stack representing the ISA of Rigetti Ankaa-2 which supports four native gate types (Rx, Rz, ISWAP, CZ). Each gate is associated with 3 to 5 constraints, totaling 15 leaf rules. SHARP validated OpenQASM programs at a rate of approximately 30 microseconds per line. In comparison, parsing the same set of programs on the same compute environment took roughly 200 microseconds per line, using the default parser of OpenQASM [27]. This indicates that rule-based validation introduces only a modest overhead relative to other stages.

Looking ahead, as programs grow beyond several hundred lines or rule sets become more complex, further optimization may be required. Several approaches are available to improve scalability:

1) High-performance implementation: SHARP is currently implemented in Python for flexibility. Rewriting performance-critical components in a compiled language such as C++ or Rust could significantly reduce runtime.
2) Rule stack compilation for solver-based inference: For very complex or interdependent constraints, the rule stack could be translated into logical formulas and validated using SAT or SMT solvers. This would allow SHARP to scale beyond simple pattern checks into formal constraint verification.
3) Pipelining and parallelization: Multiple quantum programs can be validated concurrently across threads or distributed systems. Furthermore, pipelining allows stages such as program ingestion, validation, QPU execution and result processing to overlap between different programs.

E. Applicability to fault-tolerant quantum computing

While SHARP is motivated by near-term quantum hardware with rapidly evolving ISAs, the framework is also applicable to fault-tolerant quantum computing (FTQC) systems. FTQC platforms are expected to implement logical operations over error-corrected qubits, often within a constrained set of fault-tolerant gate primitives (e.g., Clifford+T [56] or lattice surgery [57]). These logical-level constraints can vary depending on the underlying code (e.g., surface code, color code) or the architectural layout, and SHARP can encode such constraints as rules. Furthermore, as fault-tolerant protocols evolve to support more efficient logical operations or alternative decoding strategies, SHARP's rule-based mechanism allows validation logic to adapt without modifying the core engine. For example, enforcing distance-based scheduling rules, T-count limitations, or logical qubit layout constraints can be implemented as ISA rules, providing early validation of FTQC program suitability.

V. Conclusion

In this work, we introduced SHARP, a rule-based, hardware-configurable static analysis framework designed for quantum computing services. By decoupling hardware-specific validation from core static analysis functionalities, SHARP enables support for evolving quantum hardware without requiring extensive code modifications. Through a rule-based validation mechanism, the framework enforces program constraints while maintaining adaptability across different quantum ISAs.

Through its modular and analysis-driven architecture, SHARP supports instruction- and program-level validations and feature access control. The framework's design not only streamlines the onboarding of new hardware features but also promotes maintainability and extensibility for future quantum programming paradigms, including fault-tolerant systems. Additionally, the language-agnostic design of SHARP ensures compatibility across various quantum programming interfaces, thereby broadening its applicability within the quantum computing ecosystem.

Acknowledgment

The authors thank Abe Coull, Daniela Becker, Gandhi Ramu, Jacob Feldman, Cody Wang, Ramanathan Ramanathan, Shobhit Srivastava, Vivek Dwivedi and Yunong Shi for fruitful discussions.

References

[1] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, pp. 505–510, 10 2019.
[2] S. Storz *et al.*, "Loophole-free Bell inequality violation with superconducting circuits," *Nature*, vol. 617, pp. 265–270, 5 2023.
[3] E. A. Sete, W. J. Zeng, and C. T. Rigetti, "A functional architecture for scalable quantum computing," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 10 2016, pp. 1–6.
[4] L. Abdurakhimov *et al.*, "Technology and performance benchmarks of IQM's 20-qubit quantum computer," *arXiv e-prints*, p. arXiv:2408.12433, 8 2024.

[5] K. Wright *et al.*, "Benchmarking an 11-qubit quantum computer," *Nature Communications*, vol. 10, p. 5464, 11 2019.

[6] J. M. Pino *et al.*, "Demonstration of the trapped-ion quantum CCD computer architecture," *Nature*, vol. 592, pp. 209–213, 4 2021.

[7] I. Pogorelov *et al.*, "Compact ion-trap quantum computing demonstrator," *PRX Quantum*, vol. 2, p. 020343, 6 2021.

[8] J. Wurtz *et al.*, "Aquila: QuEra's 256-qubit neutral-atom quantum computer," *arXiv e-prints*, p. arXiv:2306.11727, 6 2023.

[9] L. Henriet *et al.*, "Quantum computing with neutral atoms," *Quantum*, vol. 4, p. 327, 9 2020.

[10] M. AbuGhanem, "Photonic quantum computers," *arXiv e-prints*, p. arXiv:2409.08229, 9 2024.

[11] N. M. Linke *et al.*, "Experimental comparison of two quantum computing architectures," *Proceedings of the National Academy of Sciences*, vol. 114, pp. 3305–3310, 3 2017.

[12] S. Blinov, B. Wu, and C. Monroe, "Comparison of cloud-based ion trap and superconducting quantum computer architectures," *AVS Quantum Science*, vol. 3, 9 2021.

[13] T. Alexander *et al.*, "Qiskit pulse: programming quantum computers through the cloud with pulses," *Quantum Science and Technology*, vol. 5, p. 044006, 8 2020.

[14] K. N. Smith *et al.*, "Programming physical quantum systems with pulse-level control," *Frontiers in Physics*, vol. 10, 8 2022.

[15] M. Kuzmanović, I. Björkman, J. J. McCord, S. Dogra, and G. S. Paraoanu, "High-fidelity robust qubit control by phase-modulated pulses," *Physical Review Research*, vol. 6, p. 013188, 2 2024.

[16] R. de Keijzer, O. Tse, and S. Kokkelmans, "Pulse based variational quantum optimal control for hybrid quantum computing," *Quantum*, vol. 7, p. 908, 1 2023.

[17] J. P. T. Stenger, C. S. Hellberg, and D. Gunlycke, "Preparing quantum statistical ensembles using mid-circuit measurements," *Quantum Information Processing*, vol. 23, p. 219, 5 2024.

[18] M. DeCross, E. Chertkov, M. Kohagen, and M. Foss-Feig, "Qubit-reuse compilation with mid-circuit measurement and reset," *Physical Review X*, vol. 13, p. 041057, 12 2023.

[19] M. Foss-Feig *et al.*, "Experimental demonstration of the advantage of adaptive quantum circuits," *arXiv e-prints*, p. arXiv:2302.03029, 2 2023.

[20] A. Waterman, "Design of the RISC-V instruction set architecture," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:63861396

[21] B. W. Mezger, D. A. Santos, L. Dilillo, C. A. Zeferino, and D. R. Melo, "A survey of the RISC-V architecture software support," *IEEE Access*, vol. 10, pp. 51 394–51 411, 2022.

[22] W. Ali, "Exploring instruction set architectural variations: x86, ARM, and RISC-V in compute-intensive applications," 8 2023.

[23] Braket contributors, "Amazon Braket: a fully managed quantum computing service provided by AWS," 2019. [Online]. Available: https://aws.amazon.com/braket

[24] L. Huang, J. Zhang, L. Yang, S. Ma, Y. Wang, and Y. Cheng, "RVAM16: a low-cost multiple-ISA processor based on RISC-V and ARM Thumb," *Frontiers of Computer Science*, vol. 19, p. 191103, 1 2025.

[25] Y. Cheng, L. Huang, Y. Cui, and Y. Wang, "Efficient multiple-ISA embedded processor core design based on RISC-V," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:252082856

[26] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv e-prints*, p. arXiv:1707.03429, 7 2017.

[27] A. Cross *et al.*, "OpenQASM 3: a broader and deeper quantum assembly language," *ACM Transactions on Quantum Computing*, vol. 3, pp. 1–50, 9 2022.

[28] M. Paltenghi and M. Pradel, "Analyzing quantum programs with LintQ: a static analysis framework for Qiskit," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 2144–2166, 7 2024.

[29] S. Ali, T. Yue, and R. Abreu, "When software engineering meets quantum computing," *Communications of the ACM*, vol. 65, pp. 84–88, 4 2022.

[30] M. Roberson and C. Boyapati, "A static analysis for automatic detection of atomicity violations in Java programs," 2 2010.

[31] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 6 2007, pp. 1–8.

[32] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 4 2009, pp. 141–150.

[33] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, "An empirical study of static analysis tools for secure code review," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 9 2024, pp. 691–703.

[34] J. Zhu, J. Xie, H. R. Lipford, and B. Chu, "Supporting secure programming in web applications through interactive static analysis," *Journal of Advanced Research*, vol. 5, pp. 449–462, 7 2014.

[35] J. McMahan, M. Christensen, K. Dewey, B. Hardekopf, and T. Sherwood, "Bouncer: static program analysis in hardware," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 711–722.

[36] A. Javadi-Abhari *et al.*, "Quantum computing with Qiskit," 2024.

[37] M. D. Stefano, D. D. Nucci, F. Palomba, and A. D. Lucia, "An empirical study into the effects of transpilation on quantum circuit smells," *Empirical Software Engineering*, vol. 29, p. 61, 5 2024.

[38] K. Svore *et al.*, "Q#: enabling scalable quantum computing and development with a high-level DSL," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2 2018. [Online]. Available: http://dx.doi.org/10.1145/3183895.3183901

[39] F. Hayes-Roth, "Rule-based systems," *Communications of the ACM*, vol. 28, pp. 921–932, 9 1985.

[40] A. Ligêza, *Logical Foundations for Rule-Based Systems*. Springer Berlin Heidelberg, 2006, vol. 11.

[41] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial Intelligence*, vol. 267, pp. 1–38, 2 2019.

[42] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, 12 2004.

[43] P. Jackson, *Introduction to expert systems*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 1998.

[44] J. C. Giarratano and G. D. Riley, *Expert systems: principles and programming*. Brooks/Cole Publishing Co., 2005.

[45] B. Buchanan and E. Shortliffe, *Rule-based expert system – the MYCIN experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 2 1984.

[46] A. Moller and M. I. Schwartzbach, "Static program analysis," 10 2018.

[47] N. C. Zakas, "ESLint: pluggable JavaScript linter," 2013. [Online]. Available: https://eslint.org

[48] LLVM Project, "Clang-tidy: a clang-based c++ "linter" tool," 2025. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[49] Google, "Error Prone: a static analysis tool for Java," 2015. [Online]. Available: https://github.com/google/error-prone

[50] M. Kahla, "Automatic source to source code transformation to pass compiler optimization," Ph.D. dissertation, 2023.

[51] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: an expressive annotation-directed dynamic compiler for C," *Theoretical Computer Science*, vol. 248, pp. 147–199, 10 2000.

[52] S. Abeysinghe, A. Xhebraj, and T. Rompf, "Flan: an expressive and efficient Datalog compiler for program analysis," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 2577–2609, 1 2024.

[53] J. Heinsoo *et al.*, "Rapid high-fidelity multiplexed readout of superconducting qubits," *Physical Review Applied*, vol. 10, p. 034040, 9 2018.

[54] M. Dupont, T. Oberoi, and B. Sundar, "Optimization via quantum preconditioning," *arXiv e-prints*, p. arXiv:2502.18570, 2 2025.

[55] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, "A quantum-classical cloud platform optimized for variational hybrid algorithms," *Quantum Science and Technology*, vol. 5, p. 024003, 4 2020.

[56] S. Bravyi and A. Kitaev, "Universal quantum computation with ideal Clifford gates and noisy ancillas," *Physical Review A*, vol. 71, no. 2, p. 022316, 2005.

[57] D. Litinski, "A game of surface codes: large-scale quantum computing with lattice surgery," *Quantum*, vol. 3, p. 128, 2019.