Quantum Compilation Process: A Survey

F. Javier Cardama[1(✉)], Jorge Vázquez-Pérez[1], Tomás F. Pena[1,2], Juan C. Pichel[1,2], and Andrés Gómez[3]

[1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain
{javier.cardama,jorgevazquez.perez,tf.pena,juancarlos.pichel}@usc.es
[2] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain
[3] Galicia Supercomputing Center (CESGA), Avda. de Vigo S/N, 15705 Santiago de Compostela, Spain
andres.gomez.tato@cesga.es

Abstract. Quantum compilation, critical for bridging high-level quantum programming and physical hardware, faces unique challenges distinct from classical compilation. As quantum computing advances, scalable and efficient quantum compilation methods become necessary. This paper surveys the landscape of quantum compilation, detailing the processes of qubit mapping and circuit optimization, and emphasizing the need for integration with classical computing to harness quantum advantages. Techniques such as Variational Quantum Eigensolver (VQE) exemplify hybrid approaches, highlighting the potential synergy between quantum and classical systems. It is concluded that, while quantum compilation retains many classic methodologies, it introduces novel complexities and opportunities for optimization and verification, essential for the evolving field of quantum computing.

Keywords: quantum computing · compilation process · quantum languages · qubit allocation

1 Introduction

With the current growth in the field of quantum computing in recent years, the need for efficient quantum compilation becomes increasingly apparent as researchers strive to harness the potential of quantum processors. Quantum compilation, like its classic counterpart, plays a pivotal role in bridging the gap between high-level quantum programming languages and physical quantum hardware. However, as quantum computing systems continue to evolve, the demand for scalable and efficient quantum compilation methods arises. This paper delves into the realm of quantum compilation, exploring techniques for mapping qubits to clusters, optimising quantum circuits for parallel execution, and managing the complexity of large-scale quantum programs. The principal aim is to organise the current information pertaining to quantum compilation and guide it into a scheme similar to classic compilation, facilitating a

S. Caino-Lores et al. (Eds.): Euro-Par 2024, LNCS 15385, pp. 100–112, 2025.
https://doi.org/10.1007/978-3-031-90200-0_9

comprehensive understanding of the quantum computing landscape. However, within this survey, an examination of quantum computing interpreters is omitted because they are not considered to be of interest in terms of efficiency. This is attributed to the fact that each quantum circuit performs a predetermined number of shots. Consequently, the contention is that it is more efficient to compile and execute the circuit N times rather than interpreting it N times.

Section 2 will first give a brief introduction to the modular functioning of a classic compiler, and then, in Sect. 3, we will discuss the main novelties introduced by quantum compilation and how these can be integrated into a compiler scheme. Finally, Sect. 4 shows the conclusions drawn from this survey.

2 Classic Compilation

Compilers are crucial in software engineering, translating high-level programming languages into machine code and optimizing machine resources. They follow a structured process involving analysis and synthesis stages, as depicted in Fig. 1. The emergence of quantum computing has extended these principles to quantum compilers, which adapt these phases for quantum programming languages, highlighting the enduring relevance of classic compiler methodologies in new computing paradigms [4].

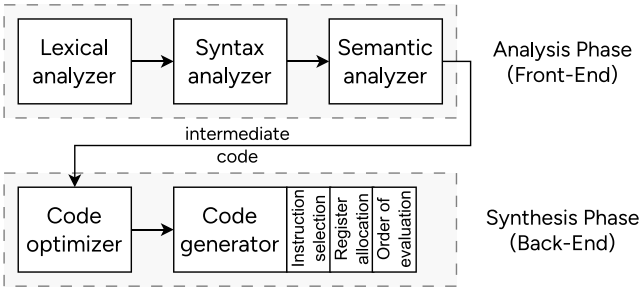


Fig. 1. Sequential phases of classic compiler process - analysis and synthesis stages.

The analysis phase marks the start of the compilation process, focusing on lexical, syntactical, and semantic evaluations to detect errors and form an intermediate representation. It includes tokenization, syntax tree construction, and semantic analysis like type-checking. The front-end generates an Intermediate Representation (IR), crucial for compiler efficiency and adaptability across various languages. This IR facilitates the compilation of languages like C++, Rust, and Java into popular Instruction Set Architectures (ISAs), enhancing code efficiency and quality through machine-independent optimizations.

The final phase in the compilation scheme, the code generator, is crucial as it adapts the code to the target machine's architecture, highlighting significant

differences between classic and quantum computing. This phase ensures the target program adheres to the semantics of the source program while optimizing resource utilization on the target machine. Key tasks included in the code generator phase include the following: instruction selection, register allocation and order of evaluation.

Instruction selection, which chooses the appropriate machine instructions or quantum logic gates, depending on whether the target is a classic or quantum computer; **register allocation**, crucial for managing limited computational resources like registers in classic computing or qubits in quantum computing; and **order of evaluation**, which determines the most efficient execution sequence for independent instructions.

The compilation scheme shown is the one that has been classicly used in compiler development in recent years. Therefore, there are many questions to be asked once quantum computing is incorporated: is the same development scheme being followed? And if so, should this be the scheme to follow? These questions will be addressed in the Sect. 3 on quantum compilation.

3 Quantum Compilation

When discussing quantum compilation, several pressing questions emerge: Is the process carried out in a purely quantum manner? How does it differ from classical compilation? What challenges and bottlenecks are intrinsic to quantum compilation? These questions will be addressed as we explore the landscape of quantum compilation.

To address whether the compilation is done entirely in a quantum fashion, it is enlightening to study the current state of quantum computing. Most quantum computers today are either quantum annealers or small-scale universal quantum computers in the NISQ era. These systems lack the capability to compile a comprehensive programming language.

Understanding this requires revisiting the definition of computation. There is no consensus on what "computing" is due to many advances, so an absolute definition is pending in the literature [15]. What we understand today is rooted in the 1930's concept of Automatic Computation (AC), aiming for machines to perform mathematical calculations automatically. Alan Turing's abstract machine, the Turing machine, forms the basis for all computing systems [49]. Computation, based on Turing's definition, is a process where a formal mathematical system operates according to a set of rules in a given physical architecture [14].

A computer executes automatic calculations, traditionally using binary computation, discerning two states, 0 and 1. From vacuum tubes to multicore processors, this binary model has been the foundation of computational implementations. However, the landscape shifted with quantum computation and Shor's algorithm [43]. Beyond Shor's demonstration of quantum superiority, physical limitations of silicon transistors also spurred interest. Will quantum computing replace the current paradigm? Tentatively, no.

Quantum computers can theoretically perform any task a classical computer can, but with different efficiencies. For some calculations, classical computers

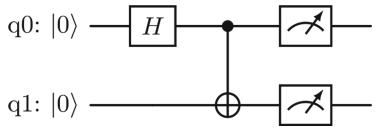
take exponentially more time. This difference is rooted in their principles. Classical computing uses the Cartesian product of systems with two distinct states, while quantum computing employs the tensor product of Hilbert spaces with a basis of two elements, leading to a larger computational space due to superposition and entanglement. However, this does not mean basic operations like NOT are inherently more efficient on a quantum platform. This computational parity explains why quantum computing will not replace the current paradigm. Key parts of a computer-processors, memory, and I/O devices-differ in quantum systems. Specifically, quantum memory and I/O devices pose challenges. Quantum mechanics nature does not align with the needs of I/O devices, which require constant information exchange. Using quantum computers for I/O is inefficient as it reduces qubits to bits.

In quantum computing, memory, often referred to as quantum memory, sparks debate. Quantum memory is volatile, based on quantum Random Access Memory (qRAM) [20,25]. Decoherence, a physical phenomenon, limits non-volatile quantum memory, making full quantum computation inoperable. Qubits are idealized as closed systems, but real quantum experiments can not achieve this, leading to decoherence.

Even if quantum processors, memory, and I/O devices were perfect, would quantum computers replace classical ones? No. Classical computation performs well in many fields with 100% certainty, a major lack in quantum computation. Replacing classical systems does not make sense, but integrating quantum and classical computing offers benefits. Hybrid quantum algorithms like the variational algorithms exemplify the potential synergy between quantum and classical computing, solving specific problems like molecular simulations and optimization [47]. These hybrid algorithms represent a collaborative approach, harnessing the strengths of quantum computing while mitigating challenges like error correction and hardware limitations [17].

After analyzing quantum computation's future, we can answer the first question: no, quantum compilation is not fully quantum. Analyzing the state of the art shows quantum compilation is a classic task with the quantum workload left for execution. Moreover, the workflow mirrors classical compilation. There is an analysis phase, termed the *front-end*, and a synthesis phase, termed the *back-end*, linked via a quantum Intermediate Representation (qIR), analogous to the classical IR. These phases persist throughout quantum computation, with the qIR abstracting multiple target types in the quantum scene.

The following section will describe the analysis phase, similar to its classical counterpart, followed by a section analyzing the synthesis phase, the most complex and divergent from classical counterparts due to its connection with quantum architecture.



(a) Bell circuit with measurement.

```
1 q0 = |0>
2 q1 = |0>
3 H(q0)
4 CNOT(q0, q1)
5 c0 = Measure(q0)
6 c1 = Measure(q1)
```

(b) Pseudocode for the circuit 2a.

Fig. 2. Example of codification of a circuit.

3.1 Compilation Phases for Quantum Computing

Analysis Phase. This phase bears the closest resemblance to the classic compilation world. This similarity should come as no surprise because the problem at hand is exactly the same as in classic compilation: translating a high-level language into an IR (referred to as qIR in this context) that will subsequently be utilized in the synthesis phase. To gain a better understanding of how this problem persists within the quantum workflow, we can engage in an abstraction exercise using the circuit presented in Fig. 2a. For instance, this circuit can be implemented using the structure outlined in the pseudocode found in Fig. 2b. The process of generating an object used for translation into qIR will entail the same steps as those outlined in Sect. 2. It will be presented below with a simplified version of each step, sufficiently profound to illuminate the fundamental concepts.

1. Lexical Analyzer. It detects `q0, q1, c0, c1, H, CNOT` and `Measure` as identifiers, `=` as comparison operator and `|0>` as, for instance, a native token (such as integers or floats). So the lexical result for the line 1 and 5 of Fig. 2b could be:

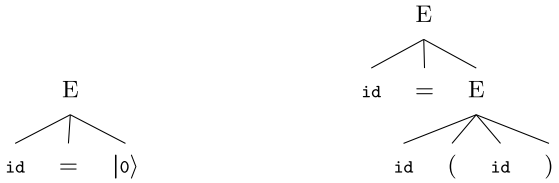
$$\texttt{q0 = } |0\rangle \longleftrightarrow \{\texttt{id,0}\}\,\{=\}\,\{|0\rangle\}$$

$$\texttt{c0 = Measure(q0)} \longleftrightarrow \{\texttt{id,3}\}\,\{=\}\,\{\texttt{id,6}\}\,\{(\}\,\{\texttt{id,0}\}\,\{)\}$$

2. Syntax Analyzer. The previous lexical analysis is now fed into this phase in order to generate a parse tree, which will be generated following the grammar defined for the language. For the purpose of this work, a dummy grammar will be defined in order to define the parse tree of lines 1 and 6.

$$E \longrightarrow \texttt{id} = \texttt{ket} \mid \texttt{id} = \texttt{E} \mid \texttt{id}(\texttt{id})$$

   This grammar in conjunction with the lexical analysis defined above, produce the desired parse trees for lines 1 and 6, portrayed in Fig. 3a and Fig. 3b, respectively.
3. Semantic Analyzer: In this phase, the parse tree generated in the previous step is used to verify the semantic consistency of the source code with the language's definition. For instance, a possible check involves determining if the value $|0\rangle$ is valid for the `q0` identifier and which type should be considered



(a) Parse tree of line 1. (b) Parse tree of line 6.

Fig. 3. Example of parse trees.

for it. In this scheme, it is treated as a native value, similar to integers or floats. However, during this stage, it can be represented as a two-integer array, such as $[0, 0]$, or any other suitable format, which can then be translated into an appropriate signal at the physical layer.

This simplistic example suffices to achieve an important task: showing how similar it is the analysis phase in quantum computing in comparison with classic computing. In fact, if someone abstracts from the common concepts of gates, qubits and measurements, and reads the pseudocode with its grammar and parse trees it is impossible for them to tell whether the language purpose is modeling quantum computing or, for example, embedded computing.

This is precisely why, when conducting a current survey of the state-of-the-art in quantum programming languages [1,18,42], numerous libraries emerge, being among them a set of the most used technologies of the quantum computing domain, as it is the case for Qiskit [13], Circ [16], ProjectQ [45], Pennylane [9], among others. Table 1 lists the different software tools for the development of quantum applications. In this survey, these tools have been categorised into four distinct categories:

- **Frameworks and libraries:** offering pre-written code to enhance efficiency. While libraries provide specific functionalities allowing developers to maintain control over the application's flow, frameworks dictate the project's structure through "inversion of control", where the framework calls the developer's code. Despite some software being labeled differently, most tools like Qiskit, Cirq, ProjectQ, and PennyLane, as well as others such as Qulacs, are generally categorized under frameworks and libraries without a strict adherence to the 'inversion of control' principle.
- **Language extensions:** enhance existing languages by adding features, tools, or syntax. These extensions are not standalone languages but augment the base language's capabilities. Notably, QCOR extends C++ and is highlighted for its relevance, while Quipper, although no longer in use, is significant in quantum software literature.
- **Standalone languages:** self-sufficient languages with their own syntax, semantics, and standard libraries, allowing for the development of various applications independently of other languages. It specifically highlights quantum programming languages.

Table 1. Programming software for quantum computing.

Frameworks libraries		Language extensions		Standalone language		Quantum process algebras
Name	Basis	Name	Basis	Imperative	Functional	
Blueqat	Python	QCOR [36]	C++	isQ [22]	cQPL [34]	CQP [36]
Cirq [16]		Proto-Quipper [41]	Haskell	LanQ [37]	Q [10]	eQPAlg [23]
Penny Lane [9]		Quipper [21]		Q#	QFC [7]	QPAlg [27]
ProjectQ [45]		LIQUi|⟩ [51]	F#	QCL [53]	QML [5]	
Perceval [24]		Chisel-Q [31]	Scala	Q|SI⟩ [30]	QPL	
Qiskit [13]		qGCL [52]	GCL	Quingo [2]	QuaFL [29]	
Quantify		qPCF [32]	PCF	QWIRE [39]	Qunity [50]	
Strawberry Fields		Lambda-q [35]	Lambda Calculus	Scaffold [3]		
Tequila		λ_q [48]		Silq [11]		
Quantum++ [19]	C++					
QuEST [26]						
Qulacs [46]						
QuantumOptics.jl [28]	Julia					
Yao.jl [33]						
Cove [40]	C#					

– **Quantum process algebras:** are theoretical constructs aimed at modeling and analyzing quantum computing systems, especially in quantum communication. They build on traditional process algebras by incorporating quantum mechanics concepts like states, operations, entanglement, and superposition. These algebras serve as formal languages for accurately describing and reasoning about quantum processes, aiding in the development and verification of quantum algorithms and protocols. Although these frameworks are not currently in use, they are recognized for their significant contributions to the initial phases of quantum software development.

Understanding the difference between libraries and programming language extensions in quantum computing can be challenging. For instance, Qiskit operates as a library within Python, offering tools like the `QuantumCircuit` for building quantum circuits without altering Python's syntax. In contrast, QCOR extends C++ by introducing quantum-specific constructs, such as the `__qpu__` keyword for quantum kernels, directly integrating quantum capabilities into the language's syntax. This example is shown clearly in Fig. 4.

Synthesis Phase. In classic compilation, each line of code translates to machine-level instructions. However, in quantum languages, most components are classical except for the quantum circuit part, which is generated by the classical elements of the program. Quantum computing languages typically exist as libraries or extensions of classical languages to optimize quantum circuits for

```
__qpu__ void ghz(qreg q) {
  auto first_qubit = q.head();
  H(first_qubit);
  for (auto i : range(q.size()-1))
    X::ctrl(q[i], q[i+1]);
  Measure(q);
}
```

(a) QCOR code (C++ extension).

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

qc.measure(1, 0)
```

(b) Qiskit code (Python library).

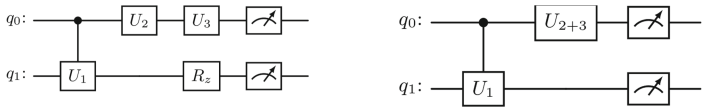
Fig. 4. Example of the difference between a library and a language extension.

minimal gate count and depth. The synthesis phase in quantum compilation is crucial for generating these optimal circuits, whether for actual quantum computers or simulators. This review focuses on the quantum aspects of this phase, highlighting the unique challenges of quantum compilation.

The synthesis phase of quantum compilation can be broadly divided into two main components: **circuit optimization** and **qubit mapping**. Of course, these two parts are not the only ones which can be included in the synthesis phase. Different techniques and processes such as error correction, error mitigation, analysis of metrics extracted from the quantum circuit and so on and so forth, can be included in this phase. But there is not an agreement among the different compilers and software tools whether other techniques are necessary in this phase.

Circuit optimization operates on the principle of modifying a quantum circuit to minimize a specific metric, essential due to the complexity of quantum computers. The choice of metric varies: for NISQ devices, reducing the count of two-qubit gates might be crucial, while for others, minimizing circuit depth due to qubit decoherence may be preferred. The main challenge in circuit optimization is ensuring the modified circuit retains its original functionality. This necessitates verification to confirm the circuit still performs the same computational task. Thus, the optimization phase can be divided into two subphases: **improvement** and **verification**.

The aim of **circuit improvement** is to enhance the performance and efficiency of quantum circuits through techniques such as removing redundant inverse gates, optimizing subcircuits, and employing various algorithms for better circuit synthesis. Figure 5 illustrates an example of circuit improvement using gate fusion and deletion. As can be seen, gates U_2 and U_3 are susceptible to be fused, while the rotations in the Z axis do not modify the measurement in the computational basis, therefore they can be eliminated. Conversely, **circuit verification** ensures the correctness of these optimized quantum circuits, verifying that the transformations applied during optimization preserve the intended functionality, with techniques like [6] and ZX-Calculus [12] maintaining circuit integrity. In quantum compilation, while many compilers like Qiskit, ProjectQ, and ScafCC focus primarily on optimization, the integration of verifica-



(a) Non-optimized circuit. (b) Optimized circuit.

Fig. 5. Example of the optimization of a circuit applying gate fusion techniques.

tion underscores the importance of ensuring circuit correctness throughout the compilation process.

Qubit mapping: allocation and routing is analogous to register allocation in classical computing, aims to efficiently assign logical qubits to physical qubits in a quantum device, considering constraints like device connectivity [44], while minimizing resources and execution time. This process is complicated by the varying error rates and connectivity of qubits, making it an NP-hard problem where exact algorithms are feasible only for small numbers of qubits. Consequently, heuristic and approximation techniques are often employed to find near-optimal solutions. This task involves two main processes:

- **Quantum allocation:** refers to the process of physically assigning specific logical qubits in a quantum processor. For a correct qubit allocation, in most cases, it is necessary to add additional SWAP gates to move the qubit information [38].
- **Quantum routing:** refers to the task of finding efficient paths for communication between qubits in a quantum processor. This is important when mapping gates of two logic qubits that are not interconnected to maximise efficiency [8].

4 Conclusions

This survey has examined the state-of-the-art in quantum compilation, focusing on the processes and challenges unique to quantum systems. Quantum compilation retains many parallels with classical methodologies, including the analysis and synthesis phases, but it introduces significant complexities. The necessity for qubit mapping and circuit optimization, coupled with verification techniques, underscores the distinct nature of quantum compilation. Furthermore, the potential of hybrid algorithms, such as the variational algorithms, demonstrates the benefits of integrating quantum and classical computing. As quantum technology progresses, efficient and scalable compilation methods will be crucial for maximizing the performance and applicability of quantum processors. Future work should continue to explore and refine these methods, ensuring robust and effective compilation processes that can meet the demands of advancing quantum hardware. In the same way, it is also necessary to focus on innovative methodologies, such as distributed quantum compilers.

Acknowledgments. This work was supported by MICINN through the European Union NextGenerationEU recovery plan (PRTR-C17.I1), and by the Galician Regional Government through the "Planes Complementarios de I+D+I con las Comunidades Autónomas" in Quantum Communication. This work was also supported by the Ministry of Economy and Competitiveness, Government of Spain (Grant Numbers PID2019-104834GB-I00, PID2022141623NB-I00 and PID2022-137061OB- C22), Consellería de Cultura, Educación e Ordenación Universitaria (accreditations ED431C 2022/16 and ED431G-2019/04), and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Center in Intelligent Technologies of the University of Santiago de Compostela as a Research Center of the Galician University System.

Disclosure of Interests. The authors declare no conflict of interest.

References

1. Quantum programming languages (2020). https://doi.org/10.1038/s42254-020-00245-7
2. Fu, X., et al.: Quingo: a programming framework for heterogeneous quantum-classical computing with nisq features. ACM Trans. Quant. Comput. **2**, 1–37 (2021). https://doi.org/10.1145/3483528
3. Abhari, A.J., et al.: Scaffold: Quantum programming language (2012)
4. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
5. Altenkirch, T., Grattage, J.: A functional quantum programming language (2005)
6. Amy, M.: Towards large-scale functional verification of universal quantum circuits. Electron. Proc. Theor. Comput. Sci. EPTCS **287**, 1–21 (2018). https://doi.org/10.4204/EPTCS.287.1. http://arxiv.org/abs/1805.06908
7. Bal, H.E., Steiner, J.G., Tanenbaum, A.S.: Programming languages for distributed computing systems. ACM Comput. Surv. (CSUR) **21**, 261–322 (1989). https://doi.org/10.1145/72551.72552
8. Bandic, M., Almudever, C.G., Feld, S.: Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques. Quant. Mach. Intell. **5**, 1–30 (2023). https://doi.org/10.1007/S42484-023-00124-1/TABLES/4. https://link.springer.com/article/10.1007/s42484-023-00124-1
9. Bergholm, V., , et al.: Pennylane: Automatic differentiation of hybrid quantum-classical computations (2018). http://arxiv.org/abs/1811.04968
10. Bettelli, S., et al.: Toward an architecture for quantum programming. Eur. Phys. J. D - Atomic Molec. Opt. Plasma Phys. **25**(2), 181–200 (2003). https://doi.org/10.1140/EPJD/E2003-00242-2
11. Bichsel, B., et al.: Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 286–300 (2020). https://doi.org/10.1145/3385412.3386007
12. Coecke, B., Duncan, R.: Interacting quantum observables. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). LNCS, vol. 5126, pp. 298–310 (2008). https://doi.org/10.1007/978-3-540-70583-3_25/COVER

13. Contributors, Q.: Qiskit: an open-source framework for quantum computing (2023). https://doi.org/10.5281/zenodo.2573505
14. Copeland, B.J.: What is computation? Synthese **108**, 335–359 (1996)
15. Denning, P.J.: The great principles of computing. Am. Sci. **98**(5), 369–372 (2010)
16. Developers, C.: Cirq (2023). https://doi.org/10.5281/zenodo.8161252
17. Endo, S., et al.: Hybrid quantum-classical algorithms and quantum error mitigation. J. Phys. Soc. Jpn. **90**(3), 032001 (2021)
18. Garhwal, S., Ghorani, M., Ahmad, A.: Quantum programming language: a systematic review of research topic and top cited languages. Arch. Comput. Methods Eng. **28**(2), 289–310 (2019). https://doi.org/10.1007/s11831-019-09372-6
19. Gheorghiu, V.: Quantum++: a modern c++ quantum computing library. PLoS ONE **13** (2018). https://doi.org/10.1371/journal.pone.0208073
20. Giovannetti, V., Lloyd, S., Maccone, L.: Quantum random access memory. Phys. Rev. Lett. **100**, 160501 (2008)
21. Green, A.S., et al.: Quipper: a scalable quantum programming language (2013). https://doi.org/10.1145/2499370.2462177
22. Guo, J., et al.: isq: An integrated software stack for quantum programming. IEEE Trans. Quant. Eng. **4** (2023). https://doi.org/10.1109/TQE.2023.3275868
23. Haider, S., Kazmi, D.S.A.R.: An extended quantum process algebra (eqpalg) approach for distributed quantum systems (2020). http://arxiv.org/abs/2001.04249
24. Heurtel, N., et al.: Perceval: a software platform for discrete variable photonic quantum computing. Quantum **7**, 931 (2023). https://doi.org/10.22331/q-2023-02-21-931
25. Jaques, S., Rattew, A.G.: Qram: a survey and critique. arXiv preprint arXiv:2305.10310 (2023)
26. Jones, T., et al.: Quest and high performance simulation of quantum computers. Sci. Rep. **9** (2019). https://doi.org/10.1038/s41598-019-47174-9
27. Jorrand, P., Lalire, M.: Toward a quantum process algebra. In: 2004 Computing Frontiers Conference, pp. 111–119 (2004). https://doi.org/10.1145/977091.977108
28. Krämer, S., Plankensteiner, D., Ostermann, L., Ritsch, H.: Quantumoptics.jl: a julia framework for simulating open quantum systems. Comput. Phys. Commun. **227**, 109–116 (2018). https://doi.org/10.1016/J.CPC.2018.02.004
29. Lapets, A., da Silva, M.P., Thome, M., Adler, A., Beal, J., Roetteler, M.: Quafl: a typed dsl for quantum programming, pp. 19–26. Association for Computing Machinery (2013). https://doi.org/10.1145/2505351.2505357
30. Liu, S., et al.: $Q|SI\rangle$: a quantum programming environment. In: Jones, C., Wang, J., Zhan, N. (eds.) Symposium on Real-Time and Hybrid Systems. LNCS, vol. 11180, pp. 133–164. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01461-2_8
31. Liu, X., Kubiatowicz, J.: Chisel-q: designing quantum circuits with a scala embedded language. In: 2013 IEEE 31st International Conference on Computer Design, ICCD 2013, pp. 427–434 (2013). https://doi.org/10.1109/ICCD.2013.6657075
32. Paolini, L., Zorzi, M.: qPCF: a language for quantum circuit computations. In: Gopal, T.V., Jäger, G., Steila, S. (eds.) TAMC 2017. LNCS, vol. 10185, pp. 455–469. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55911-7_33
33. Luo, X.Z., et al.: Yao.jl: extensible, efficient framework for quantum algorithm design (2020). https://doi.org/10.22331/q-2020-10-11-341
34. Mauerer, W.: Semantics and simulation of communication in quantum programming (2005)
35. Maymin, P.: Extending the lambda calculus to express randomized and quantumized algorithms (1996). https://arxiv.org/abs/quant-ph/9612052v2

36. Mintz, T.M., et al.: Qcor: a language extension specification for the heterogeneous quantum-classical model of computation. J. Emerg. Technol. Comput. Syst. **16**(2) (2020). https://doi.org/10.1145/3380964
37. Mlnarik, H.: Operational semantics and type soundness of quantum programming language lanq (2007). https://arxiv.org/abs/0708.0890v1
38. Paler, A.: On the influence of initial qubit placement during NISQ circuit compilation. In: Feld, S., Linnhoff-Popien, C. (eds.) QTOP 2019. LNCS, vol. 11413, pp. 207–217. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14082-3_18
39. Paykin, J., Rand, R., Zdancewic, S.: Qwire: a core language for quantum circuits. ACM SIGPLAN Not. **52**, 846–858 (2017). https://doi.org/10.1145/3009837.3009894
40. Purkeypile, M.D.: Cove: A practical quantum computer programming framework. arXiv org (2009)
41. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. Electron. Proc. Theor. Comput. Sci. **266** (2017). https://doi.org/10.4204/EPTCS.266.11
42. Serrano, M.A., Cruz-Lemus, J.A., Perez-Castillo, R., Piattini, M.: Quantum software components and platforms: overview and quality assessment. ACM Comput. Surv. **55**, 1–31 (2023). https://doi.org/10.1145/3548679
43. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
44. Siraichi, M.Y., et al.: Qubit allocation. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, pp. 113–125. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3168822
45. Steiger, D.S., et al.: Projectq: an open source software framework for quantum computing. Quantum **2** (2018). https://doi.org/10.22331/q-2018-01-31-49
46. Suzuki, Y., et al.: Qulacs: a fast and versatile quantum circuit simulator for research purpose. Quantum **5** (2021). https://doi.org/10.22331/Q-2021-10-06-559
47. Tilly, J., et al.: The variational quantum eigensolver: a review of methods and best practices. Phys. Rep. **986**, 1–128 (2022). https://doi.org/10.1016/j.physrep.2022.08.003
48. van Tonder, A.: A lambda calculus for quantum computation. SIAM J. Comput. **33**, 1109–1135 (2003). https://doi.org/10.1137/S0097539703432165
49. Turing, A.M., et al.: On computable numbers, with an application to the entscheidungsproblem. J. Math **58**(345–363), 5 (1936)
50. Voichick, F., et al.: Qunity: a unified language for quantum and classical computing. Proc. ACM Program. Lang. **7**, 921–951 (2023). https://doi.org/10.1145/3571225
51. Wecker, D., Svore, K.M.: Liqui|spsdoigtsps: a software design architecture and domain-specific language for quantum computing (2014)
52. Zuliani, P.: Non-deterministic quantum programming, pp. 179–195 (2004)
53. Ömer, B.: Classical concepts in quantum programming. Int. J. Theor. Phys. **44**, 943–955 (2005). https://doi.org/10.1007/S10773-005-7071-X/METRICS