tweedledum: A Compiler Companion for Quantum Computing

Bruno Schmitt, Giovanni De Micheli
Integrated Systems Laboratory (LSI), EPFL, Switzerland

***Abstract*—This work presents tweedledum—an extensible open-source library aiming at narrowing the gap between high-level algorithms and physical devices by enhancing the expressive power of existing frameworks. For example, it allows designers to insert classical logic (defined at a high abstraction level, e.g., a Python function) directly into quantum circuits. We describe its design principles, concrete implementation, and, in particular, the library's core: An intuitive and flexible intermediate representation (IR) that supports different abstraction levels across the same circuit structure.**

***Index Terms*—quantum, design automation, compilation**

I. Introduction

The emergence of quantum computers with an increasingly higher number of qubits and longer coherence times marks the beginning of an exciting era in quantum technology, as it empowers us to solve problems that are out of reach for any of the best classical supercomputers [1], [2]. However, finding applications and algorithms for which a quantum algorithm yields a significant scaling advantage in time-to-solution, known as a quantum speed-up, over classical algorithms has been a significant concern in the quantum computing research community. The "Quantum Algorithm Zoo" website [3] holds a comprehensive list of quantum algorithms with their respective speed-up factors. Finding more of these algorithms is crucial to quantum computing progress.

The quantum computing community hopes that the availability of quantum hardware and the development of quantum programming languages will stimulate and aid the discovery of new quantum algorithms [4]. However, most programming systems available for quantum computing are intertwined with the quantum circuit model, which means that the developer must describe the algorithm in terms of basic unitary operators. Not surprisingly, the implementation of quantum algorithms on such a low level of abstraction is very time-consuming, error-prone, and results in non-portable implementations—given the diversity of quantum devices.

To explore new algorithms and programs for quantum computing as well as to enhance the productivity of programmers, we have seen many quantum frameworks striving to support higher-level abstractions, e.g., `Q#` [5], `Qiskit` [6], `PyQuil/Forest` [7], `Pennylane` [8]. These frameworks allow developers to create increasingly more complex programs by combining and adapting a small set of known quantum algorithms that, often, use arbitrary technology-independent operations. Nevertheless, given the stringent resource constraints in near-term quantum hardware, the use of higher levels of abstraction is only possible when allied with sophisticated compilation algorithms capable of generating highly optimized low-level circuits. Since circuit compilation and optimization are essential for quantum computing [9], there are numerous competing compilers and toolkits, e.g., `qiskit-terra` [6], `quilc` [10], `ScaffCC` [11], `staq` [12], and `t|ket⟩` [13]. We refer to [14] for a survey of quantum software stacks.

The embodiment of our research contribution is a compiler companion library for the synthesis and compilation of quantum circuits called `tweedledum`. In contrast to most solutions, we designed it to enhance other compilers and frameworks, and some of these other tools indeed use it already, e.g., `qiskit-terra`, `quilc` and `staq`. We implement `tweedledum` as an open-source library[1] in C++-17 and provide Python bindings for easy integration into existing compilers/frameworks. The library integrates state-of-the-art algorithms used for quantum compilation, targeting most of the pipeline of a quantum software stack: from the abstract higher algorithmic layers to the physical mapping layer.

II. Overview

The goal of compilation is to bridge the gap between high-level quantum programs and technology-dependent implementations. Internally, compilation breaks down into a sequence of tasks: lexing the source code into a sequence of tokens, parsing these tokens into an abstract syntax tree (or AST), validating this AST, and finally, translating it into technology-dependent representation.

At this point, `tweedledum` mainly aims at enhancing a compiler's capabilities of doing the last translation step. Hence its compiler companion denomination. The library provides various algorithms for synthesizing, optimizing, and manipulating quantum circuits. Fig. 1 shows a bird's eye view of a compilation process. A compiler can use `tweedledum` to build a high-level quantum circuit and then use its passes to progressively lower it into a technology-dependent circuit using the fewest resources possible. For example, `tweedledum` empowers `qiskit-terra` to accept classical logic functions, written in Python, as an oracle definition and handles their translation into quantum circuits. Also, it adds the ability to synthesize permutation to Rigetti's `quilc`.

III. Representation of quantum functionality

The basic mathematical objects to be dealt with when representing quantum functionality are Hamiltonians of a quantum system. These are linear, unitary mappings $\mathbb{C}^{2^n} \mapsto \mathbb{C}^{2^n}$ that describe the system's evolution. There are several ways for representing quantum operators, and `tweedledum` supports the most common ones. Each way has its strengths and

[1] https://github.com/boschmitt/tweedledum

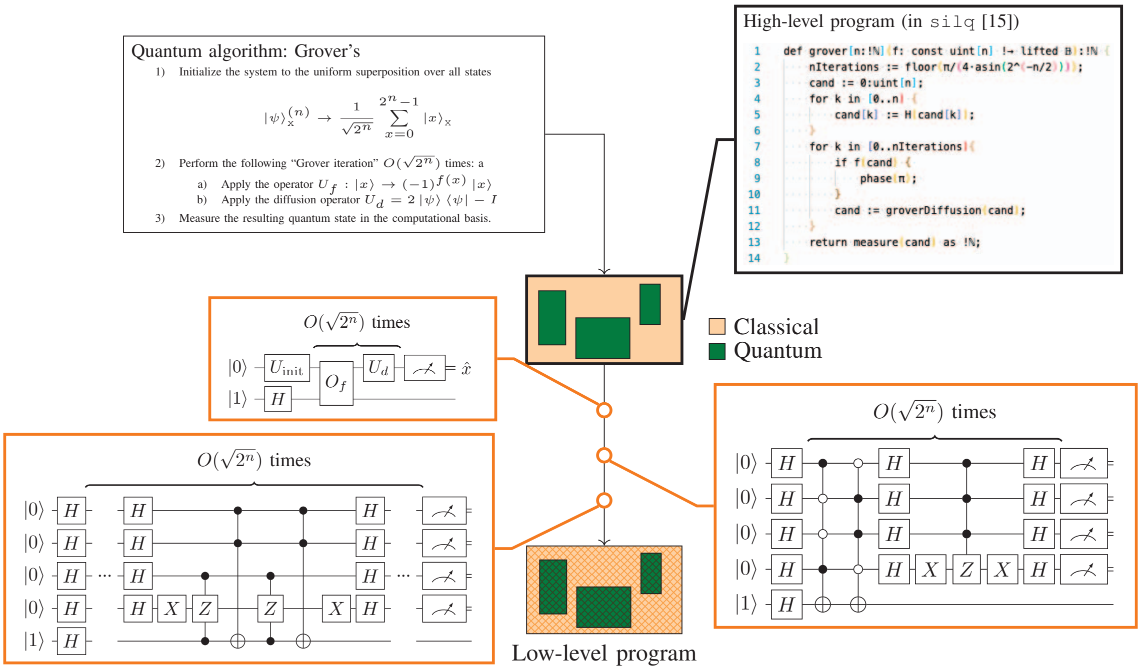


Fig. 1: Compilation flow overview. A quantum algorithm contains both classical operations and quantum operations. The latter can be given in various forms and sometimes can even be classically defined, e.g., O_f. A compiler translates a high-level program into a sequence of operations executable by a quantum processing unit (QPU).

weaknesses. We evaluate the efficiency of a representation by its succinctness in describing operators and its capability of supporting transformations and manipulations. Since no representation is universally suitable for all applications, conversion is essential during compilation, where various synthesis and transformation techniques are applied.

Fig. 2 shows some of the different ways to define quantum functionality. The most natural way to represent a Hamiltonian is to choose a basis of the Hilbert space and then consider the corresponding transformation matrix, a $2^n \times 2^n$ complex-valued unitary matrix. Unitary matrices are canonical: two quantum operators are equivalent if and only if they have the same unitary matrix. Note that canonicity is an essential property for many applications of synthesis and verification. However, they are impractical to represent operators acting on many qubits since their size grows exponentially with the number of qubits.

On the other hand, the quantum circuit model is a simple and convenient tool for representing quantum programs that do not suffer from the same exponential growth. One of the disadvantages is its lack of canonicity, i.e., there are many different ways of representing a given computation using quantum circuits. We can simplistic view many parts of the compilation process as a search for a circuit that betters optimizes a cost function of our interest. We refer to [16] and [17] for discussions on quantum decision diagrams and phase polynomials, respectively.

IV. Intermediate Representation

In tweedledum, the standard IR is a quantum circuit. A circuit has a set of wires (qubits and bits) and a sequence of operators applied to those wires, the so-called instructions. An operator is an abstract effect that may modify the state of a subset of wires. An instruction is an operator applied to a specific subset of qubits and bits. In other words, to represent a quantum computation, we first create an empty circuit to which we add qubits and bits. Then we create instructions by applying operators to these qubits and bits.

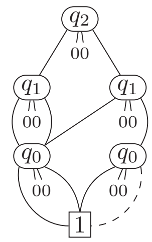
A. Fundamental concepts

1) Wires: Qubits and bits: We represent qubits and bits using memory semantics, which means instructions act on references to qubits (and bits) and do not consume their value; we say they affect their state via side effects. A benefit of using memory semantics is that the IR inherently prevents a program from violating the no-cloning theorem [18]. Indeed, no IR mechanism allows the copy of a quantum state.

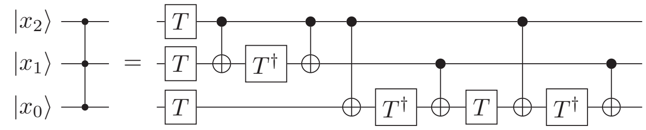
2) Operators: Conceptually, an operator is an effect that we can apply to a subset of qubits and bits. Most often, this effect is unitary evolution. To ensure a high level of customizability, the library does not have a fixed set of operators nor imposes restrictions on how their effects are defined. On the contrary, it encourages and facilitates the implementation of user-defined ones. Indeed, one of the main strengths of tweedledum lies in its use of a uniform concept, known as *Operator*, to enable

$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&e^{i\pi}\end{pmatrix}$$

(a) Unitary matrix



(b) Decision diagram



(c) Quantum circuit

$$(\tfrac{\pi}{4}, x_1), (\tfrac{\pi}{4}, x_2), (\tfrac{\pi}{4}, x_3), (-\tfrac{\pi}{4}, x_1 \oplus x_2),$$
$$(-\tfrac{\pi}{4}, x_1 \oplus x_3), (-\tfrac{\pi}{4}, x_2 \oplus x_3), (\tfrac{\pi}{4}, x_1 \oplus x_2 \oplus x_3)$$

(d) Phase polynomial

Fig. 2: Some different ways to represent quantum functionality.

describing different levels of abstractions and computations. For example, we can define high-level operators such as the truth table operator, the permutation operator, or low-level operators like the Hadamard operator, e.g, Fig. 3.

The library imposes two minimal requirements for the Operator concept. Namely, every Operator must be identifiable and must target at least one qubit. (Note that the number of targets is intrinsic to an operator; the number of controls is not.)

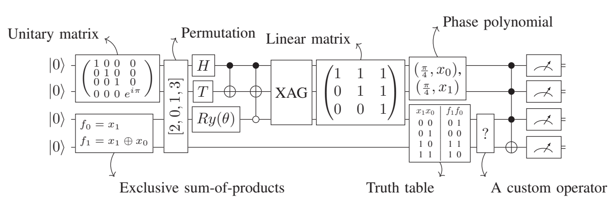


Fig. 3: `tweedledum`'s IR flexibility.

3) Instructions: An instruction is the embodiment of the application of an operator to a specific subset of wires. The number of qubits must be equal to or greater than the number of targets required by the operator. If the latter, the extra qubits are considered controls. This design choice aims to counterbalance the easiness of defining new operators, which might lead to an explosion in their number. Hence, in `tweedledum`, the instructions NOT, CNOT and TOFFOLI are all the same operator X applied to a different number of qubits.

4) Circuit: In `tweedledum`, a quantum circuit is a directed acyclic graph with both labeled vertices and labeled edges. Vertices correspond to instructions. Their label determines which operator the instruction applies. The edges encode input/output relationships between instructions, and their label indicates the qubit (bit) associated with this relationship. Its underlying implementation also allows it to be treated as a simple list of instructions (netlist).

V. Synthesis

We require a synthesis process whenever the desired quantum functionality is not given in terms of a quantum circuit, e.g., a transformation matrix describing the unitary operator, a Boolean function, a permutation, to name a few.

An exciting feature of `tweedledum` is the ability to insert classical logic directly into quantum circuits and to synthesize it during compilation. Moreover, the library provides means for a user to define such classical logic as a Python function. By combining most of the EPFL logic synthesis libraries [19], we can create compilation flows that handle the translation of this Python function into a sequence of elementary quantum operators.

The library also accepts Boolean functions provided in other forms, e.g, a logic network. Sometimes the input is already a reversible Boolean function. In such cases, we can use specialized methods to synthesize a circuit [20], [21]. More often, however, the Boolean function will be irreversible. Given an irreversible function f, it is known that there must exist a reversible Boolean function $f' : \{0,1\}^{n+1} \mapsto \{0,1\}^{n+1}$ such that

$$f'(x, y) = (x, y \oplus f(x)),$$

where $x = x_0, \ldots, x_{n-1}$ and '$\oplus$' refers to the XOR operation. (For the sake of clarity, we limit the discussion to single-output Boolean functions, but the technique can be extended to accommodate multiple-output functions.) Such an embedding is also referred to as Bennett embedding [22], and implies the existence of the following quantum operation:

$$B_f : |x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$$

The operation B_f is also known as a single-target operator. Single-target operators describe complex operations that cannot generally be implemented natively on a quantum computer. `tweedledum` provides three techniques to synthesize single-target operators directly. Starting from a functional representation of f, i.e., a truth table, both `pkrm_synth` and `pprm_synth` techniques synthesize a particular case of an exclusive sum-of-products (ESOP) expression for f. We can easily translate such ESOP into a cascade of multiple-control X operators. These techniques, however, are only applicable to small Boolean functions as they can be both very time-consuming and generate a quantum circuit with a prohibitive number of instructions [23], [24]. `spectrum_synth` [25] uses the Rademacher-Walsh spectrum of a truth table to generate a circuit over the operator set Clifford+Rz directly.

For a more scalable solution, we combine these direct methods with the hierarchical synthesis approaches (Fig. 4).

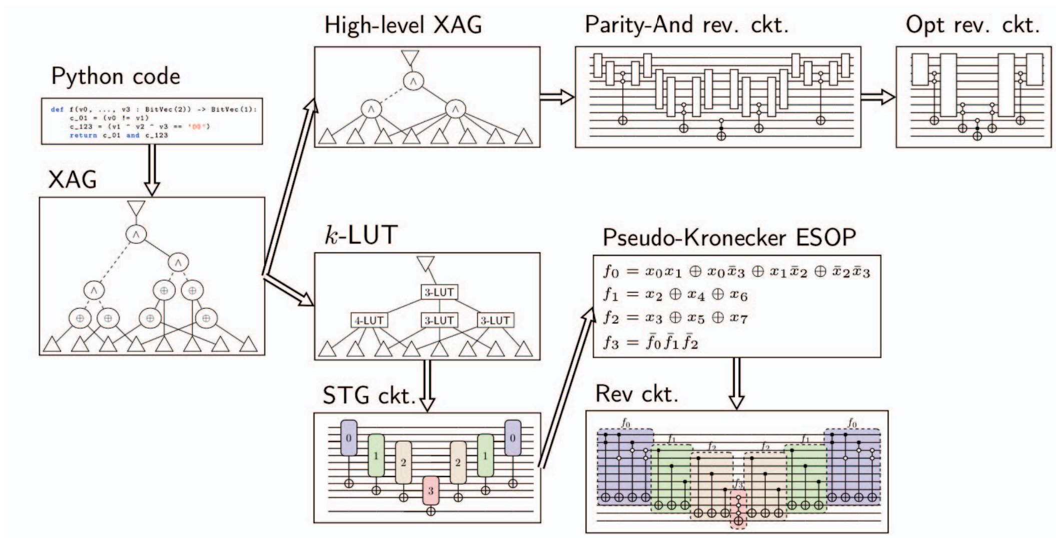


Fig. 4: Bird's eye view of possible Boolean function synthesis flows.

The latter allows us to achieve scalability by decomposing the initial function into small parts suitable for functional synthesis. The synthesis method generates a reversible circuit for each part and combines them following the structural representation of the function. The combination of subcircuits might require additional qubits, which store intermediate computation steps. Given an irreversible Boolean function f they find an $(n+1+a)$-qubit quantum circuit that realizes the unitary

$$B_f : |x\rangle|y\rangle|0\rangle^a \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^a$$

where $a \geq 0$, which means that the synthesis algorithm can use the a additional qubits to store intermediate computations. `lhrs_synth` [26] and `xag_synth` [27] are examples of hierarchical synthesis.

Linear reversible circuits form a subclass of quantum circuits in which implementation requires only CNOT operators. Synthesis methods aiming to reduce the size of these circuits play an essential role in `tweedledum` since other algorithms depend on them. Given a binary matrix describing the classical function, the library has two techniques for synthesizing such circuits. Both methods rely on a modified implementation of Gaussian elimination, which yields asymptotically optimal circuits. They differ in that the `steiner_gauss_synth` [12], [28] algorithm can synthesize circuits that respect the connectivity constraints of device architectures. We also provide two techniques, `a_star_swap_synth` and `sat_swap_synth` [29], to synthesize an even more constrained class of linear reversible circuits: those composed entirely of SWAP operators.

VI. Compilation Passes

A. Utility

Passes in this category provide simple utilities that do not fit any other category, which more complex passes might either require or significantly benefit from using it. For example, since `tweedledum` does not modify circuits in place, all passes that modify a circuit must do a shallow duplication, i.e., create a new circuit with the same wires. There are also passes to reverse and invert circuits. The `reverse` pass creates a new circuit with the instruction applied in the reverse topological order. Inversion is similar, but with the addition of applying the adjoint instruction.

The library also provides an instruction canonicalization pass. The goal of canonicalization is to make optimizations more effective. Very often, we can write instructions in multiple forms. For example, we can write them with equivalent operators $T = P(\frac{7\pi}{8}) = P(-\frac{\pi}{4})$; or apply an operator to a permutation of the same wires $\mathrm{SWAP}(q_0, q_1) = \mathrm{SWAP}(q_1, q_0)$. Canonicalization means selecting one of these forms to be canonical and then going through a circuit and rewriting all instructions into the canonical form. Thus, canonicalization allows optimization passes that look for specific patterns to focus only on the canonical forms rather than all forms.

B. Decomposition

We define decomposition as the process of systematically breaking down high-level instructions into a series of lower-level ones. We emphasize "systematically" because it is the characteristic that differentiates decomposition from synthesis. A decomposition technique builds a lower-level implementation through the application of some construction rule(s). Compared to synthesis, decomposition techniques are faster and produce predictable results, i.e., we know in advance the resulting number of instructions and qubits.

The cost to decompose a multiple-controlled Pauli instruction depends on whether the circuit has clean ancillae available. (Here, we measure cost by the number of instructions.) For example, the circuit might have enough clean ancillae to allow a "v-chain" decomposition, as illustrated in Fig. 5. If that is not the case, we need to use the more costly "dirty-ancilla" decomposition (based on Lemmas 7.1 and 7.2

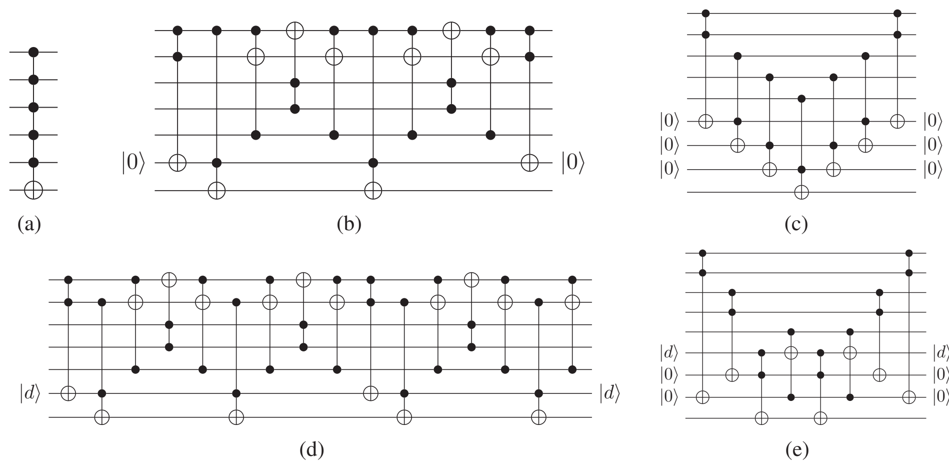


Fig. 5: Different ways of decomposing a multiple controlled X instruction (a) depending on the number of ancillae available.

from [30]). An interesting feature of our implementation of `barenco_decomp` is its adaptability. It uses a "v-chain" for as long as there is a clean ancilla, then switches to "dirty-ancilla." Furthermore, depending on the user configuration, the algorithm can automatically add clean ancillae and use relative phase instructions for intermediate computations.

C. Mapping

The physical qubits in most quantum hardware are not fully connected, which means that not every pair can participate in the same physical instruction. These connectivity restrictions are known as coupling constraints. `tweedledum`'s intermediate representation has no mechanism to enforce them, i.e., we can apply an operation between any pair of qubits. Therefore, synthesized or constructed circuits typically cannot be directly executed on a quantum device. We refer to them as unmapped circuits and define virtual (or logical) qubits and instructions as those present in them. The task of finding a mapping of virtual instructions to allowed physical instructions is known as quantum circuit mapping. The completion of this task is not always possible without applying additional operators to the circuit.

Formally, we model the connectivity requirements of an unmapped circuit using an undirected graph $G_v = (V, E_v)$ where V is the set of virtual qubits $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and $E_v \subseteq \binom{V}{2}$ is a set of qubit pairs $\{v_i, v_j\}$ used by the instructions in the circuit. (Note that one-qubit instructions can be safely ignored since the coupling constraints do not affect their mapping.) We model the coupling constraints in a similar way. A unidirected graph (P, E_p) where $P = \{p_0, p_1, \ldots, p_{m-1}\}$ is a set of physical qubits and an edge $\{p_u, p_w\} \in E_p \subseteq \binom{P}{2}$ means that a instruction can be executed using the two physical qubits p_u and p_w. The library divides mapping into two sub tasks: placement and routing.

1) Placement: The goal is to find a subgraph isomorphism $\pi : V \mapsto P$ which respects $(v_i, v_j) \in E_v \implies (\pi(v_i), \pi(v_k)) \in E_p$. If it exists, then we call π a perfect placement and the mapping task requires only a relabeling of the virtual qubits to physical qubits, i.e., replace each virtual qubit v_i in the unmapped circuit by a physical qubit $\pi(v_i)$. Otherwise, we must resort to routing.

`tweedledum` implements several placement algorithms. One of them, a SAT-based algorithm, tries to find a perfect placement by solving a Boolean satisfiability problem (SAT): We encode a qubit placement problem as a Boolean function. This function is satisfiable if and only if there is a perfect qubit placement. A solver determines the Boolean function's satisfiability. When the problem is satisfiable, the solver provides a satisfying assignment from which we extract the perfect placement. If such placement does not exists, it returns a best-effort placement.

2) Routing: Given an initial placement, a router transforms a circuit so that all two-qubit instructions operate on physically adjacent qubits. Thus, our goal becomes finding a way of mapping a given circuit on a given device architecture with low overhead, whether in the number of additional instructions or depth of the resulting circuit. All routers implemented in `tweedledum` guarantee the compilation of any quantum circuit to any architecture represented as a simple connected graph. They are, therefore, completely hardware agnostic. These routing algorithms iteratively construct a new circuit that conforms to the desired architecture constraints.

D. Optimization

With the limited space and time resources available on current quantum devices, aggressive circuit optimization techniques are essential to extract all performance out of the machines. They often play a crucial role in whether a circuit can or cannot execute in a device or a simulator. Furthermore, they provide more accurate resource estimates, which might guide quantum algorithms and hardware development.

`tweedledum` provides circuit optimizations as a set of orthogonal compilation passes that can be composed into compilation flows. Optimization techniques range from purely structural, relying only on the relationship between instructions on a quantum circuit representation, to purely functional, e.g., when they rely on resynthesizing a circuit from a unitary matrix.

There exists a significant trade-off between the quality of results and scalability. On the one hand, structural transformations offer better scalability at the cost of inferior quality of results. On the other hand, functional optimizations offer the contrary: better quality of results and poor scalability.

a) Instruction cancellation: As the name implies, this pass performs basic instruction cancellation: it traverses a circuit and removes pairs of adjacent adjoint instructions. This optimization is purely structural, and its effectiveness is highly dependent on the canonicalization of the instructions.

b) Phase folding: This optimization pass extends [17]'s T-count optimization algorithm to enable merging parameterized phase operators and handling arbitrary operators. The implementation handles operators not belonging to set $\{X, \mathrm{CNOT}, \mathrm{SWAP}, P(\theta)\}$ conservatively. It ignores their phase contribution: this pass only keeps track of phase polynomial terms that are trivially mergeable.

c) Linear resynthesis: The instruction cancellation pass suffers from structural bias: the input/output relationship between instructions (the structure) strongly influences the quality of results. Resynthesis techniques circumvent this problem by traversing the circuit searching subcircuits that they know how to represent functionally and synthesize. In the linear resynthesis pass, we first identify linear subcircuits and represent their functionality as a binary matrix. Then we try to find a less (or equally) costly implementation by resynthesizing a new subcircuit using `linear_synthesis` algorithm.

VII. Conclusion

It is widely believed that a language's expressive power influences the depth at which people can think. Existing quantum computing programming languages and frameworks limit the kinds of control structures, data structures, and abstractions developers can use; thus, the forms of algorithms they can construct are likewise limited. This paper introduced `tweedledum`—a library that augments the expressive power of current frameworks by providing methods for synthesis, compilation, and optimization of quantum circuits. The library also seeks to create an open-source environment where state-of-the-art quantum compilation techniques can be developed and compared.

Finally, we see that the field of quantum software is evolving rapidly, driven by various factors, ranging from progress in quantum hardware to improved compilation techniques. While we cannot foresee what the future will hold, the flexible design of `tweedledum` presents many possibilities for future improvements.

Acknowledgments

We thank Mathias Soeken for valuable discussions regarding the library implementation and the development of its various algorithms. This research was partially supported by the ERC project H2020-ERC-2014-ADG 669354 CyberCare.

References

[1] J. M. Martinis and al., "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019.

[2] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu, P. Hu, X.-Y. Yang, W.-J. Zhang, H. Li, Y. Li, X. Jiang, L. Gan, G. Yang, L. You, Z. Wang, L. Li, N.-L. Liu, C.-Y. Lu, and J.-W. Pan, "Quantum computational advantage using photons," *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020.

[3] S. Jordan. [Online]. Available: https://quantumalgorithmzoo.org/

[4] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," 2017.

[5] K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL," in *RWDSL*. ACM Press, 2018, pp. 1–10.

[6] Contributors, "Qiskit: An Open-source Framework for Quantum Computing," 2019.

[7] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A Practical Quantum Instruction Set Architecture," 2017.

[8] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri, K. McKiernan, J. J. Meyer, Z. Niu, A. Száva, and N. Killoran, "PennyLane: Automatic differentiation of hybrid quantum-classical computations," 2020.

[9] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits—a survey," 2013.

[10] R. S. Smith, E. C. Peterson, M. G. Skilbeck, and E. J. Davis, "An open-source, industrial-strength optimizing compiler for quantum programs," *Quantum Science and Technology*, vol. 5, no. 4, p. 044001, 2020.

[11] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: A framework for compilation and analysis of quantum computing programs," in *ACM CF*, 2014, pp. 1–10.

[12] M. Amy and V. Gheorghiu, "Staq—A full-stack quantum processing toolkit," *Quantum Science and Technology*, vol. 5, no. 3, p. 034016, 2020.

[13] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "T|ket>: A retargetable compiler for NISQ devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014003, 2020.

[14] R. LaRose, "Overview and Comparison of Gate Level Quantum Software Platforms," 2019.

[15] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics," in *PLDI*. ACM, Jun. 2020.

[16] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton, and R. Drechsler, "QMDDs: Efficient quantum function representation and manipulation," *IEEE TCAD*, vol. 35, no. 1, pp. 86–99, 2016.

[17] M. Amy, D. Maslov, and M. Mosca, "Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning," 2014.

[18] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," 1982.

[19] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The epfl logic synthesis libraries," *preprint arXiv:1805.05121*, 2018.

[20] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *DAC*, 2003, pp. 318–323.

[21] A. De Vos and Y. Van Rentergem, "Young subgroups for reversible computers," *Advances in Mathematics of Communications*, vol. 2, no. 2, p. 183, 2008.

[22] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM Journal on Computing*, vol. 18, no. 4, pp. 766–776, 1989.

[23] B. Schmitt, M. Soeken, G. De Micheli, and A. Mishchenko, "Scaling-up ESOP Synthesis for Quantum Compilation," in *ISMVL*, 2019.

[24] H. Riener, R. Ehlers, B. d. O. Schmitt, and G. De Micheli, "Exact synthesis of ESOP forms," in *Advanced Boolean techniques*. Springer, 2020, pp. 177–194.

[25] M. Soeken, F. Mozafari, B. Schmitt, and G. D. Micheli, "Compiling Permutations for Superconducting QPUs," in *DATE*, 2019.

[26] M. Soeken, M. Roetteler, N. Wiebe, and G. D. Micheli, "LUT-Based Hierarchical Reversible Logic Synthesis," 2019.

[27] B. Schmitt, A. Javadi-Abhari, and G. De Micheli, "Compilation flow for classically defined quantum operations," *DATE*, 2021.

[28] A. Kissinger and A. M.-v. de Griend, "CNOT circuit extraction for topologically-constrained quantum memories," 2019.

[29] B. Schmitt, M. Soeken, and G. D. Micheli, "Symbolic Algorithms for Token Swapping," in *ISMVL*, 2020.

[30] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," 1995.