Quantum Science and Technology

RECEIVED
1 November 2019

REVISED
3 June 2020

ACCEPTED FOR PUBLICATION
9 June 2020

PUBLISHED
7 July 2020

PAPER

An open-source, industrial-strength optimizing compiler for quantum programs

R S Smith[1,2], E C Peterson[2,3], M G Skilbeck[2] and E J Davis[2]

[1] HRL Laboratories, 3011 Malibu Canyon Rd #4797, Malibu, CA 90265, United States of America
[2] Rigetti Computing, 775 Heinz Ave, Berkeley, CA 94710, United States of America
[3] Author to whom any correspondence should be addressed.

E-mail: peterson.eric.c@gmail.com

Keywords: quantum compilation, quantum computing, circuit optimization, Common Lisp

Abstract

Quilc is an open-source, optimizing compiler for gate-based quantum programs written in Quil or QASM, two popular quantum programming languages. The compiler was designed with attention toward NISQ-era quantum computers, specifically recognizing that each quantum gate has a non-negligible and often irrecoverable cost toward a program's successful execution. Quilc's primary goal is to make authoring quantum software a simpler exercise by making architectural details less burdensome to the author. Using Quilc allows one to write programs faster while usually not compromising—and indeed sometimes improving—their execution fidelity on a given hardware architecture. In this paper, we describe many of the principles behind Quilc's design, and demonstrate the compiler with various examples.

1. Introduction

Noisy intermediate-scale quantum (NISQ) computers are an active area of research. New quantum computer architectures are sometimes the result of incremental improvements in the manufacturing process, and at other times are paradigm-shifts in the qubit technologies themselves. While each new architecture is universal in a computational sense, the impermanence of their designs challenges one's ability to write software for them. As has been the case with classical computers, the role of a compiler is to attenuate this challenge. Software for a quantum computer is ideally written in the manner that is simplest and most straightforward to the programmer, without necessarily requiring knowledge of the particulars of the target architecture. It is then the job of the compiler to produce both an efficient and an appropriate expression of this software which accounts for the details of the target architecture.

In this paper we present Quilc, an open-source[4] software application used to compile quantum programs written in Quil [3, 24] into an optimized program that is expressed in the native operations of a target quantum computer architecture. Quilc does not require—and indeed has no means to accept—instruction from the user on a fine-grained compilation strategy. Instead, it consumes a simple description of the architecture for which Quilc must compile the user's program. The architecture description language is general enough to handle most[5] manufactured gate-based computer architectures to date, and anticipates new ones. For these reasons, we say Quilc is *automatic* and *retargetable*. Quilc is also more than a desk calculator—a convenience to avoid doing manual, repetitive calculations—as it acts as a repository of knowledge about the compilation of programs, and it is able to synthesize this information to discover non-trivial expressions of a quantum program. We provide examples of this in section 5. It is also production-grade, and is deployed as an essential component of Rigetti Computing's software stack.

The structure of the paper is as follows. First, in section 2 we provide an overview of Quilc, including a mathematical formulation of quantum architectures as they pertain to compilation. This formalism is used in section 3 to describe how Quilc achieves retargetablity, a high-level overview of which is presented in

[4] The source code of Quilc is licensed under the Apache 2.0 license. The source code can be found at http://github.com/rigetti/quilc. This document refers to Quilc version 1.12.1.
[5] Currently, architectures with particularly exotic gates acting on 3+ qubits are not well supported.

© 2020 IOP Publishing Ltd

original	⇝	layout	⇝	nativization	⇝	optimization
				RX(pi/2) 5		
				RZ(pi/2) 5		
				RX(-pi/2) 5		RZ(-pi/2) 5
				CZ 5 4		RX(pi/2) 5
CNOT 0 4	⇝	CNOT 4 5	⇝	RZ(pi) 4	⇝	CZ 5 4
				RX(pi/2) 5		RZ(pi) 4
				RX(0.0) 4		RX(-pi/2) 5
				RZ(-pi/2) 5		RZ(pi/2) 5
				RZ(0.0) 4		
				RX(-pi/2) 5		

Figure 1. Compilation of the program `CNOT 0 4`. The layout stage looks for an optimal initial mapping of logical qubits to physical qubits: in this example, the compiler has opted to relabel qubit 0 as qubit 4, and qubit 4 as qubit 5, so that in the (here unspecified) chip topology the two-qubit interaction can occur without introducing `SWAP` instructions. The nativization stage converts any non-native gate into a native gate; in this example `CNOT` is first compiled to `RY`, `Z`, and `CZ` gates, and then finally nativization compiles those `RY` and `Z` gates into native `RX`, `RZ` and `CZ` gates. The final optimization stage here rewrites several `RX` or `RZ` instructions acting on the same target qubit into fewer instructions.

```
DECLARE beta  REAL
DECLARE gamma REAL
DECLARE ro    BIT[3]
H 0                                  RX(gamma) 0
H 1                                  RX(gamma) 1
H 2                                  RX(gamma) 2
CPHASE(beta) 0 1                     MEASURE 0 ro[0]
CPHASE(beta) 0 2                     MEASURE 1 ro[1]
CPHASE(beta) 1 2                     MEASURE 2 ro[2]
```

Figure 2. A typical, small instance of a 'QAOA' quantum program. The instructions `H`, `CPHASE`, `RX`, and `MEASURE` respectively put the qubit registers into superposition states, commingle the registers, attempt to 'unmix' them, and measure the residual commingling.

figure 1. In section 4 we consider two compilation stages (the 'addressing' and 'compression' stages) in more detail. We follow this in section 5 with a few non-trivial examples, which make use of many of the features present in Quilc. In section 6 we investigate the performance of Quilc on a set of benchmarks. Finally, in the appendix we consider some implementation details which may be of interest to compiler authors or potential contributors to Quilc, including examples of our 'compilation subroutine' domain-specific language, additional features of Quilc, and a short history of the project's development.

2. A quantum compiler target

The structure of Quilc is informed by the task at hand: it must conform to the features and constraints of gate-based quantum computational devices (i.e., the shape of its output), and it must understand the features and constraints of the quantum programming language, Quil (i.e., the shape of its input). Gate-based quantum computational devices are made up of quantum resources (typically qubits) and support operations which affect the state of a subsystem, commingle the states of two or more subsystems, or collapse and copy the state of a subsystem for classical interpretation. The Quil language itself provides support for all of these operations in an assembly-like format: the system's resources are addressed by 'quantum registers', the individual assembly instructions correspond to the aforementioned operations, and there are mechanisms for further specifying other classical requirements (e.g., memory to store user-defined parameters) and their operation (figures 2 and 3).

To remain portable, the Quil language is also designed to be hardware-agnostic: it makes no particular assumptions on the availability of resources or what particular operations they support. Instead, its execution semantics are formally specified against a mathematical backend in such a way that makes it clear how to abstractly simulate the effects of a Quil program. At the other extreme, physical devices do labor under a host of severe constraints: there is a fixed (and typically small) number of resources, there are only a few very particular instructions which the device can enact on those resources, operations are subject to

```
DEFCIRCUIT RESET q scratch:
    MEASURE q scratch
    JUMP-UNLESS @done scratch
    X q
    LABEL @done
```

```
DEFGATE U(%alpha, %beta):
    cis(2*pi*%alpha),  0, 0, 0
    0, cis(-2*pi*%alpha), 0, 0
    0, 0,  cis(-2*pi*%beta), 0
    0, 0, 0,   cis(2*pi*%beta)
```

Figure 3. On the left: a Quil snippet demonstrating the implementation of a RESET instruction as applied to a qubit q. The instruction X is the quantum equivalent of a NOT instruction. On the right: a Quil snippet defining a custom instruction.

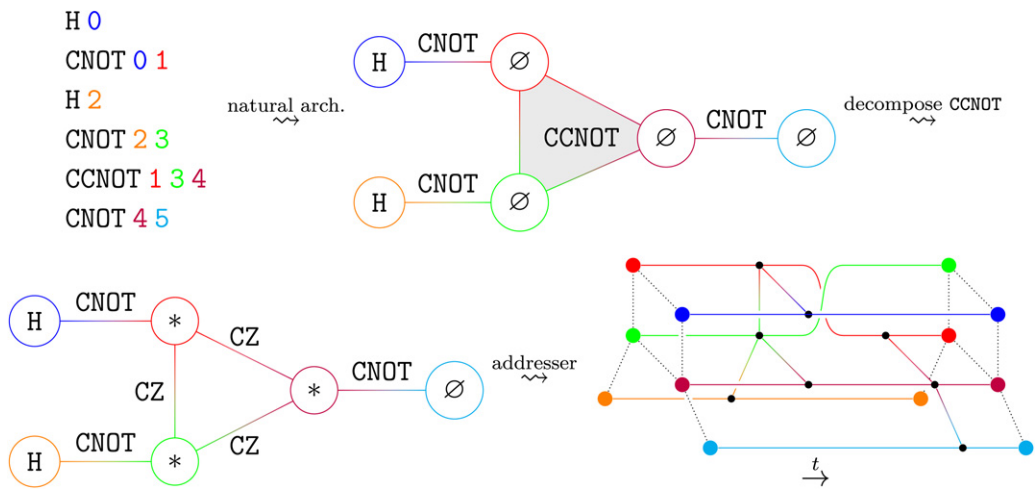

Figure 4. An example input program, its associated natural architecture, its native architecture after decomposing the CCNOT into two-qubit interactions, and the effect of the addresser (cf section 3, only CZs displayed) when targeting a particular device shaped like a square with two arms.

other requirements (e.g., spatial locality: distant qubits are typically unable to directly interact), and operations may be error-prone.

The role of a quantum compiler is to convert an abstract specification of such a quantum program into machine-executable bytecode, interpretable by the classical electronics which manipulate the engineered quantum system. The compilation process cleaves roughly into two parts: the conversion of the program's quantum aspects to a form that comports with the constraints of the engineered quantum system, followed by the conversion of the classical aspects to a form that comports with the structure of the control electronics. The quantum concerns are primarily those announced above, and while we will chiefly concern ourselves with them, we also name some classical concerns for completeness: memory management, timing and synchronization, classical communication, as well as a host of others. Quilc's approach to the satisfaction of the quantum constraints is to order them by severity:

(a) Any operations of large arity must be decomposed into an arity supported by the system.
(b) Any operations between non-interacting or indirectly interacting regions must be spatially rearranged to accommodate the system's preferred set of interactions.
(c) All operations must be written in terms of the set of operations (presumed universal) that the system can perform.
(d) The program should be structured so as to avoid suffering performance penalties.

The first three requirements are all equally important, in the sense that the program cannot be executed on a given physical device if any are not satisfied—but on a heterogeneous device, it is not possible to discern what operations the system can perform without first resolving (b), which in turn requires resolving (a), so that only then can (c) be redressed. Quilc leaves the resolution of (d) for last, since it can be satisfied 'by degrees': for instance, it is generally better for (d) to use fewer instructions, but there are also no clear hard limits to either of the maximum count tolerable or the minimum count achievable.[6]

[6] The decision to resolve arity and addressing before circuit optimization does incur a loss of information. In particular, certain high-level circuit identities may not be obvious after the program has been lowered to elementary operations. However, in many instances such high-level circuit optimizations are most appropriately handled by library authors, in concert with the low-level optimizations of the compiler. In this respect, programming a quantum computer does not differ so much from programming a classical one.

```
{"1Q": {
  "0": {
   "gates": [
    {"operator": "RX", "parameters": [1.5707963267948966], "arguments": [0]},
    {"operator": "RZ", "parameters": ["_"],                "arguments": [0]}]},
  "1": {
   "gates": [
    {"operator": "RX", "parameters": [1.5707963267948966], "arguments": [0]},
    {"operator": "RZ", "parameters": ["_"],                "arguments": [0]}]}},
 "2Q": {
  "0-1": {
   "gates": [
    {"operator": "CZ", "parameters": [],                   "arguments": [0, 1]}]}}}
```

Figure 5. A simple target architecture in serialized form. Note that $\pi/2 = 1.57\ldots$.

There is a natural data structure which captures these constraints of the target architecture and which forms the backbone of Quilc. To describe it, fix a set S of quantum resources. A target architecture amounts to specifying the available *interacting subsets* $S' \subseteq S$, each indicating a collection of resources that are permitted to interact—for instance, a pair of qubits on which one can perform a two-qubit gate. Let us make the additional assumption that if S' is an interacting subset, then every further nonempty subset $S'' \subseteq S'$ is also an interacting subset. Under this assumption, such a collection of interacting subsets forms a simplicial complex Σ [25, section 3.1] with S as its set of vertices (or 0-simplices), interacting pairs as its edges (or 1-simplices), interacting triplets as its 2-simplices, and so on. Additionally, we tag each simplex with a set of *instructions*, each corresponding to a specific means by which the interacting set can evolve as an ensemble. Finally, each such instruction is tagged with metadata, e.g., a matrix defining the associated unitary transformation, the average fidelity of the device's execution of the instruction compared to the ideal, or the temporal duration of the instruction execution on the device.

We have arranged the description of the target architecture into these tiers because this reflects the kind of information needed as input to the four compilation constraints:

(a) The limit on the arity of an instruction corresponds to the dimension of the largest simplex.
(b) The simplicial complex structure of interactions describes both the spatial constraints to which a quantum program is subject, as well as the pathways by which information can be rerouted or permuted in order to satisfy these constraints.
(c) The instruction tags indicate into which gateset the program's components must be compiled.
(d) Metadata can be used to make decisions that boost overall program performance. For instance, fidelity information might prefer one region of the device over another, or a particular instruction decomposition over another.

As an example, we have included in figure 4 an example input program, an architecture Σ for which it is fully native, and an example target architecture. As a further simple example, we include in figure 5 a serialized such target architecture (without any extraneous metadata, like instruction fidelities).

3. The large-scale structure of Quilc

Let us now turn to our high-level overview of Quilc. Quilc segments the satisfaction of the above constraints into the following stages:

Lexing/parsing. First, Quilc parses plain-text input into a syntax tree.

Control-flow graph construction. It then segments the input program into classical and non-classical components, so that consideration of the effects of classical instructions can be deferred to the compiler backend. The primary mechanism for 'hiding' these instructions (e.g., jumps) is to store them as labels in a control flow graph.

Addressing. Quilc resolves constraints (a), (b), and (c) by walking the graph of quantum instructions, ordered by resource dependence, in a breadth-first manner and tracking a mapping from logical quantum resources (i.e., those specified in the user's program) to physical quantum resources (i.e., those actually available on the device).

Compression. Lastly, Quilc gives attention to (d) by finding sequences of instructions that act on overlapping resources through a kind of depth-first walk and applying reduction techniques (e.g., a peephole rewriter) to the paths appearing during the walk. See figure 6 for an example.

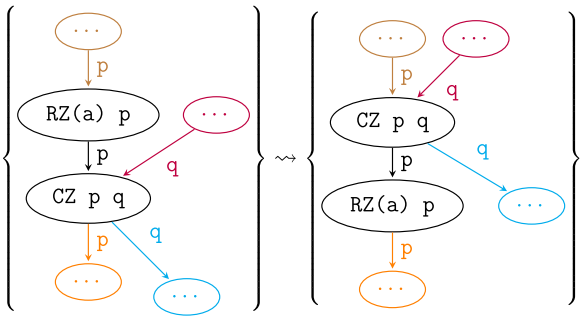

Figure 6. An example graph-based rewriting rule, commuting an RZ past a CZ. The vertices are labeled by instructions, the edges are labeled by the resource shared by the source and target, and the direction indicates which instruction precedes the other in the program. The black region consists of vertices which are matched, destroyed, and recreated; the various colored regions are detached from the old black vertices and reattached to the new black vertices as indicated.

There are two important points to note. First, the addressing and compression stages require intensive computation to fully explore the graph: for example, deciding whether the addressing problem admits a solution without the insertion of SWAP instructions is an instance of the (NP-complete) subgraph isomorphism problem [21]. Instead, we employ heuristics, randomization, and approximate algorithms to tame their computational complexity; see section 4 for more details, as well as figures 14(a) and (b) in appendix A for some empirical analysis of these approximations. The second important point is that although the addressing and compressing stages crawl the program in a highly different way, the actual manipulation of quantum instructions is similar in each. This promotes the segmentation of these stages into small, interruptible *compilation subroutines* which adopt a uniform interface. This API which we expect compilation subroutines to provide consists of

- A literal subroutine, which consumes some fixed number of instructions and some data about the state of the compiler, and which emits a sequence of instructions which may replace its input. The subroutine is allowed to be partially defined: compilation subroutines can signal that they are not applicable in a given situation by employing an interrupt which is handled appropriately by the caller.
- A description of the kinds of instructions that the subroutine can consume.
- A description of the kinds of instructions (and, ideally, their counts) that the routine can emit.

Equipped with this extra data, Quilc can decide whether a particular subroutine falls into one of two privileged classes (or neither):

Nativizers. A compilation routine is relevant for nativization when it consumes a single instruction and when its output belongs to the native interactions of Σ (perhaps after further applications of other nativizers).

Optimizers. A compilation routine is relevant for optimization when it both consumes and emits instruction sequences which belong to the native interactions of Σ and its emitted sequences have better execution properties than its inputs.

The addressing step employs the first class of compilation subroutines in order to convert the input program's non-native instructions to instructions that are native for Σ's interactions. The compression step makes use of both of these special classes: the optimizers are directly relevant to the compression of instruction sequences, but it is also fruitful to destroy some of the structure of a sequence of instructions by considering their holistic effect and to re-nativize that effect.

This kind of design lends itself to the support of a few features:

Planning. Verifying that a particular long sequence of reductions gives an optimal strategy (from the perspective of constraint (d)) is computationally expensive, frequently remitted to heuristic, and costly to guess incorrectly. It is less expensive to make analogous decisions about individual, simple reductions, and so interruptibility permits the compiler a greater degree of flexibility in quickly planning its next best move.

Internal reusability. Requiring that subroutines leave the overall compiler in a good intermediate state puts dramatic limitations on the API to which they conform. From the perspective of Quilc, this tends to make such subroutines suitable for use at various stages of compilation. It also promotes a separation of concerns between the code responsible for crawling the input program (as described above in, e.g., 'addressing' and 'compression') and these subroutines, so that the crawlers are written in such a generic way that their specification does not directly depend on knowing the set of subroutines to be applied to the input program.

Ease of authorship. The same limitations on the API means that the author of quantum-specific subroutines need not concern themselves with the precise implementation of the crawlers—or, in Quilc's case, even how to instruct the crawlers that they should make use of a new subroutine.
External reusability. It is possible to wrap routines provided by external compilation libraries via this API, so that Quilc can make seamless use of their specialized routines without reimplementation or serious diversion. One such example is `tweedledum`, a quantum circuit optimizing library which provides an efficient routine for compilation of gates that can be represented as permutation matrices [8].

4. A closer look at addressing and compression

Much of the 'action' of the compiler occurs in the addressing and compression stages. Indeed, these are the most computationally intensive stages of compilation, and the techniques employed are critically responsible for the quality of the output of the compiler (e.g., with respect to gate depth). In this section we give brief summaries of the underlying techniques of these two compilation stages, along with points of contact in the literature.

4.1. Addressing

In principle, the addressing stage may be resolved by any sequence of circuit transformations which preserve the semantics of the source program and which result in an output program conforming to the constraints of the hardware. However, the set of such possible transformations is immense. Here we constrain ourselves to two classes of transformations: those which serve to translate gates into native constituents (e.g., the aforementioned 'nativization' routines) and those which manage the assignment of logical to physical qubits (known as 'qubit allocation' in the literature).

A number of general-purpose techniques for qubit allocation have been proposed; we briefly mention some here. In [21], the authors consider four sorts of program transformations ('virtual `CNOT`s', reversals, bridges, and swaps), formulating the problem precisely in these terms and presenting heuristics to approximate the optimal sequence of transformations to solve the qubit allocation problem.

In [28], the authors first produce an initial decomposition of the source circuit (consisting of 1Q and 2Q gates) into layers, then construct an initial qubit mapping, and then finally identify swap operations between layers via A* search using a cost function which may incorporate look-ahead to successive layers. The method of [5] uses a similar layered decomposition, but with different heuristics for constructing the initial mapping and for selecting swaps between layers.

Alternative approaches have formulated the addressing problem in a form amenable to solution by off-the-shelf solvers. For example, in [12], the authors formulate qubit allocation as a satisfiability modulo theories (SMT) problem which may be solved by an SMT solver. Along similar lines, [27] proposes and evaluates a formulation which is solvable by temporal planners.

Core to the approach taken by Quilc is a representation of the source program which expresses the constraints, with respect to hardware usage, implicit in the linear source program. Here, the source instructions are taken as the vertices of a directed, acyclic graph, with edges expressing resource conflicts (whether classical or quantum) between logically successive operations. The addressing pass proceeds in a greedy fashion, consuming the source program in topological order while maintaining a certain amount of state, including a partial logical-to-physical qubit mapping, estimated swap costs between pairs of qubits, and a buffer of emitted instructions operating on physical qubits.

In this scheme, gate applications involving a number of qubits exceeding the underlying arity of the device (e.g., 3Q gates for Rigetti's hardware) are first translated to an equivalent series of smaller-arity operations by means of any number of nativization routines, as described in section 3. We note that the goal here is not to find an 'optimal' sequence of native operations, but rather to quickly find a viable realization of the gate in native terms. However, Quilc does attempt to select the translation so as to be in harmony with the particular qubits' native gate sets: for instance, `ISWAP`-based decompositions are preferred to `CNOT`-based decompositions when the native gate sets includes the `ISWAP` gate and not `CNOT`.

As each low-arity operation (e.g., 1Q and 2Q gates on Rigetti's hardware) is processed, the logical-to-physical qubit mapping may be updated, either by assigning a logical qubit to a currently unassigned physical qubit, or via the introduction of SWAP operations in order to satisfy addressing constraints. At each such decision point, the ambition of Quilc is to select the action which minimizes the total cost of the final scheduled program.

To this end, Quilc employs heuristics along two axes. The first are *cost heuristics*, which, given a logical Quil program and a compilation target, determine a cost indicating the 'badness' of the program on the underlying hardware. At present, there are two of these available: a duration-based heuristic, informed by

the underlying gate times of the target architecture, and a fidelity-based heuristic, informed by the reported gate fidelities of the target architecture. The second axis consists of *search heuristics*, which are used to select from available swap operations in order to assign a logical gate to physical qubits in a cost-minimizing fashion. These include both A* search as well as greedy search heuristics.

A challenge with such an approach is that `SWAP`s inserted cheaply at one point in a program may end up being costly with respect to later operations. To account for this, both the duration-based and the fidelity-based cost heuristics incorporate a look-ahead: here the cost associated with a Quil program depends not just on the next instructions, but also more weakly on those following them (via an exponential discounting factor). Here, the motivation is to dampen the miserliness of the (otherwise greedy) addressing strategy.

4.2. Compression

During the compression stage, Quilc employs two kinds of rewriting strategies:

(a) Directly apply a peephole rewriter to an instruction sequence.

(b) Convert a (pure quantum) instruction sequence to a composite matrix, rewrite the matrix as native instructions, and apply a peephole rewriter to the resulting sequence.

Both of these are best served by sequences with two properties:

Lengthiness. Long sequences provide more opportunities for the rewriter to act and give the bounded nativization routines better odds of producing a shorter sequence.

Resource-sparsity. Sequences which act only on a few resources have correspondingly fewer false negatives in the form of instructions which are nonadjacent in the sequence but which could be commuted next to one another.

With this in mind, the compressor has been designed to produce contiguous sequences of instructions with these properties.

The compressor first arranges the instructions in a program into a dependency graph by resource usage. It then begins forming subgraphs, tagged by resource utilization, and peforms a topological walk of the instruction graph while adhering to the following rules:

- If this instruction's resources do not meet those of any subgraph and it does not contain a forbidden resource, start a new subgraph containing this instruction, and restart consideration of the next instruction in the walk.
- Otherwise, this instruction's resources meet one or more existing subgraphs or are forbidden. Compute the sum of their resource tags with this instruction's resources.
- If the resource sum contains a forbidden resource collection *or* if the resource sum is larger than the compressor's limit, then do nothing with this instruction for now. Remove each met subgraph from the overall graph. For each met subgraph:
  - If the resource sum contains a prohibited resource collection, then mark this subgraph's resource tag as forbidden. If the resource sum is larger than the compressor's limit, then mark the sum as forbidden.
  - Linearize the contents of the subgraph into a sequence, and pass that sequence to the peephole rewriter.
  - Re-walk the instructions emitted by the peephole rewriter (i.e., try to form subgraphs out of them).
  - Unmark any forbidden resources added in this step.

Write this instruction, as well as any instructions in any met subgraphs (which may be newly formed, as in the above loop), out to the end results.

- Otherwise, merge the subgraphs which meet this instruction, add it to the newly formed subgraph, tag the subgraph with the resource sum, and proceed to the next instruction in the walk.

This walk is similar in effect to the walk considered in Iten, Soetter, and Werner [9]. However, ours is somewhat less thorough, since it separately walks the graph and applies template rewriting, and since it does not perform backward matching. It partially makes up for these deficiencies in its output by its simplicity of implementation.

In the course of the compressor's graph-walking, a given instruction may be considered by the peephole rewriter as many times as there are subresources which contain the instruction and which are contained by the subgraph's tag. By installing a limit to the size of subgraph tags which the compressor will consider, this value becomes bounded. In practice, even a small such limit (e.g., less than four qubits) has good run time properties without appreciable decline in output quality.

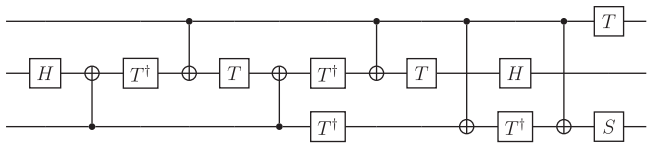

Figure 7. CCNOT on a fully-connected chip.

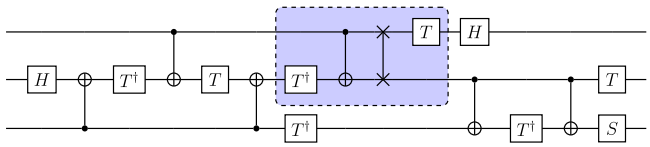

Figure 8. CCNOT with nearest-neighbor connectivity, placing an SWAP on the top two qubit lines. The entire highlighted subcircuit may be translated to at most 3 CNOT gates.

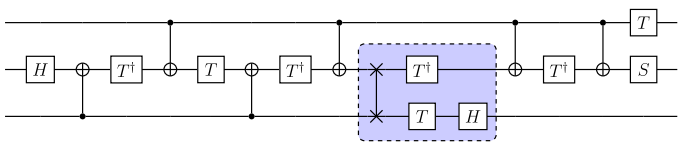

Figure 9. CCNOT with nearest-neighbor connectivity, placing an SWAP on the bottom two qubit lines. The entire highlighted subcircuit may be translated to at most 3 CNOT gates.

5. Long-form examples

In this section, we demonstrate the above considerations via a few practical examples which highlight the influence of target architecture and the ubiquitous role of compilation subroutines in the compilation process. The first two subsections consider compilation of the Toffoli gate, a well studied example which has been implemented on a variety of architectures and for which optimal decompositions are known [19]. Our aim here is to demonstrate how various compilation routines may be combined to realize CCNOT across several device architectures. In the third subsection, we consider state-aware compilation applied to an example from computational chemistry.

5.1. SWAP recombination with different targets

One of the basic tasks of the 'addressing' stage of the compiler is to construct a mapping from logical qubits, as expressed in the source program, to physical qubits, as realized in a specific architecture. A guiding principle here is that constraints in physical qubit connectivity may be satisfied through the addition of appropriate SWAP instructions. This introduction of SWAP gates comes at a price: namely, an increase in the total number of logical operations performed. This is further complicated by the demands of nativization, since for many architectures of interest SWAPs must be translated to native operations. Bounds for the complexity of the resulting native instruction sequence have been considered in the literature. For example, it is known that SWAP requires 3 CNOT gates [26]. On the other hand, an arbitrary two-qubit unitary operator is equivalent, up to a global phase factor, to one expressed as a circuit with at most 3 CNOT operations [20].

Thus, for many architectures of interest, the demand of nativization presents itself as an opportunity when selecting SWAP targets, since the native gate cost of a single SWAP gate is the same as that of any subcircuit consisting of the SWAP gate and adjacent gates, if these are on the swapped qubits. For example, figure 7 shows one realization of a CCNOT gate on a fully-connected chip, which is optimal in the sense that involves a total of 6 CNOT operations [19]. If nearest-neighbor connectivity is imposed, then there is a natural decision of where an SWAP should be inserted. Considering that nativization can convert any two-qubit unitary subcircuit into an equivalent one involving at most 3 CNOT gates, placing an SWAP on the first two qubit lines (cf figure 8) is preferred to placing on the second two lines (cf figure 9), as this is cheaper by a CNOT gate.

```
(define-compiler CCNOT-to-CNOT
    ((input ("CCNOT" () q0 q1 q2)))
  (inst "H"    ()       q2)
  (inst "CNOT" ()       q1 q2)
  (inst "RZ"   '(-pi/4) q2)
  (inst "CNOT" ()       q0 q2)
  (inst "RZ"   '( pi/4) q2)
  (inst "CNOT" ()       q1 q2)
  (inst "RZ"   '(-pi/4) q2)
  (inst "CNOT" ()       q0 q2)
  (inst "RZ"   '( pi/4) q1)
  (inst "RZ"   '( pi/4) q2)
  (inst "CNOT" ()       q0 q1)
  (inst "H"    ()       q2)
  (inst "RZ"   '( pi/4) q0)
  (inst "RZ"   '(-pi/4) q1)
  (inst "CNOT" ()       q0 q1))
```

Figure 10. Compilation subroutine for implementing CCNOT via CNOT. See appendices A.1 and A.2 for a description of the syntax.

Table 1. Two-qubit native gate complexity of CCNOT with linear nearest-neighbor connectivity. Gate counts are shown both with and without the use of the CCNOT-to-CNOT compilation subroutine. Note the uniform improvement in gate count introduced by the addition of the subroutine, whether or not CNOT appears in the target instruction set.

Native gates			Gate count	
CZ	ISWAP	CPHASE	Without	With
✓			9	7
	✓		12	9
		✓	8	8
✓	✓		7	6
✓		✓	8	6
	✓	✓	10	9
✓	✓	✓	7	6

The Quilc addresser makes use of this information, and more, when selecting SWAP placements. In practice, we observe that this 'SWAP recombination' trick is compatible with additional optimizations. For example, when compiling CCNOT 0 1 2 to a chip supporting only nearest-neighbor connectivity, Quilc is able to produce native circuits containing only 7 CNOT gates, which is less than the 8 suggested by figure 8.

5.2. Native targets for CCNOT

One of the guiding philosophies of Quilc is that the burden of deciding whether a given compilation subroutine should be preferred to another in some specific context need not be borne by the quantum programmer. Instead, what is specified by the user is a set of hardware constraints.[7] Presented with this information, Quilc performs the tasks of selecting those compilation subroutines suited to the problem at hand. To demonstrate the flexibility afforded by this approach, we consider a simple experiment in compiling CCNOT, and in particular consider the effect that the choice of native gateset and availability of particular compilation subroutines has on the resulting gate complexity.

Recalling our previous example, we note that the circuit expressed in figure 7 is embodied in Quilc as a compilation subroutine, CCNOT-to-CNOT (cf figure 10 and the discussion in appendix A). Strictly speaking, specific subroutines such as this one are not needed, as Quilc supports fully generic techniques such as the recursive 'Quantum Shannon Decomposition' of [18]. Nonetheless, specific compilation subroutines may be preferred to general ones when they offer a reduction in final native gate counts.

In what follows, we consider a three qubit chip with linear connectivity, so that qubit 0 is connected to 1, and qubit 1 is connected to 2. Amongst the possible two-qubit operations, we restrict attention to CZ, ISWAP, and CPHASE, and for each subset of these we consider a target architecture in which those operations are native across connected qubits. With respect to compilation subroutines, Quilc has several enabled by default, and we consider only the effect of including or excluding CCNOT-to-CNOT from this set. In all instances, the compiler is able to translate CNOT gates to the native gate of choice and is able to take advantage of 'SWAP recombination' as described earlier.

In table 1 we show the complexity of CCNOT in terms of native two-qubit gate counts. In all instances, it is always advantageous to incorporate the special information provided by CCNOT-to-CNOT. The best results occur for a device supporting CZ along with one of {ISWAP, CPHASE}, which results in a circuit using only 6 two-qubit gates.

[7] Users do have the option of providing hints to the compiler about properties it might exploit, e.g., via command line arguments to quilc executable, or in Quil programs via PRAGMA directives.

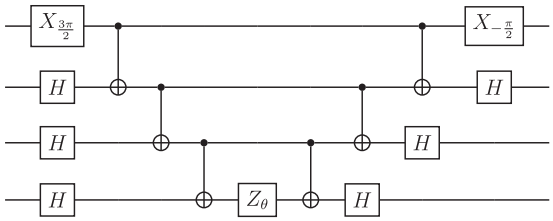

Figure 11. UCCSD ansatz for H_2 in the STO-3G basis.

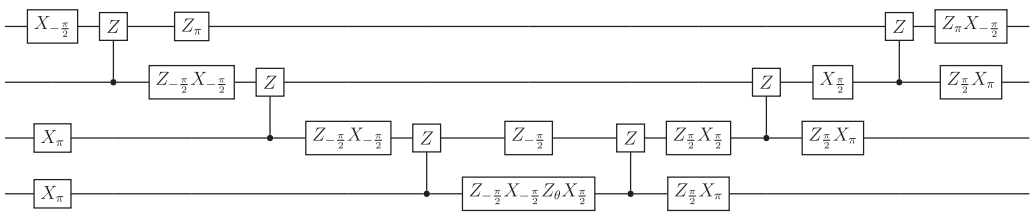

Figure 12. UCCSD ansatz for H_2 in the STO-3G basis, naively compiled to Rigetti's native gates, requiring 6 CZ instructions.

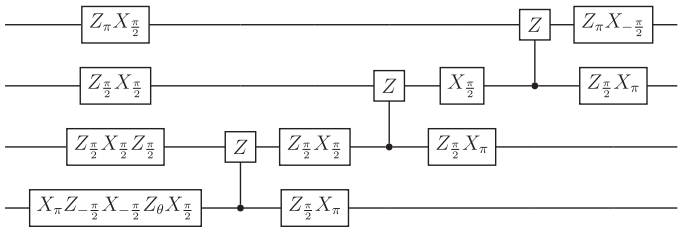

Figure 13. UCCSD ansatz for H_2 in the STO-3G basis, compiled to native gates with state-aware optimizations enabled, requiring only 3 CZ instructions.

5.3. State-aware compilation

Many compilation routines express simple circuit equivalencies, which depend only on the syntactic structure of some instruction or block of instructions. However, Quilc is capable of providing more information to those routines which might benefit from this context. When *state-aware* compilation is enabled, Quilc performs a partial simulation of addressed Quil instructions during the compression stage, until such a simulation is obstructed. Typical such obstructions include:

(a) The state becomes highly entangled. The time and space costs of tracking a quantum state scales exponentially with the number of entangled qubits. Since the gains in output quality do not also scale exponentially, Quilc limits this cost by discarding components of state information with entanglement beyond some fixed limit—by default, three qubits.

(b) A run-time data dependency is encountered. This makes compile-time optimization impossible, since such an optimization involves data unavailable at that point.

The results of this simulation are made available to additional compilation routines. Those routines which make use of this additional information are called *state-aware*.

As an example, note that in general a compilation subroutine translates a gate application or sequence of gate applications to some 'equivalent' sequence. Under most circumstances, the notion of equivalence here is that the corresponding unitary transformations, represented by the instructions, should be equal, perhaps up to some discretization error. However, when the quantum state is known prior to the execution of an instruction or block of instructions, the requirement of unitary equivalence may be relaxed. Indeed, given full information about the initial state, a sufficient notion of equivalence is that the resulting states be equal. The corresponding task, of preparing a target state given an initial state, is known as *state preparation*. Many instruction sequences which are not unitarily equivalent may be equivalent in this sense, and this increase in flexibility allows for additional synthesis techniques [16], [18, section 4].

Table 2. QASM benchmarks from [28] targeting the IBM qx5 architecture, performed in the same environment as figure 14(b). Compare with [28, table 1] and [5, table 4]. Columns labeled with an asterisk '*' hold data for circuits which were produced with the additional hypothesis that the quantum device begins in the ground state. The percentage in parentheses column indicates the rounded percentage reduction in 2Q *depth* by taking this assumption into account. Quilc is instructed to give up on applying state-aware methods to any component of the quantum state in which more than 3 qubits are entangled. Wall time is the elapsed real time of compilation from start to finish.

Filename	SWAPs added	Wall time (s)	2Q depth	Wall time∗ (s)	2Q depth∗ (%)
0410184_169	101	8.767	235	9.802	55 (77%)
3_17_13	11	0.485	28	0.79	23 (18%)
4_49_16	66	7.271	230	10.738	109 (53%)
4gt10-v1_81	44	5.311	143	8.771	86 (40%)
4gt11_82	5	1.705	28	1.387	7 (75%)
4gt11_83	4	0.809	22	0.837	0 (100%)
4gt11_84	3	0.436	15	0.544	13 (13%)
4gt12-v0_86	76	9.364	241	13.11	117 (51%)
4gt12-v0_87	73	10.018	231	12.33	116 (50%)
4gt12-v0_88	52	6.11	176	8.015	87 (51%)
4gt12-v1_89	67	7.353	234	9.621	91 (61%)
4gt13-v1_93	16	1.878	60	3.257	31 (48%)
4gt13_90	28	3.111	102	5.355	47 (54%)
4gt13_91	27	2.92	97	4.105	30 (69%)
4gt13_92	19	1.195	57	1.913	23 (60%)
4gt4-v0_72	66	7.307	218	12.685	88 (60%)
4gt4-v0_73	108	12.503	371	20.436	183 (51%)
4gt4-v0_78	67	7.194	233	10.774	77 (67%)
4gt4-v0_79	66	6.69	225	10.17	91 (60%)
4gt4-v0_80	57	5.896	185	9.777	87 (53%)
4gt4-v1_74	78	7.352	260	15.558	184 (29%)
4gt5_75	24	3.108	88	4.995	38 (57%)
4gt5_76	27	3.235	83	6.879	45 (46%)
4gt5_77	37	3.547	119	7.457	84 (29%)
4mod5-bdd_287	19	1.75	75	3.04	25 (67%)
4mod5-v0_18	20	1.23	69	1.781	46 (33%)
4mod5-v0_19	9	0.694	35	1.103	21 (40%)
4mod5-v0_20	6	0.875	21	1.956	14 (33%)
4mod5-v1_22	4	0.354	18	0.512	14 (22%)
4mod5-v1_23	19	2.024	68	2.855	37 (46%)
4mod5-v1_24	8	1.474	32	2.435	19 (41%)
4mod7-v0_94	51	4.291	155	6.248	89 (43%)
4mod7-v1_96	50	5.16	172	6.595	62 (64%)
C17_204	138	14.616	436	20.53	268 (39%)
aj-e11_165	49	5.232	144	7.957	59 (59%)
alu-bdd_288	22	1.382	78	1.721	39 (50%)
alu-v0_26	22	3.405	88	6.048	51 (42%)
alu-v0_27	8	1.239	30	2.459	15 (50%)
alu-v1_28	8	1.083	33	1.765	14 (58%)
alu-v1_29	7	1.098	33	1.683	17 (48%)
alu-v2_30	140	14.819	471	22.585	228 (52%)
alu-v2_31	146	15.981	428	21.86	182 (57%)
alu-v2_32	57	6.641	160	11.628	82 (49%)
alu-v2_33	8	0.965	28	1.637	16 (43%)
alu-v3_34	12	1.629	50	2.34	22 (56%)
alu-v3_35	11	1.1	36	1.86	28 (22%)
alu-v4_36	29	2.242	97	3.456	53 (45%)
alu-v4_37	8	1.247	31	2.318	16 (48%)
cnt3-5_179	63	6.172	139	6.467	45 (68%)
cnt3-5_180	139	9.593	385	16.345	183 (52%)
decod24-bdd_294	22	1.497	73	2.423	51 (30%)
decod24-enable_126	122	11.186	364	18.681	158 (57%)
decod24-v0_38	10	0.745	43	5.446	31 (28%)
decod24-v1_41	25	1.319	76	2.143	34 (55%)
decod24-v2_43	15	1.181	53	1.686	10 (81%)
decod24-v3_45	41	2.715	130	4.79	70 (46%)
ex-1_166	6	0.314	15	0.563	11 (27%)
ex1_226	1	0.564	6	0.786	0 (100%)
ex2_227	179	21.328	612	27.749	268 (56%)
ex3_229	100	9.691	367	18.069	223 (39%)
graycode6_47	0	0.638	5	0.774	0 (100%)

Table 2. Continued.

Filename	SWAPs added	Wall time (s)	2Q depth	Wall time∗ (s)	2Q depth∗ (%)
ham3_102	6	0.355	15	0.454	13 (13%)
ham7_104	72	9.683	307	15.797	149 (51%)
hwb4_49	74	6.803	227	11.394	119 (48%)
ising_model_10	0	1.665	20	2.31	19 (5%)
ising_model_13	0	2.862	20	4.738	19 (5%)
ising_model_16	0	3.748	20	6.182	19 (5%)
miller_11	12	0.55	47	0.384	0 (100%)
mini-alu_167	94	9.481	272	16.332	125 (54%)
mini_alu_305	62	4.388	173	7.373	39 (77%)
mod10_171	69	4.293	216	6.471	78 (64%)
mod10_176	41	5.314	164	8.637	81 (51%)
mod5adder_127	164	15.054	498	24.652	286 (43%)
mod5d1_63	6	0.591	25	0.804	11 (56%)
mod5d2_64	21	2.52	70	3.943	34 (51%)
mod5mils_65	10	1.489	34	2.825	20 (41%)
mod8-10_177	127	13.593	412	22.071	195 (53%)
mod8-10_178	85	8.877	310	13.051	146 (53%)
one-two-three-v0_97	90	6.717	263	12.593	167 (37%)
one-two-three-v0_98	48	4.321	148	7.224	73 (51%)
one-two-three-v1_99	36	4.06	128	6.711	61 (52%)
one-two-three-v2_100	21	2.998	69	4.28	26 (62%)
one-two-three-v3_101	19	1.656	72	2.529	36 (50%)
qft_10	28	2.805	44	3.491	0 (100%)
qft_16	86	7.495	111	9.063	6 (95%)
rd32-v0_66	8	0.495	26	0.877	21 (19%)
rd32-v1_68	8	0.501	26	0.865	21 (19%)
rd32_270	20	2.985	81	4.421	31 (62%)
rd53_131	130	15.426	439	24.783	186 (58%)
rd53_135	94	10.164	312	17.044	178 (43%)
rd53_138	38	4.54	122	5.982	48 (61%)
rd53_311	98	9.199	258	14.522	119 (54%)
rd73_140	68	7.166	211	10.845	104 (51%)
rd84_142	110	7.454	227	9.077	110 (52%)
sf_274	204	16.251	691	28.094	458 (34%)
sym6_316	98	7.566	269	14.242	99 (63%)
sym9_146	91	8.76	257	15.803	126 (51%)
sys6-v0_111	67	6.076	158	9.943	60 (62%)

Quilc incorporates a number of methods for state preparation, ranging from special purpose compilation subroutines (for example, in the case of one-qubit, two-qubit, or four-qubit systems), to generic subroutines which may recurse to one of these special cases. State preparation is fully compatible with the optimizations available through other compilation subroutines, although it is only applicable in circumstances in which the initial state may be effectively computed.

We demonstrate this by way of an example. In figure 11, we have a short program which expresses the unitary coupled cluster ansatz for deuterium, truncated to single and double excitation levels [11, section 7.1]. This program may be used as part of a hybrid classical/quantum variational algorithm such as described in [14]. In such hybrid algorithms, it is typical that a single parametric program 'template' is executed for a variety of numerical parameter values.[8]

On near term devices, numerical accuracy of variational methods reflects a trade-off between the sophistication of the ansatz and the resulting depth or complexity of the circuit. When compiled to native hardware without state preparation routines, the resulting program involves 6 two-qubit operations as in figure 12. When state preparation routines are allowed, knowledge that the program begins in the zero state $|0000\rangle$ is exploited to reduce this number to 3, as in figure 13.

In this example, one can see the practical effect of state-aware compilation is to reduce the initial portion of the circuit up until a point at which the state is no longer feasible to track. Here we remark that the presence of run-time parameters obstructs the partial-state simulation, and hence the reduction in gate count is primarily due to optimizations in the first half of the circuit. Additional details of state-aware compilation are discussed in appendix A.

[8] Quilc supports *parametric compilation*, in which a program may be compiled once and then executed with parameter values determined at run-time. This method of compilation can be used to greatly improve hybrid computation [10].

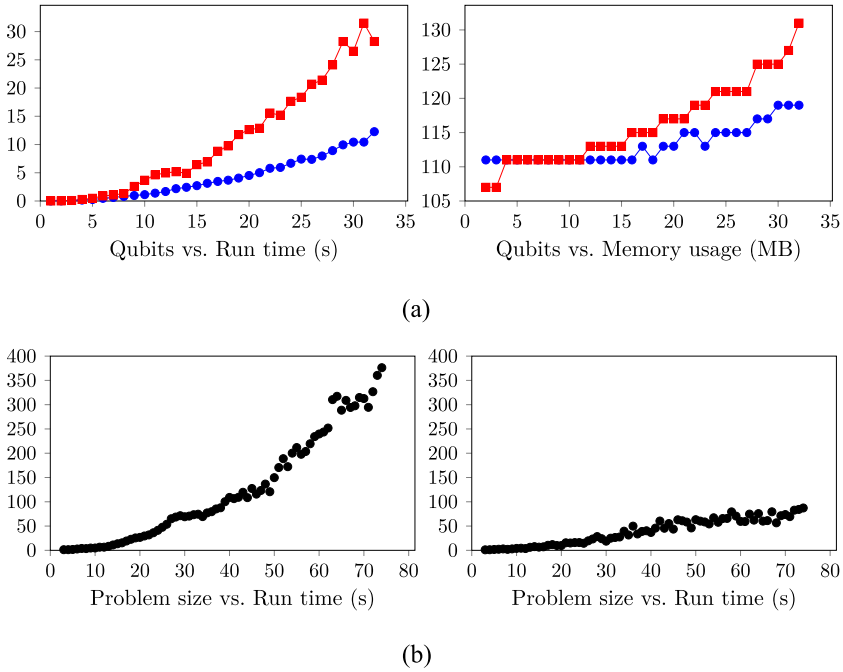

Figure 14. Run time performance measurements of Quilc. (a) Performance characteristics for compilation of QFT circuits of varying sizes to an Aspen-type device () and a fully connected device (). The left plot depicts the run time of Quilc, and the right plot depicts the minimum required heap size of Quilc (to within 2 MB). Quilc was compiled with Steel Bank Common Lisp (SBCL) version 1.5.8 and run on a 2018 MacBook Pro. (b) Performance characteristics for compilation of random 3-valent QAOA-type problems of varying sizes to a 128-qubit Aspen-type device, with and without commutation information (left and right respectively). Both plots depict the run time of Quilc, and both are best fit by a quadratic with small leading coefficient. Quilc was compiled with Steel Bank Common Lisp (SBCL) version 1.5.5 and run on a 2016 MacBook Pro. See figure 2 for an example of a QAOA-type program.

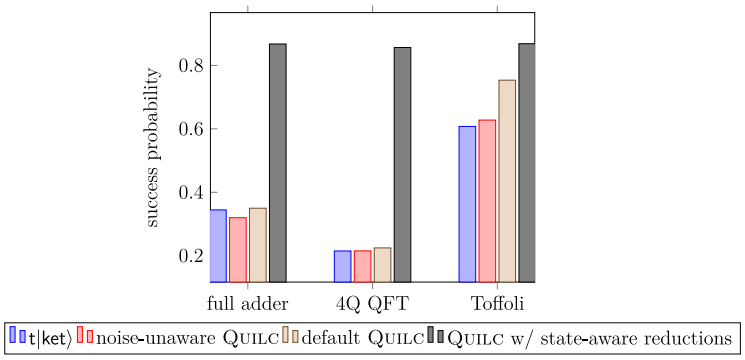

Figure 15. Comparison between the abilities of t|ket⟩ 0.5.5 and Quilc 1.17.0 at enabling *success probability*: the population percentage successfully transferred from the ground state to the program's target state (cf [12]), so that a perfectly executing device would earn a score of 1.0. Each program family was run on Rigetti's Aspen-8 device. No other noise mitigation mechanisms were used—indeed, readout error accounts for all of the observed failures in the QFT and Toffoli programs compiled using state-aware methods. For substantially larger programs, the gains from state-aware methods lessen but do not vanish (cf the rightmost column of table 2).

6. Performance

There are two meanings to the word 'performance' when it comes to compilers: its effectiveness at compiling a quantum program, and how many resources it consumes to perform that task.

To measure compilation effectiveness, we use the benchmarks from [28]—a suite of QASM files—that test Quilc holistically in two ways: with state-aware compilation disabled (i.e., the input and output of the compiler are equivalent unitaries), and with state-aware compilation enabled (i.e., the input and output of the compiler are only guaranteed to act identically on the ground state). Table 2 contains the benchmarks.

To date, Quilc has not been optimized for *run time* or *space* performance (i.e., how long it takes to compile and how much memory is required), and with an appropriate combination of engineering tricks, those metrics could be improved. At present, some competing compilers significantly outperform Quilc in terms of run time: t|ket⟩ reliably ran 100–200× faster than Quilc in a variety of comparison tests. Still, even in the absence of these optimizations, the run time statistics presented in figures 14(a) and (b) demonstrate the scaling laws associated with our approximate solvers deployed during the addressing and compressing stages. Wall clock performance is also included in table 2.

The results presented in table 2 may be compared with those presented in [5, 28]. In many cases our reported *2Q depth* shows an improvement, and our use of state-aware compilation further improves that metric. State aware-compilation generally comes at the expense of a longer compilation run time. In a number of cases state-aware compilation was able to reduce the program to the null-program consisting of no instructions. Figure 15 illustrates how these metrics translate into improved program performance on a live quantum device.

In general we see competitive performance metrics when compared with contemporary tools. Our state-aware compilation capabilities are at the time of writing not present in other tools and can thus be considered a first-in-class.

Acknowledgments

The authors would like to thank Rigetti Computing for its vibrant and stimulating workplace and Quantum Science and Technology (and B Peropadre in particular) for the invitation to contribute to their special issue. The authors are also grateful for the helpful comments of those readers of an early draft: M Appleby, J Bello-Rivas, M Brett, and M Hodson, as well as the anonymous referees. The authors are all themselves significant contributors to Quilc, but they have also all benefited from others' contributions to Quilc as an open-source project, which have bolstered the project generally and shaped some of the ideas presented in this paper. Further contributor acknowledgments can be found in Quilc's `ACKNOWLEDGMENTS.md` file. All authors contributed equally to this paper.

Authors contribution

Quilc is distributed as free and open source software under a permissive license. Contributions from anybody are highly welcome. Contributions come in a variety of forms, including bug reports, feature suggestions, UI/UX criticisms, platform testing, new circuit identities, or new compilation algorithms. The source code and issue tracker can be found here: https://github.com/rigetti/quilc

Appendix A. Implementation details

A.1. Implementation language

Quilc is written in ANSI Common Lisp, and can be extended with code written in languages with C ABI compatibility. Common Lisp was chosen because it provides a highly-performant substrate for both dynamic, interactive work as well as batch-mode computation, and offers convenient abstractions for implementing embedded domain-specific languages (DSLs), which are useful for many of the tasks of program analysis and manipulation. Quilc is also compatible with Rigetti's open-source quantum computer simulator, the quantum virtual machine [2].

One reason Common Lisp is particularly suitable for implementing DSLs is its syntax. Common Lisp has very regular syntax, most of which follows just a single syntactic construction:

$$(\langle \text{operator} \rangle\ \langle \text{operand} \rangle_1\ \cdots\ \langle \text{operand} \rangle_n).$$

That is, operators precede their operands and are surrounded by a pair of parentheses; see table A1 for a sample of syntax. Parentheses do not indicate precedence, but instead play double duty: as prefix-notation for the language, but also as syntax for (sometimes nested) lists. Since the syntax can be viewed as both notation and a data structure, Lisp code is ripe for both automatic generation and manipulation. For this reason, we call Lisp a *homoiconic language*. Interested readers can find lengthy discussions of these ideas at different levels in the literature [1, 13]. In the subsections to follow, excerpts from Quilc's source code will make use of this syntax.

The Quilc source code is separated into an application domain and a library domain. The application can be used to consume textual input as a UNIX command-line tool, or it can be used to provide a persistent

Table A.1. A comparison between usual algebraic syntax as found in mathematics and many programming languages and the syntax of Common Lisp.

Algebraic syntax	Lisp's syntax
$2(1-n)$	`(* 2 (- 1 n))`
$f(x, g(y), x+y)$	`(f x (g y) (+ x y))`
$k \mapsto k^3$ (i.e., $\lambda k{\cdot}\, k^3$)	`(lambda (k) (expt k 3))`
let $z = \sqrt{5}$ in $\zeta(z/2)$	`(let ((z (sqrt 5)))` `  (zeta (/ z 2)))`

server front-end. The library domain includes all of the routines for interpreting and manipulating Quil code, as well as some features that benefit from that availability but which do not directly participate in the compilation pipeline (e.g., generation and manipulation of Clifford group elements).

A.2. Compiler subroutine DSL

Quilc implements a domain-specific language (DSL) for writing compilation subroutines. This makes it easy to specify algebraic relationships which Quilc can use as a part of its automatic process of program decomposition and optimization.

The basic method of definition is `define-compiler`, whose basic syntax is given by the following `defun` mimic:

```
(define-compiler name ([binding-with-options ...]
                       [global-option ...])
  [body-form ...])
```

This defines a compiler which consumes an instruction argument for each binding, evaluates the body forms in order, and returns the collection of instructions that they aggregate. Each binding is specified by a variable in which the input gate is stored, as well as an optional destructuring pattern to capture its operator name, parameter list, and argument list. This can be further manipulated by specifying options, which might install a further matching predicate on the destructured information. Altogether, these take the following form:

```
binding-with-options := (name (operator parameter-list argument-list)
                         [local-option ...])
                      | (name [local-option ...])
                      | name

parameter-list := ([parameter ...])
argument-list  := [argument ...]
local-option   := :where PREDICATE | ...

operator  := SYMBOL | WILDCARD | STRING
parameter := SYMBOL | DECIMAL-LITERAL
argument  := SYMBOL | INDEX-LITERAL
```

The forms describing the body of the compiler are largely identical to forms elsewhere in Lisp, but there are a few special-use forms available as well. The `inst` and `inst*` operators send an instruction to the output queue, which gets emptied for use as the return value at the conclusion of the compiler body. Alternatively, `finish-compiler` and `give-up-compilation` can be used to manually signal the end of the compiler body, additionally providing optional manual control over what is used for the return value.

```
body-form      := (inst INSTRUCTION)
                 | (inst operator parameter-list argument-list)
                 | (inst operator matrix argument-list)
                 | (inst* operator parameter-list argument-list
                                                  (argument-list))
                 | (inst* operator matrix argument-list
                                          (argument-list))
                 | (finish-compiler [return-form])
                 | (give-up-compilation)
                 | FORM
```

We now demonstrate the construction of compilation routines of increasing levels of complexity. We remark that these examples are taken to be didactic; Quilc houses roughly a hundred distinct compilation routines, ranging from algebraic rewriting rules specialized on gates which are common targets for implementation on current and near term hardware (e.g. RZ, CNOT, CZ) to general purpose recursive routines applicable to arbitrary unitary operations.

Example 1. Linearity of certain gates allows the compiler to collapse sequences of applications of a single gate. The optimizer `agglutinate-RZs` below rests on the linearity property

$$\mathtt{RZ}(\theta) \cdot \mathtt{RZ}(\phi) = \mathtt{RZ}(\theta + \phi).$$

It matches against two RZ instructions (bound to the variables `x` and `y`) acting on a particular common qubit `q`, and it binds their parameter values to the variables `theta` and `phi` respectively. The compiler then emits a single instruction RZ $(\theta + \phi)$ that replaces the two input instructions.

```
(define-compiler agglutinate-RZs
    ((x ("RZ" (theta) q))
     (y ("RZ" (phi)   q)))
  (inst "RZ" `(,(param-+ theta phi)) q))
```

We note that on Rigetti's current hardware, the RZ gate is the primary target for parametric compilation, and the above routine is capable of acting both on numeric and symbolic values of θ and ϕ in a manner in which numeric routines (e.g. depending on the underlying gate matrix) would not be.

Example 2. Parametric gates for some parameter values will be equivalent to the identity operation (NOP). This is easily seen for those gates that impart a rotation upon a qubit's state about some axis: a rotation of 2π is equivalent to not having rotated at all.[9] The `eliminate-full-CPHASE` optimizer implements this reduction for the instruction CPHASE. The compiler matches against a CPHASE instruction, binds the parameter value to `theta`, and via the `: where` guard restricts input instructions to those whose parameter values are integer multiples of 2π. The optimizer emits no instructions, thereby reducing the instruction stream by one.

```
(define-compiler eliminate-full-CPHASE
    ((x ("CPHASE" (theta) _ _)
        :where (and (typep theta 'double-float)
                    (double= 0d0 (mod theta (* 2 pi))))))
  nil)
```

Example 3. If the compiler can find no compilation routines that specifically match against a given input instruction, then it will look for less specific routines. The nativizer `euler-zyz-compiler` below only requires that it match against a single-qubit instruction, without specifying the name of the instruction or its parameters.[10] Since any single-qubit instruction is equivalent to a rotation about some axis, it can be

[9] The compiler here takes the idealist's view: that rotations are perfect and without noise. If we were to consider noise, then it might not be the case that RX (2π) is the identity.

[10] Note that in this routine, explicit numeric values are expected for the gate entries in order to compute the CSD composition. With respect to parametric compilation, such routines are inapplicable at sites where symbolic parameters are present.

decomposed into three rotations about the Y and Z axes. This compilation routine is unlike the previous examples in that it emits more instructions than it consumes.[11]

```
(define-compiler euler-zyz-compiler ((instr (_ _ q)))
  ;; Cosine-sine decomposition (CSD) of a unitary matrix U is
  ;; described as U = (u0 ⊕ u1)D(v0 ⊕ v1) with
  ;;   * u0, u1, v0, v1 unitary;
  ;;   * D00 = D11 = diag{cos θ : θ ∈ angles}; and
  ;;   * D10 = −D01 = diag{sin θ : θ ∈ angles}.
  (multiple-value-bind (u0 u1 v0 v1 angles)
      (magicl:lapack-csd (gate-matrix instr) 1 1)

(let ((alpha (- (phase (magicl:ref v1 0 0))     ; (magicl:ref M i j) = M_{i,j}
                (phase (magicl:ref v0 0 0))))
      (beta (first angles))
      (gamma (- (phase (magicl:ref u1 0 0))
                (phase (magicl:ref u0 0 0)))))
 (inst "RZ" (list alpha) q)
 (inst "RY" (list beta)  q)
 (inst "RZ" (list gamma) q)))))
```

Example 4. Given the preceding routines, we give an example trace of the compiler for a simple Quil program. The ordering of steps below is *automatically* determined by Quilc, and changes with target architecture and other factors. In particular, with respect to the taxonomy of section 3, `agglutinate-RZs` and `eliminate-full-CPHASE` are *optimizing* compilation routines, whereas `euler-zyz-compiler` is a *nativizing* routine.

In this sample trace, we consider a hypothetical two-qubit device which supports `RZ`, `RY`, and `CPHASE` natively, but for which `H` is not a native gate. We start with the following program:

```
RZ(-pi)       0
CPHASE(2*pi)  1 0
H             0
```

During the addressing phase, the non-native Hadamard gate will be translated to native gates by means of `euler-zyz-compiler`, yielding:

```
RZ(-pi)       0
CPHASE(2*pi)  1 0
RZ(pi)        0
RY(pi/2)      0
RZ(0)         0
```

At this point, the 'logical' qubits 0 and 1 have been assigned to 'physical' qubits 0 and 1, without the need for any `SWAP` operations. During the compression stage, Quilc scans the program and will first apply `eliminate-full-CPHASE` to the spurious gate, yielding:

```
RZ(-pi)   0
RZ(pi)    0
RY(pi/2)  0
RZ(0)     0
```

Finally, it applies `agglutinate-RZs` to the first pair of Z-rotations to produce:

```
RZ(0)     0
RY(pi/2)  0
RZ(0)     0
```

If we were to additionally tell Quilc that $RZ(0)$ is the identity:

```
(define-compiler ((instr ("RZ" (0d0) _)))
  nil)
```

[11] This subroutine makes use of `magicl`, a Common Lisp library for numerical linear algebra.

then Quilc will strike the outer instructions, leaving just:

```
RY(pi/2) 0
```

One can explore Quilc's treatment of an input program with the `--verbose` option at the command line.

A.3. Other features of quilc

We have described the backbone of Quilc's operation but have not described exhaustively all of Quilc's features. We mention a few additional salient features here.

Details of state-aware compilation. Quilc includes a minimalistic Quil interpreter which is optionally used for various optimizations. If the user indicates that the state of the quantum system is initialized to $|0\rangle$, as is defined by Quil, then Quilc will attempt to partially simulate the program up to a certain entanglement limit. The partially simulated state is then supplied to compilers defined with `define-compiler`, where it can be optionally used. For instance, we can detect if the state is an eigenvector of an instruction and consequently eliminate it:

```
(define-compiler elide-applications-on-eigenvectors
    ((instr :acting-on (psi qubit-indices)
            ;; (collinearp u v) returns true iff v = e^{iθ}u for some θ.
            :where (collinearp
                    psi
                    (nondestructively-apply-instr-to-wf instr psi qubit-indices))))
  nil)
```

More generally, if U represents our original partial program and $|\psi\rangle = U|0\rangle$ represents the partially simulated state, then Quilc can find an alternative V such that $|\psi\rangle = V|0\rangle$ which compiles to fewer instructions than U does. These features are enabled at the command line with `--enable-state-prep-reductions`.

Permutation matrices and contributed modules. Quilc does not have native support for compiling certain special kinds of unitary matrices, such as permutation matrices or diagonal matrices. This, in turn, makes Quilc particularly poorly suited for compilation of classical reversible logic. However, due to its flexible mechanism for extension and automatic selection of compilers, and due to easy integration of libraries written in C or C++, Quilc makes use of a library called Tweedledum [8]. Tweedledum has specialized routines for synthesizing such circuits and can be employed automatically by Quilc.

Approximate compilation. Quilc is able to profitably make use of gate fidelities to produce programs which have better overall fidelity. The fundamental observation is as follows. If a program $U = U_m, \ldots, U_1$ is expressed as m native gates, then the program as executed on a quantum computer will be some other $U' = U'_m, \ldots, U'_1$, where the difference or uncertainty of U_i and U'_i is reflected in the native gates' fidelity. We might deliberately compile U as some *non-equivalent* sequence of native gates $V = V_n, \ldots, V_2 V_1$ such that, when run on a quantum computer with initial state $|\psi_0\rangle$ and realized as $V' = V'_n, \ldots, V'_2 V'_1$, the average fidelity is increased:

$$\int |\langle\psi|U^\dagger U'|\psi\rangle|^2 \, d\psi < \int |\langle\psi|U^\dagger V'|\psi\rangle|^2 \, d\psi.$$

The existence of such alternative circuits is discussed in [6, 15].

Parametric compilation. Quilc is capable of dealing with symbolic parameters in a variety of cases beyond the modest use in section 5. As a small example, Quilc reduces the following snippet

```
DECLARE a REAL
RZ(a) 0
RZ(0.5*a) 0
RZ(0.2) 0
```

to the single instruction `RZ(0.2+1.5*a) 0`. As a more complicated example, the instruction `CPHASE(t/3) 0 1` compiles to

```
RZ(-pi/2) 1
RX(pi/2) 1
CZ 1 0
RX(-pi/2) 1
RZ(-t/6) 1
RX(pi/2) 1
CZ 1 0
RZ(t/6) 0
RX(-pi/2) 1
RZ(pi/2 + t/6) 1
```

on a CZ-based architecture.

Hamiltonians consisting of Pauli sums. Sums of Pauli operators occur frequently, especially in Hamiltonians used to describe qubit–qubit interactions. For instance, for qubits a and b, the 'XY' interaction is defined by the Hamiltonian

$$H_{XY} := X_a \otimes X_b + Y_a \otimes Y_b,$$

or simply $XX + YY$ when context is clear. The interaction itself is described by the unitary operation

$$U_{XY}(\theta) := \exp(i\theta H_{\mathrm{XY}}).$$

Quil is able to specify such unitary operators with the DEFGATE (see figure 2) construct:

```
DEFGATE Uxy(%theta) a b AS PAULI-SUM:
    XX(%theta) a b
    YY(%theta) a b
```

Quilc is able to compile operators defined in such a way, even in some cases for undetermined parameters via parametric compilation of the last section. For instance, consider the program:

```
DECLARE t REAL
Uxy(t) 0 1
```

Providing this Quilc will emit the following 12-gate program

```
RZ(-pi/2)    0
RX(pi)       0
RX(pi/2)     1
CZ           0 1
RZ(pi)       0
RX(pi)       0
RX(pi/2)     1
RZ(pi - 2*t) 1
RX(pi/2)     1
CZ           0 1
RZ(-pi/2)    0
RX(-pi/2)    1
```

for a CZ-based architecture.

Clifford manipulation. Quilc's source code includes a comprehensive library for manipulating the Clifford group, including subroutines suitable for randomized benchmarking and stabilizer simulation of Clifford gates of arbitrary dimension. Implementation details of the routines included in Quilc for randomized benchmarking can be found in [23].

Appendix B. A chronology of Quilc

Quilc originated as a 'compilation framework' for Quil in the summer of 2016. Much of the early work was on the the 'front-end' scaffolding common to most classical compilers (e.g. control-flow graphs, resource parallelization, and nano-pass concepts [17]).

Core work on the modern architecture of this paper began in the summer of 2017 with an implementation of the recursive cosine–sine decomposition of [22]. By fall of 2017, the broad structure of the compiler, including the division into separate addressing and compression stages, had been worked out. At this point, duration-based heuristics (with look-ahead) were adopted as the default for `SWAP` selection, and a number of additional compilation routines (including the optimal 2Q implementation of [20] as well as the recursive quantum Shannon decomposition of [18]) had been implemented.

In spring of 2018, both parametric compilation and state-aware compilation were introduced. Over the following summer, a number of enhancements to the addressing stage were implemented, including the introduction of additional heuristics (such as A* search for `SWAP` selection) as well as the adoption of partial logical-to-physical qubit mappings.

On January 30, 2019 Quilc was released as an open source project [4]. A number of contributions have been made since then. Of relevance to this paper, we mention that work on fidelity-based addressing heuristics continued into fall of 2019, and the compiler front-end was extended to support QASM [7] programs in December of that year.

ORCID iDs

R S Smith https://orcid.org/0000-0002-6340-9589
E C Peterson https://orcid.org/0000-0002-1633-0050
M G Skilbeck https://orcid.org/0000-0002-0558-4231
E J Davis https://orcid.org/0000-0001-7696-2538

References

[1] Abelson H, Jay Sussman G and Sussman J 1996 *Structure and Interpretation of Computer Programs* 2nd edn (Cambridge: MIT Press)
[2] Rigetti Computing 2019 *Quantum Virtual Machine* https://github.com/rigetti/qvm
[3] Rigetti Computing 2019 *Quil: A Portable Quantum Instruction Language* https://github.com/rigetti/quil
[4] Rigetti Computing 2019 *Quilc: The Rigetti Optimizing Quil Compiler* https://github.com/rigetti/quilc
[5] Alexander C, Dilkes S, Duncan R, Alexandre K, Simmons W and Sivarajah S 2019 On the qubit routing problem (arXiv:1902.08091)
[6] Cross A W, Bishop L S, Sheldon S, Nation P D and Gambetta J M 2019 Validating quantum computers using randomized model circuits *Phys. Rev.* A **100** 032328
[7] Cross A W, Bishop L S, Smolin J A and Gambetta J M 2017 Open quantum assembly language arXiv:1707.03429)
[8] EPFL Logic Synthesis Group 2018 *Tweedledum* https://github.com/boschmitt/tweedledum
[9] Iten R, Sutter D and Woerner S 2019 Efficient template matching in quantum circuits (arXiv:1909.05270)
[10] Peter J K, Tezak N A, Peterson E C, Ryan C A, da Silva M P and Smith R S 2020 A quantum-classical cloud platform optimized for variational hybrid algorithms *Quantum Sci. Technol.* **5** 024003
[11] McArdle S, Endo S, Aspuru-Guzik A, Benjamin S and Yuan X 2018 Quantum computational chemistry **92** 015003
[12] Prakash M, Baker J M, Javadi-Abhari A, Chong F T and Martonosi M 2019 Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers *Proc. of the 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* pp 1015–29
[13] Peter N 1992 *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* 1st edn (San Francisco, CA: Morgan Kaufmann)
[14] Alberto P, McClean J, Shadbolt P, Man-Hong Y, Zhou X-Q, Love P J, Aspuru-Guzik A and O'Brien J L 2014 A variational eigenvalue solver on a photonic quantum processor *Nat. Commun.* **5** 4213
[15] Peterson E C, Crooks G E and Smith R S 2019 Fixed-depth two-qubit circuits and the monodromy polytope (arXiv:1904.10541)
[16] Plesch M and Brukner Č 2011 Quantum-state preparation with universal gate decompositions *Phys. Rev.* A **83** 032302
[17] Sarkar D, Waddell O and Kent Dybvig R 2005 Educational pearl: a nanopass framework for compiler education *J. Funct. Program.* **15** 653–67
[18] Shende V V, Bullock S S and Markov I L 2006 Synthesis of quantum logic circuits *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **25** 1000–10
[19] Shende V V and Markov I L 2008 On the cnot-cost of toffoli gates (arXiv:0803.2316)
[20] Shende V V, Markov I L and Bullock S S 2004 Minimal universal two-qubit CNOT-based circuits *Phys. Rev.* A **69** 062321
[21] Siraichi M Y, Fernandes dos Santos V, Collange S and Quintão Pereira F M 2018 Qubit allocation *Proc. of the 2018 Int. Sym. on Code Generation and Optimization* pp 113–25
[22] Alexander S 2006 *Quantum Gate Decomposition Algorithms. Technical report, Sandia National Laboratories*
[23] Smith R S 2018 The computational structure of the Clifford groups *Proc. of the 11th European Lisp Symp.* ed D Cooper (New York: ACM)
[24] Smith R S, Curtis M J and Zeng W J 2016 *A Practical Quantum Instruction Set Architecture*
[25] Spanier E H 1966 *Algebraic Topology* (New York: McGraw-Hill)
[26] Vatan F and Williams C 2004 Optimal quantum circuits for general two-qubit gates *Phys. Rev.* A **69** 032315
[27] Venturelli D, Do M, Rieffel E and Frank J 2018 Compiling quantum circuits to realistic hardware architectures using temporal planners *Quantum Sci. Technol.* **3** 025004
[28] Zulehner A, Paler A and Wille R 2019 An efficient methodology for mapping quantum circuits to the IBM QX architectures *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **38** 1226–36