Qubit Allocation

Marcos Yukio Siraichi, Vinicius Fernandes Dos Santos, Caroline Collange, Fernando Magno Quintão Pereira

Qubit Allocation

Marcos Siraichi, Vinicius Fernandes Dos Santos, Caroline Collange, Fernando Magno Quintão Pereira

▶ To cite this version:

Marcos Siraichi, Vinicius Fernandes Dos Santos, Caroline Collange, Fernando Magno Quintão Pereira. Qubit Allocation. CGO 2018 - International Symposium on Code Generation and Optimization, Feb 2018, Vienna, Austria. pp.1-12, 10.1145/3168822 . hal-01655951v2

HAL Id: hal-01655951

https://hal.archives-ouvertes.fr/hal-01655951v2

Submitted on 7 Oct 2020

Qubit Allocation

Marcos Yukio Siraichi
Universidade Federal de Minas Gerais
Brazil
yukio.siraichi@dcc.ufmg.br

Vinícius Fernandes dos Santos
Universidade Federal de Minas Gerais
Brazil
viniciussantos@dcc.ufmg.br

Caroline Collange
Inria, Univ Rennes, CNRS, IRISA
France
caroline.collange@inria.fr

Fernando Magno Quintão Pereira
Universidade Federal de Minas Gerais
Brazil
fernando@dcc.ufmg.br

Abstract

In May of 2016, IBM Research has made a quantum processor available in the cloud to the general public. The possibility of programming an actual quantum device has elicited much enthusiasm. Yet, quantum programming still lacks the compiler support that modern programming languages enjoy today. To use universal quantum computers like IBM's, programmers must design low-level circuits. In particular, they must map logical qubits into physical qubits that need to obey connectivity constraints. This task resembles the early days of programming, in which software was built in machine languages. In this paper, we formally introduce the qubit allocation problem and provide an exact solution to it. This optimal algorithm deals with the simple quantum machinery available today; however, it cannot scale up to the more complex architectures scheduled to appear. Thus, we also provide a heuristic solution to qubit allocation, which is faster than the current solutions already implemented to deal with this problem.

CCS Concepts • **Computer systems organization** → **Quantum computing**; • **Hardware** → **Quantum technologies**; • **Theory of computation** → Algorithm design techniques;

ACM Reference Format:
Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2018. Qubit Allocation. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3168822

1 Introduction

The recent introduction of cloud access to quantum computer prototypes has made experimental quantum computing (QC) available to a wide community [11]. For instance, the IBM Quantum Experience program[1] lets users build experiments based on either a visual circuit representation or a gate-level language based on the Quantum Assembler (QASM) syntax [9, 40]. However, the level of abstraction offered by quantum circuits is low, and circuits need to obey machine-specific constraints [19]. Today's quantum computer prototypes have tight resource constraints. For instance, the IBM qx2 computer supports 5 qubits, connected by a partial network. A 16-qubit computer qx3 is also under beta testing by IBM, while 20-qubit and 50-qubit versions have been announced [16]; however, the connectivity between qubits of these computers remains very restrictive. Consequently, manual mapping and tuning of QC algorithms is difficult.

In addition, decoherence and noise effects severely constrain the execution time. Unlike classical digital gates that are inherently self-stabilizing, quantum gates accumulate noise. Although quantum error-correcting codes (QEC) hold the promise to address decoherence issues [25], current hardware do not provide nearly enough resources to implement realistic QEC [9, 30]. The longer a quantum program runs and the more operations it performs, the more it is susceptible to noise. Therefore, minimizing runtime and complexity is crucial, as it does not just affect the time-to-solution, but also the accuracy of the solution itself. For these reasons, compilation of quantum circuits demands extremely accurate compiler optimization.

Quantum circuits manipulate *qubits* – the quantum analogue of the classical bit. These qubits, which exist as abstractions within a quantum circuit, shall be called *pseudo* or *logical*. In this paper, we are interested in mapping *pseudo qubits* into *physical qubits*, which denote the actual hardware units that store quantum bits. This problem henceforth shall be called *qubit allocation*. Just like registers in a classical computer architecture, quantum computers have a limited number of qubits. Furthermore, these units are not always

[1] http://research.ibm.com/ibm-q/

fully connected, meaning that not every subset of physical qubits can participate as inputs and outputs to the same quantum gates. As we explain in Section 2, solving qubit allocation involves dealing with hard combinatorial problems.

In this paper, we formally describe the qubit allocation problem in Section 3, and introduce an exact solution to solve it in Section 4.1. The exact solution is exponential. Although it works well for the small quantum systems available today, it cannot scale up to the more complex architectures that are likely to emerge in the future. Nevertheless, it sets a mark against which we can test different heuristics. To support this statement, we show how state-of-the-art implementations of qubit allocators fare against this exact baseline. This comparison has motivated us to go beyond these implementations; a task that we accomplish with a novel allocator of our own craft, which we introduce in Section 4.2.

Section 5 provides a thorough evaluation of the different algorithms that exist today to perform qubit allocation. Not many classes of quantum algorithms are known; and even fewer accommodate the constraints of early quantum computers [31]. Thus, we have assembled a small collection of microbenchmarks that are part of known quantum algorithms, and have implemented a generator of random programs. Together, the actual and synthetic benchmarks give us a number of samples that is comprehensive enough to test our ideas, and demonstrate their effectiveness.

2 Overview

This section introduces the qubit allocation problem. Qubit allocation involves modifying quantum circuits with specific combinations of quantum gates, which we call *transforms*. Although familiarity with qubits and quantum gates might be helpful to understand the problem, we shall try to keep our discussion on a level that suits the reader unversed with quantum computing. For a more thorough discussion, we refer the interested reader to [31].

Qubits and Quantum Gates. Quantum programs are made of qubits and reversible quantum gates, which receive qubits as inputs, and produce qubits as outputs. Figure 1 shows a quantum circuit, which implements two boolean functions. This circuit has four pseudo qubits: a_0, a_1, b_0 and b_1, which are represented as horizontal lines. It uses four different types of gates to operate on these qubits: H, T, $T^\dagger$ and CNOT, where CNOT_{ab} is depicted with a dot on qubit a and $\oplus$ on qubit b. Gates change the *state* of qubits. The state of a single qubit is represented as a two dimensional complex vector:

$$\alpha|0\rangle + \beta|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}$$

In this case, $|0\rangle$ and $|1\rangle$ are the basis states of a 2D complex vector space, and α and β are complex numbers. Under this terminology, quantum gates can be understood as unitary matrix operations applied on vectors that describe quantum states. Example 2.1 illustrates this view.

Example 2.1. The *Hadamard-Walsh* gate H maps the basis state $|0\rangle$ to $(|0\rangle + |1\rangle)/\sqrt{2}$, and $|1\rangle$ to $(|0\rangle - |1\rangle)/\sqrt{2}$. Thus, it is equivalent to multiplying the quantum state by the matrix:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$$

Like the H and other single-qubit gates, the T gate is represented as a 2×2 matrix that multiplies a quantum state. Its adjoint $T^\dagger$ is its inverse, so that $TT^\dagger$ is the identity matrix. The CNOT (short for *Controlled Not*) gate applies on two qubits. CNOT_{ab} indicates that *a controls b*. Informally, it negates b, the second qubit, when a, the first qubit, is $|1\rangle$. When a is $|0\rangle$, the gate leaves b unchanged.

Architectural Constraints. The exact semantics of quantum gates will be immaterial for the rest of this paper. The only important aspect of these gates in our work is their *arity*: how many qubits they read, and how many they write. Indeed, it has been shown that all single-qubit gates and the CNOT gate form a universal set of gates that can implement arbitrary circuits [2]. Even though single-qubit gates may affect the resulting circuit through optimizations, we shall focus only on the CNOT gates in this paper.

The placement of CNOT gates matters due to *architectural constraints*. Actual quantum computers might not allow CNOTs to be performed between arbitrary pairs of qubits. In particular, quantum computers based on superconducting qubit technology are made of solid-state circuits that only allow local interactions between qubits that are physically connected [12, 23]. Technological reasons restrict the number of possible couplings and their organization [14]. As an example, Figure 2 (a) shows the *coupling graph* of the IBM qx2 computer [11]. The coupling graph determines which qubits can communicate, typically through CNOT gates. We define the coupling graph in terms of CNOT gates as follows:

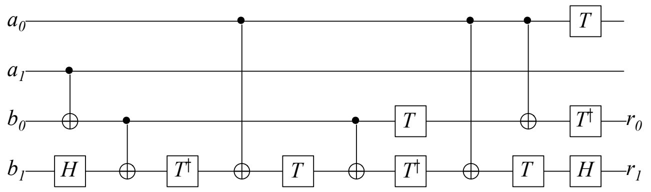


Figure 1. This quantum circuit implements the boolean functions $b_0 = a_0 \oplus b_0 \oplus a_1$, and $b_1 = (a_0 \wedge b_0) \oplus a_1 \oplus b_0 \oplus b_1$. $\text{CNOT}_{a_1 b_0}$ represents $a_1 \oplus b_0$, and the rest represents a Toffoli gate which is equivalent to $(a_0 \wedge b_0) \oplus b_1$. There should be a final $\text{CNOT}_{a_0 b_0}$ to complete the Toffoli gate, however as there is a $\text{CNOT}_{a_0 b_0}$ right after this gate, they cancel each other.

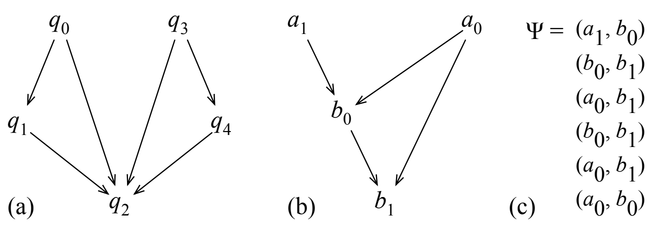


Figure 2. (a) The coupling graph of the IBM qx2 computer. (b) Interactions between qubits of the circuit seen in Figure 1. (c) Dependences that have created these interactions.

Definition 2.2 (Coupling Graph)**.** Given a quantum architecture A with a set Q of qubits, its coupling graph is a directed graph $G_q = (Q, E_q), E_q \subseteq Q \times Q$. The edge $q_1 \to q_2 \in E_q$ if, and only if, $\text{CNOT}_{q_1 q_2}$ is valid in A.

Qubit Allocation – An Informal Overview. The connectivity relations in a quantum circuit need to be mapped to the coupling graph. For instance, in Figure 1, we have that the pseudo qubit a_0 controls pseudos b_0 and b_1. When allocating pseudo qubits onto the coupling graph, we would like to enable such control relations. However, *perfect mappings* that enable all the control relations in a quantum circuit are not always possible, as Example 2.3 illustrates.

Example 2.3. It is not possible to map the control circuit of Figure 1 directly onto the coupling graph of Figure 2 (a). The graph in Figure 2 (b) represents the control relations in that circuit. This graph contains two nodes of in-degree two, which have no equivalent in Figure 2 (a).

In its simplest version, the qubit allocation problem receives an ordered list of pairs, describing control relations in the quantum circuit, plus a coupling graph. This problem, which Definition 2.4 states, asks for a mapping between pseudos and physical qubits that respects the control relations. Because Definition 2.4 does not ask for ways to adapt a circuit to fit into a coupling graph, we call this version of qubit allocation the *Assignment Problem.*

Definition 2.4 (The Qubit Assignment Problem)**. Input:** a coupling graph $G_q = (Q, E_q)$, plus a list $\Psi = (P \times P)^n, n \geq 1$ of n control relations between pseudo qubits. **Output:** yes, if there is a mapping between pseudo and physical qubits that respects the control relations in Ψ.

Perfect mappings might not exist, as seen in Example 2.3. In other words, the instance of qubit allocation in Figure 2 (a & c) does not have a positive answer. In this case, we must resort to *circuit transformations* to solve qubit allocation. This is a notion that we discuss in the rest of this section.

Circuit Transformations. A transformation is a combination of gates that we can insert into a quantum circuit to emulate the semantics of non-existing CNOT relations or switch the state of physical qubits. We call the first category of transformations *virtual CNOTs*, and the latter *state changes*. Example 2.5 describes some of these transformations.

Example 2.5. Below we list three kinds of transformations:

Reversal: Emulation of a virtual CNOT between p_a and p_b controlled by p_a using a CNOT from p_b to p_a (controlled by p_b) and 2 extra levels of Hadamard gates, as shown in Figure 3 (a).
Bridge: Emulation of a virtual CNOT between p_a and p_c controlled by p_a using two CNOTs from p_a to p_b (controlled by p_a), plus two CNOTs from p_b to p_c (controlled by p_b), as shown in Figure 3 (b).
Swap: exchanges two pseudo qubits p_a and p_b, as shown in Figure 3 (c), at the expense of three CNOT and two levels of Hadamard gates.

As Figure 3 shows, a CNOT reversal allows the mapping of "backward" edges on the coupling graph, at the cost of extra gates. A bridge uses four CNOTs to implement a virtual gate at distance 2 in the coupling graph. Finally, a CNOT swap allows the migration of pseudo qubits across physical qubits. Whereas reversals and bridges are gate transformations, swaps transform states. That is to say: a reversal inverts the meaning of a CNOT gate, and a swap exchanges the position of two pseudo qubits. These transformations can be combined to map a quantum circuit onto a given architecture. Example 2.6 shows that.

Example 2.6. Figure 4 outlines a solution to qubit allocation for the program in Figure 1 using two CNOT reversals. Reversals add further complexity to the target circuit; however, some gates can be simplified away, given well-known quantum identities [27].

A particular instance of qubit allocation might have several different solutions. As an example, Figure 5 shows an allocation for our running example, this time using one CNOT swap, instead of two reversals. The quality of a solution is given by its *cost*, which we measure as the number of gates necessary to implement it. In Section 3, we show that finding

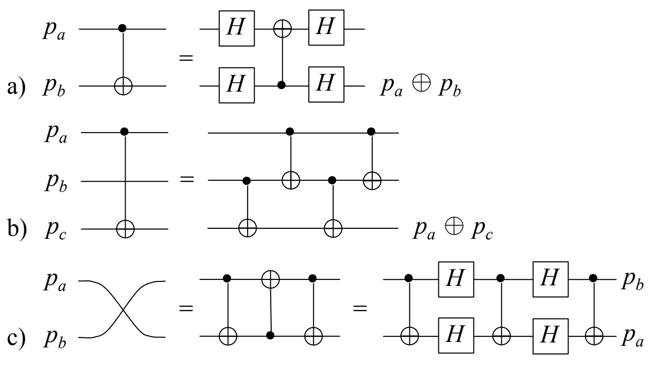


Figure 3. (a) Reversal. (b) Bridge. (c) Swap.

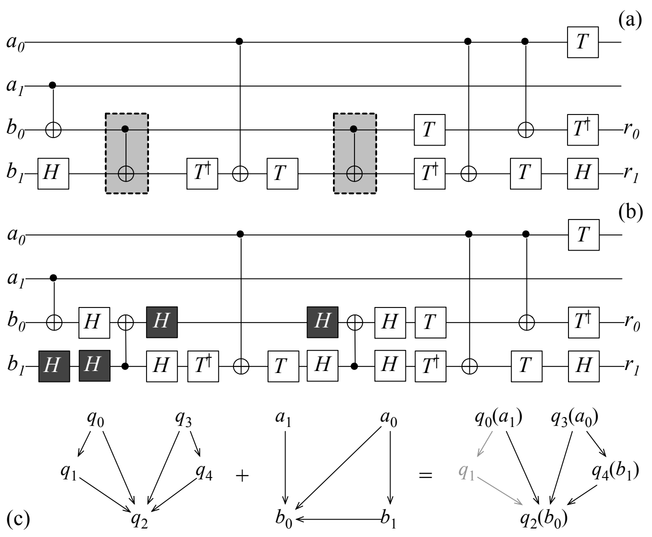


Figure 4. (a) CNOT reversals, marked as grey boxes, invert the direction of CNOT$_{b_0 b_1}$. (b) The four black Hadamard gates can be simplified away, given the identity $HH = I$. (c) Solution to qubit allocation showing embedding of the control graph onto the coupling graph.

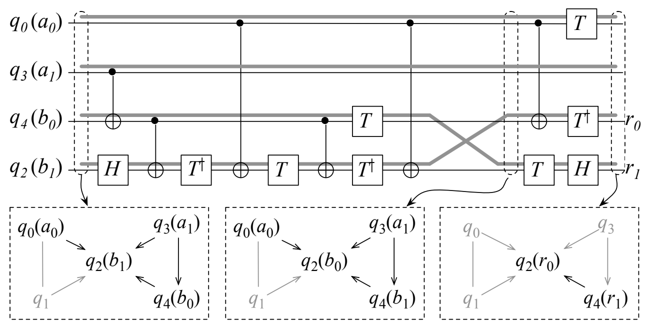


Figure 5. Solution of qubit allocation for the circuit in Figure 1, using a CNOT swap. Grey lines represent physical qubits. We show the different mappings that we have at three points of the circuit.

the best solution for several variations of qubit allocation involves solving NP-complete problems.

3 Problem Definition

Definition 2.4 states the Qubit Allocation Problem in its most basic form: given a quantum circuit and an architecture, we want to know if it is possible to map the pseudo qubits in the former to the physical qubits in the latter. Making an analogy with classic register allocation, the problem in Definition 2.4 is equivalent to knowing if we can map program variables (pseudo registers) onto the physical registers available in the target architecture. Like classic register allocation[2], Qubit Assignment is NP-complete, as Theorem 3.1 states.

Theorem 3.1. *Qubit Assignment (Def. 2.4) is NP-complete.*

Proof (Sketch): We make a reduction from subgraph isomorphism, which is known to be NP-hard [8]. First, note that finding isomorphisms between *directed* graphs is also NP-hard, since replacing every edge by two directed edges doesn't change the answer of any input. Given an instance of subgraph isomorphism, where we wish to find a subgraph of G that is isomorphich to H, we can map a graph G to the coupling graph and the edges of subgraph H to Ψ. Clearly, any solution of Qubit Allocation would find an embedding of H in G, which shows that Qubit Assignment is NP-hard. To complete the proof it is enough to notice that checking in any solution if all pairs of Ψ are properly mapped can be done in polynomial time. □

Theorem 3.1 sets our expectations about having an exact solution to solve Qubit Allocation. However, from a practical standpoint, Qubit Assignment is not very useful: most of the instances of Qubit Allocation will require quantum transformations to be effectively solved. Going back to our analogy with register allocation, most instances of register allocation lead to spilling; hence, forcing the insertion of load and store instructions in the program: program changes equivalent to our transformations. Thus, in the rest of this section we extend Definition 2.4 to encompass more pragmatic descriptions of the Qubit Allocation problem. We start with the subproblem that asks for the minimization of swaps.

Definition 3.2 (The Swap Minimization Problem)**. Input:** a coupling graph $G_q = (Q, E_q)$, a list $\Psi = (P \times P)^n, n \geq 1$ of n control relations between pseudo qubits, and an integer $K_s \geq 0$. **Output:** yes, if we can use up to K_s swaps to produce a version of Ψ that complies with G_q.

Swap Minimization is also NP-complete, because it involves solving a classic optimization problem know as the Token Swapping Problem [44]. Quoting Kawahara *et al.*, "For a given graph where each vertex has a unique token on it, token swapping requires to find a shortest way to modify a token placement into another by swapping tokens on adjacent vertices." Token Swapping has been shown to be NP-Hard [5, 22]. Swap Minimization is a special case of Qubit Allocation. In the most general problem, we can use quantum transformations other than swaps, and each one of them might have a different cost. We define this problem as follows:

Definition 3.3 (The Qubit Allocation Problem)**. Input:** a coupling graph $G_q = (Q, E_q)$, a list $\Psi = (P \times P)^n, n \geq 1$

[2]See hardness results for different aspects of the register allocation problem, due to Chaitin [6], Farach [13], Lee [24] and Pereira [34].

of n control relations between pseudo qubits, an integer $K_c \geq 0$, a list of allowed quantum transformations Θ, and a function $C : \Theta \mapsto \mathbb{N}$ that gives the *cost* to implement each transformation. **Output:** yes, if we can produce a version of Ψ that complies with G_q with transformations whose total cost does not exceed K_c.

Definition 3.3 subsumes the two simpler problems, stated in Definitions 2.4 and 3.2; therefore, it is unlikely that it can be solved exactly via a polynomial time algorithm. Definition 3.3 states the version of qubit allocation that we solve in Section 4 In Section 4.1, we provide an optimal – exponential time – solution to that problem; in Section 4.2, we provide a heuristic solution to it.

4 Solution

This section presents our solution to qubit allocation, as stated in Definition 3.3. For the reader's convenience, Figure 6 summarizes terms and notation adopted henceforth.

4.1 Exact Solution

We solve the Qubit Allocation problem, as given in Definition 3.3, using a dynamic programming algorithm. Our approach finds solutions gradually per index in the list of dependences Ψ. That is, given a collection of control dependences $\Psi = (p_1, p_2), (p_3, p_4), \ldots, (p_{2n-1}, p_{2n})$ between pseudo qubits that must be obeyed, we find the optimal cost of allocating qubits up to dependence i. This algorithm is based on a function $S(\ell, i)$, which we define below.

Definition 4.1 (Exact Solution)**.** Function $S(\ell, i) : L \times \mathbb{N} \mapsto \mathbb{N}$ is a solution to the qubit allocation problem if it gives the minimum cost of satisfying all the dependences in Ψ, up to index i, terminating with mapping $\ell \in L$.

[P] **Pseudo-qubits:** the set of qubits in Ψ.
[Q] **Physical Qubits:** the set of qubits present in the architecture. i.e. the vertices of the coupling graph.
[G_q] **Coupling Graph:** a directed graph $G_q = (Q, E_q)$.
[Θ] **Quantum Transformations:** in our implementation, we consider $\Theta = \{\theta_s, \theta_c, \theta_r, \theta_b\}$, representing swap, CNOTs, reversals and bridges, respectively.
[C] **Cost Function:** the cost of transformations $C : \Theta \to \mathbb{N}$.
[Ψ] **Input Control Relations:** the sequence $\Psi : (P \times P)^n$ of n control relations between pseudo qubits. These are the dependences that qubit allocation must satisfy.
[Γ] **Output:** a function $\Gamma : \mathbb{L} \times \mathbb{N} \to \Theta^*$ that gives the minimum-cost sequence of transformations necessary to satisfy the i-th CNOT relation, assuming an initial mapping $\ell \in L$ from pseudos to physicals.
[L] **Labeling:** We let L be the set of every mapping ℓ from P to Q, and L^{-1} be the set of every mapping ℓ^{-1} from Q to P.

Figure 6. Notation used in this paper.

We implement $S(\ell, i)$ in terms of three auxiliary functions, $\zeta : \Theta \times L \times \mathbb{N} \mapsto \mathbb{N}$, $\phi : L \times \mathbb{N} \mapsto \mathbb{N}$ and $\delta : L \times L \mapsto \mathbb{N}$. Function ζ, gives the cost of satisfying a dependence $\Psi(i)$ with a transformation θ, given a current mapping ℓ. Function ϕ yields the minimum cost to satisfy a given dependence. Finally, function δ gives the minimum cost of swaps necessary to transform a mapping ℓ_1 into another mapping ℓ_2. We define δ at the end of this section.

$$\begin{aligned} \zeta(\theta, \ell, i) = &\quad \text{if } (\ell, \theta) \models i \text{ then } C(\theta) \text{ else } \infty \\ \phi(\ell, i) = &\quad \min \zeta(\theta, \ell, i) \\ \delta(\ell_1, \ell_2) = &\quad \text{Transforms to convert } \ell_1 \text{ into } \ell_2 \end{aligned}$$

From ζ, ϕ and δ, we solve $S(\ell, i)$ as follows:

$$S(\ell, i) = \begin{cases} 0, \text{if } i = 0 \\ \infty, \text{if } \zeta(\theta, \ell, i) = \infty \\ \min\limits_{\forall \ell' \in L} S(\ell', i-1) + \delta(\ell', \ell) + \phi(\ell, i), \text{otherwise} \end{cases}$$

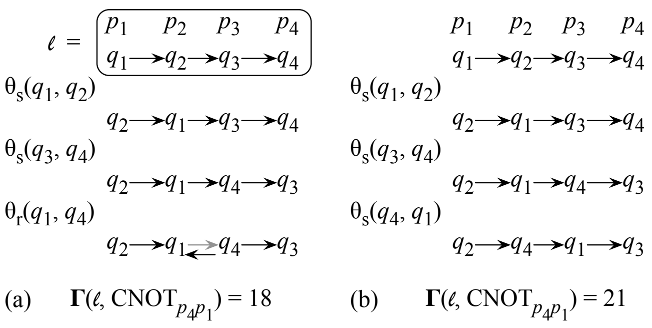
Theorem 4.2. *The problem of computing $S(\ell, i)$ has optimal substructure.*

Proof: To compute $S(\ell, i)$, we compute $(\ell', i - 1)$ independently, for each labeling ℓ'. The implication of this fact is that the recurrence relation that produces S is a Bellman Equation [3], a necessary enabler of a dynamic programming algorithm. ☐

Implementing** $S(\ell, i)$**: The function $S(\ell, i)$ gives the minimum cost to build a quantum circuit that satisfies i control relations created by the CNOT gates originally placed in the circuit. To generate code that represents $S(\ell, i)$, the compiler must insert transformations into the original quantum circuit to satisfy all the control relations. To keep track of these dependences, we define a function Γ, which describes all the transformations inserted between dependences. Thus, $\Gamma(\ell, i)$ is a list of transformations necessary to implement the i-th CNOT gate, given an initial mapping ℓ. Example 4.3 shows how Γ is used.

Example 4.3. Figure 7 shows different concretizations of Γ, assuming that $\Psi(i) = \text{CNOT}(p_4, p_1)$, the coupling graph is the path $q_1 \to q_2 \to q_3 \to q_4$, and the initial mapping is $L(p_1) = q_1, L(p_2) = q_2, L(p_3) = q_3$ and $L(p_4) = q_4$. Additionally, we assume that $C(\theta_c) = 0$, $C(\theta_r) = 4$, $C(\theta_s) = 7$ and $C(\theta_b) = 10$.

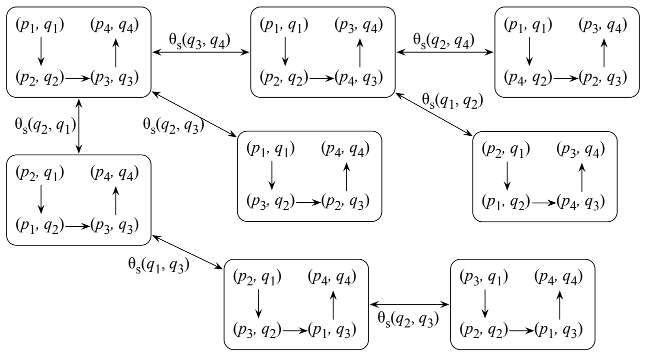
After computing $S(\ell, i)$, we build $\Gamma(\ell', i)$. The best cost of $S(\ell, i)$ uses $S(\ell', i - 1)$, according to the recurrence relation that defines S. This means that $\Gamma(\ell', i)$ equals the minimum sequence of swaps necessary to take ℓ' to ℓ, e.g., $\delta(\ell', \ell)$, plus – possibly – the cost of some state-preserving transformation such as θ_c, θ_r or θ_b, e.g., $\phi(\ell, i)$. In what follows, we discuss how we keep track of δ.



ℓ =
p_1 p_2 p_3 p_4
$q_1 \to q_2 \to q_3 \to q_4$
$\theta_s(q_1, q_2)$
$q_2 \to q_1 \to q_3 \to q_4$
$\theta_s(q_3, q_4)$
$q_2 \to q_1 \to q_4 \to q_3$
$\theta_r(q_1, q_4)$
$q_2 \to q_1 \leftarrow q_4 \to q_3$
(a) $\Gamma(\ell, \mathrm{CNOT}_{p_4 p_1}) = 18$
p_1 p_2 p_3 p_4
$q_1 \to q_2 \to q_3 \to q_4$
$\theta_s(q_1, q_2)$
$q_2 \to q_1 \to q_3 \to q_4$
$\theta_s(q_3, q_4)$
$q_2 \to q_1 \to q_4 \to q_3$
$\theta_s(q_4, q_1)$
$q_2 \to q_4 \to q_1 \to q_3$
(b) $\Gamma(\ell, \mathrm{CNOT}_{p_4 p_1}) = 21$

Figure 7. The Γ function reports the sequence of transformations of minimum cost necessary to satisfy a CNOT dependence, given an initial mapping L.

Memoizing the State Space Memoization is an optimization technique that stores the results of function calls and returns the cached result when the same inputs occur again. In our case, memoization is useful to avoid searching repeatedly for optimal sequences of transformations that change a given labeling ℓ onto another labeling ℓ'. We memoize all these paths in a table δ, already mentioned in the definition of $S(\ell, i)$. We compute δ by brute-force, performing a breadth-first search on the space of possible mappings between pseudo and physical qubits. Figure 8 illustrates this search for the coupling graph earlier seen in Example 4.3.



(p_1, q_1) (p_4, q_4) (p_2, q_2) (p_3, q_3)
$\theta_s(q_3, q_4)$
(p_1, q_1) (p_3, q_4) (p_2, q_2) (p_4, q_3)
$\theta_s(q_2, q_4)$
(p_1, q_1) (p_3, q_4) (p_4, q_2) (p_2, q_3)
$\theta_s(q_1, q_2)$
$\theta_s(q_2, q_1)$
$\theta_s(q_2, q_3)$
(p_1, q_1) (p_4, q_4) (p_3, q_2) (p_2, q_3)
(p_2, q_1) (p_3, q_4) (p_1, q_2) (p_4, q_3)
(p_2, q_1) (p_4, q_4) (p_1, q_2) (p_3, q_3)
$\theta_s(q_1, q_3)$
(p_2, q_1) (p_4, q_4) (p_3, q_2) (p_1, q_3)
$\theta_s(q_2, q_3)$
(p_3, q_1) (p_4, q_4) (p_2, q_2) (p_1, q_3)

Figure 8. Eight states reachable from the initial mapping discussed in Example 4.3. In total, we have sixteen states.

The exhaustive search of all the possible labeling gives us a graph $G_L = (L, E_L)$, whose vertices are elements $\ell \in L$. We have an edge from ℓ_1 to ℓ_2 if it is possible to convert ℓ_1 into ℓ_2 with one swap transformation. The minimum sequence of swaps necessary to map a given labeling ℓ onto another labeling ℓ' is given by the shortest path between ℓ and ℓ' in this graph. The function δ that produces the minimum sequence of swaps transforming one state into another emerges naturally from this graph. $\delta(\ell, \ell')$ is the shortest path between vertices ℓ and ℓ' in G_L. As an artifact of implementation, whenever we compute $\delta(\ell, \ell')$, for any pair of labelings, we save this result, to avoid further computations, in case the same pair of labelings need to be connected posteriorly.

On the Complexity of the Exact Solution. The preprocessing described in the last section enables us to calculate $\delta_{ll'}$ and $\Delta(l, l')$ for every $l, l' \in L$ by preprocessing the coupling graph only one time. The time complexity of this part of the algorithm is $O(|Q|! + |Q|! \cdot |E_q|)$, since we will apply a BFS in $|Q|!$ different permutations (labelings), each one with up to $|E_q|$ edges. Given the set of edges E_q, the space complexity is $O(|Q|! \cdot |E_q|)$, since we will visit $|Q|!$ vertices and for each vertex there are up to $|E_q|$ edges.

After preprocessing, there is the dynamic programming algorithm. As we can see, it iterates all possible mappings for all dependences. Since we know that: $O(|Q|!)$ is the number of possible mappings; $|\Psi|$ is the number of dependences; and δ takes linear time to execute, the time complexity of this algorithm is $O(|Q|!^2 \cdot |Q| \cdot |\Psi|)$ and its space complexity is $O(|Q|! \cdot |Q| \cdot |\Psi|)$. Finally, merging the preprocessing with the main algorithm, the time complexity becomes $O(|Q|!^2 \cdot |Q| \cdot |\Psi| + |Q|! + |Q|! \cdot |E_q|)$. Thus, $O(|Q|!^2 \cdot |Q| \cdot |\Psi|)$.

4.2 Heuristics

The algorithm of Section 4.1 provides an exact solution to qubit allocation; however, its exponential runtime renders its application impossible in large coupling graphs. To circumvent this problem, in this section we discuss a heuristic solution to qubit allocation. Later, in Section 5 we will show that this faster algorithm leads to results that are close to those found by the exponential time implementation. Our heuristic consists of two stages. The goal of the first stage is to find an initial mapping $\ell_0 \in L$ that attempts to maximize the number of satisfied control dependences. In the ensuing stage, we build a solution that satisfies all the dependence relations in the list of constraints Ψ, starting from ℓ_0.

4.2.1 Finding the Initial Mapping

Classic register allocation algorithms tend to keep in registers variables that are likely to be more used, such as those that appear in loops, or that appear in a larger number of instructions. Following this insight, in order to find an initial mapping ℓ_0 to some instance of the qubit allocation problem, we try to satisfy the dependences involving pseudo qubits that appear more times in the list of constraints Ψ.

Weighted Dependence Graph. From Ψ, we construct a weighted directed graph $G_p = (P, E_p, w_p, w_e)$, whose vertices are the pseudos that appear in Ψ. We have an edge $p_1 \longrightarrow p_2$ whenever $(p_1, p_2) \in \Psi$. The weight function $w_e : P \times P \mapsto \mathbb{N}$ counts the occurrences of dependences in Ψ. If $w_e(p_1, p_2) = n$, then the dependence (p_1, p_2) appears n times in Ψ. From w_e we define a function $w_p : P \mapsto \mathbb{N}$ as follows:

$$w_p(a) = \sum w_e(a, b), \forall (a, b) \in E_d$$

Given a dependence graph $G_p = (P, E_p, w_p, w_e)$:

1. we sort the list of pseudos P in descending order given by w_p, thus producing a list P^s of sorted pseudo qubits;
2. for each element $p \in P^s$ in order:
   a. we allocate p to a physical qubit q that has the nearest out-degree;
   b. for every $(p, p') \in \Psi$, if possible, we allocate p' to q', such that $(q, q') \in E_q$ and p' and q' have the closest out-degree;
   c. then, repeat for the children of p in the dependence graph.
3. if there are any unallocated pseudo qubits, we assign a free physical qubit to it.

Figure 9. Finding an initial mapping to qubit allocation.

From Weighted Graphs to ℓ_0. To find the initial allocation ℓ_0, we process $G_p = (P, E_p, w_p, w_e)$ according to the algorithm in Figure 9. We use the out-degree criterion as a tie-breaker as a stimulus to allocate pseudos to physicals that will be able to satisfy dependences. If pseudo p has out-degree k, then there exist k other qubits that must, ideally, be allocated to physicals adjacent to the qubit that receives p. We settle for the closest out-degree to maximize the change that other pseudo qubits can still benefit from the physical qubits of large degree still available in the coupling graph. Example 4.4 illustrates these issues.

Example 4.4. Figure 10 shows how we find the initial mapping for the circuit earlier seen in Figure 1. We shall allocate pseudos in the sequence a_0, b_0, a_1, b_1. The first pseudo, a_0, is mapped to q_0, as they have the same out-degree. In this case, the choice between q_0 and q_3 is arbitrary. From a_0, we allocate, recursively, b_0 and b_1, in a BFS-fashion.

4.2.2 Extending the Initial Mapping to handle Ψ

On the second stage of our heuristic, we extend ℓ_0, found in the previous step, so that it satisfies all the dependences in Ψ. The sequence of steps that we perform to achieve this end is enumerated in Figure 11. That algorithm traverses the list Ψ of dependences that must be satisfied. For each one of them, it might insert transformations in the quantum circuits, if the dependence is not already fulfilled by the current mapping from pseudos to physical qubits.

On How We Implement Swaps. A dependence $\Psi(i) = (p_0, p_1)$ cannot be satisfied by a mapping ℓ, if the edge $(\ell(p_0), \ell(p_1))$ is not present in the coupling graph G_q, i.e., $(\ell(p_0), \ell(p_1)) \notin E_q$. Under such circumstances, according to Figure 11, there are four possible actions that can follow. The first takes place when there are further dependences (p_0, p_1) in Ψ. In this case, we swap the state of qubits, so as to satisfy the first

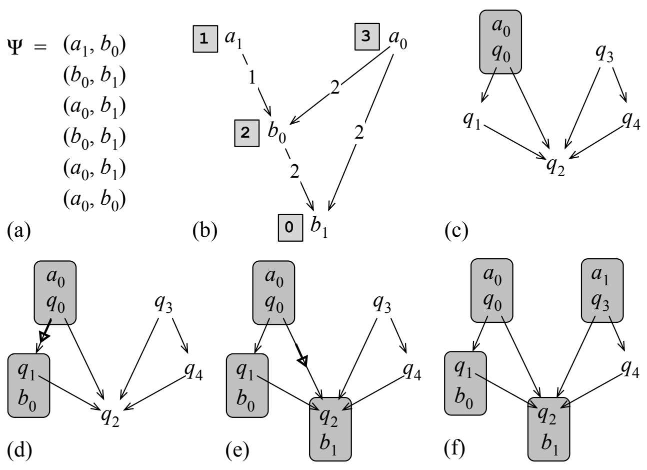


Figure 10. (a) List of control dependences from the quantum circuit seen in Figure 1. (b) Weighted dependence graph. Grey boxes represent w_p. (c-f) Step-by-step construction of the initial mapping.

Given a coupling graph $G_q = (Q, E_q)$, an initial mapping ℓ_0, and the dependences Ψ, for each i in the domain of Ψ, let $(p_0, p_1) = \Psi(i)$. If $(\ell_0(p_0), \ell_0(p_1)) \notin E_q$, then:

1. if (p_0, p_1) appears in Ψ two or more times, then we use a swap to move p_1 closer to p_0 in the coupling graph, update ℓ_0, and re-evaluate the four cases in this algorithm;
2. else if the edge $(\ell_0(p_1), \ell_0(p_0)) \in E_q$, then we use a reversal between $\ell_0(p_1)$ and $\ell_0(p_0)$;
3. else if $\exists q \in Q$, such that $(\ell_0(p_0), q) \in E_q$, and $(q, \ell_0(p_1)) \in E_q$, then we use a bridge between $(\ell_0(p_0), \ell_0(p_1)) \in E_q$;
4. else we create swaps, i.e., apply step (1) onto $(\ell_0(p_0), \ell_0(p_1))$.

Figure 11. Extending the initial mapping to satisfy Ψ.

occurrence of (p_0, p_1), and possibly others. Else, if Ψ contains only one dependence (p_0, p_1), then we use either a reversal or a bridge to create the missing CNOT gate in the coupling graph, if such is possible. Otherwise, we are in a situation in which there exists only one dependence $(p_0, p_1) \in \Psi$, and we cannot simulate the missing CNOT gate. If that is the case, then we resort to swaps, like in the first case.

To implement the dependence (p_0, p_1) with swaps, we try to move p_1 to some qubit q that is the successor of $\ell(p_0)$. When performing this movement, we choose always the shortest path from $\ell(p_1)$ to q. Sometimes, it is possible to avoid inserting a swap by changing ℓ_0, the initial mapping built in Section 4.2.1. This happens when this swap refers only to physical qubits that have not yet been visited by the loop in Figure 11. Example 4.5 clarifies this possibility.

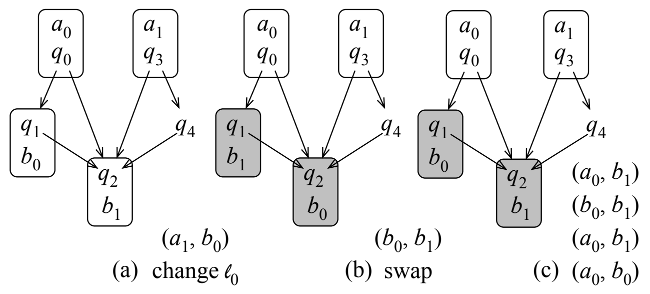


Figure 12. How the algorithm in Figure 11 works. (a) We change the original mapping to satisfy dependence $\Psi(1) = (a_1, b_0)$. No transformation is created during this action. (b) We swap b_1 and b_0, to satisfy dependence $\Psi(2) = (b_0, b_1)$. We used a swap because there are two occurrences of (b_0, b_1) in Ψ. (c) The other dependences are now satisfied.

Example 4.5. The first dependence in Fig. 10 that must be satisfied is (a_1, b_0). There is no edge $(\ell_0(a_1), \ell_0(b_0)) \in G_q$. To handle this dependence, the algorithm in Fig. 11 would swap q_1 and q_2. However, q_1 and q_2 have not been used as target or destination of any transformation thus far. Hence, we update ℓ_0, so that $\ell(b_0)$ becomes q_2, and $\ell(b_1)$ becomes q_1.

To support the optimization discussed in Example 4.5, we introduce the notion of *freezing*. A qubit is frozen the first time it is used in the loop of Figure 11. Frozen qubits cannot be modified in the original mapping. In contrast, qubits yet untouched are swapped "virtually" by changing their original allocation in ℓ_0. Figure 12 provides a final illustration on how our algorithm works when applied onto our original example, seen in Figure 1. In this case, our heuristic finds a solution to qubit allocation involving one swap. This result is similar to that seen in Figure 5, except that the swap appears earlier in the circuit.

Time Complexity. We find an initial mapping (Section 4.2.1) in $O(|Q| \cdot lg|Q| + |E_q| + |\Psi|)$, since we have to order the vertices, and update precedences. The second phase of the heuristic (Section 4.2.2) is $O(|\Psi| \cdot (|Q| + |E_q|))$. The worst case scenario happens when we have to run a BFS for each dependence due to the need to implement swaps.

5 Evaluation

In this section we evaluate the performance, in terms of time and effectiveness, of our algorithms. We use the exact dynamic programming algorithm as the reference implementation, and compare it against heuristic solutions to qubit allocation, including the algorithm that we have discussed in Section 4.2, and implementations currently available in the IBM Quantum Experience repository. Throughout this section, we shall try to provide answers to the following research questions:

- **[RQ1]**: what is the effectiveness of the different approaches to solve qubit allocation, when they are applied on actual quantum circuits;
- **[RQ2]**: what is the runtime behavior of the different solutions to qubit allocation.
- **[RQ3]**: what is the impact of the coupling graph on the different algorithms that solve qubit allocation.

Before we analyze each of these questions, we describe our experimental setup, in terms of benchmarks, competing approaches and runtime environment.

Benchmarks: There is no established standard benchmark suite for quantum compilers, as this is a relatively new research field. Therefore, we have gathered a small collection of benchmarks, made of the QASM programs that IBM has released. To complement this set of microbenchmarks, we have also generated synthetic QASM programs. These programs are randomly generated quantum circuits with uniformly distributed dependences. This suite of random benchmarks consists of 10 sets of 33 programs each. Each set contains programs with $\Psi = 10, 20, 40, \ldots, 620$ and 640 dependences. Figure 13 list the actual programs that we use as benchmarks.

The Competing Approaches: we compare six different solutions of qubit allocation. Two of them subsume the ideas discussed in this paper: wpm, our heuristic (Section 4.2); and dynprog, the optimal algorithm (Section 4.1). Two other implementations have been taken from open source projects: ibmmapper, implemented in the IBM Quantum Experience SDK, and qubiter, present in the Qubiter project [3]. Finally, the two implementations left: random, and wqubiter, result from small tweaks that we have performed onto our heuristic and in one of the IBM algorithms. Below we provide a short description of each of these four competitors:

- random: this algorithm uses the same idea as the second stage of our heuristic (Section 4.2.2), but it randomizes the initial mapping; hence, it does not use our pre-allocation phase (Section 4.2.1). We implemented this algorithm to check the impact of the initial mapping onto the overall solution that our heuristic delivers to qubit allocation.
- qubiter: this algorithm is the existing implementation of qubit allocation from the Qubiter compiler project that targets the IBM qx2 computer. The algorithm relies solely on reversal and bridge operations, without using any swap operation. In other words, it never changes the mapping from the logical qubits to the physical qubits. This implementation assumes that the target quantum computer is the IBM qx2, whose coupling graph appers in Figure 2 (a).
- wqubiter: this algorithm is an improvement of qubiter, which uses our initial pseudo-qubit placement algorithm (Section 4.2.1). Notice that qubiter's original implementation would use a kind of random placement as the initial mapping: pseudos are sorted according to their ID, and then assigned, in order, to the physical qubits.
- ibmmapper: This qubit allocator is part of IBM's compiler and runtime infrastructure. The algorithm has been implemented as a Python library called qiskit-sdk-py. Like qubiter,

[3] https://github.com/artiste-qb-net/qubiter

Id	Name	$\|\Psi\|$	Description
rb	rb	2	Example of a single instance of two-qubits randomized benchmarking
qec	qec	4	Repetition code to correct quantum errors.
w	W-state	9	Generating a 3-qubit W-state using Toffoli gates.
grv	qubit_grover_50	25	Grover's search algorithm over three qubits.
pea	pea_3_pi_8	42	4-bit Phase Estimation algorithm for a phase 3pi/8 using 5 qubits.
tel	teleport	2	Quantum Teleportation example.
mod	7x1mod15	9	Implementation of U7 (7xN mod 15) - gate used in Shor's Algorithm.
qft	qft	12	Quantum Fourier Transform on 4 qubits.
ipea	ipea_3_pi_8	30	4-bit Iterative Phase Estimation algorithm for phase 3pi/8 using two qubits.

Figure 13. Quantum programs available in the IBM repository, with the IDs that we use to identify them in this paper, the number of dependences they contain ($|\Psi|$), and a short description of the algorithm.

ibmmapper targets the IBM qx2 computer. This allocator solves the connectivity constraints by dividing the quantum program into a sequence of layers, such that each layer corresponds to a set of independent operations (operations that do not use the same qubit). To map the qubits in each layer, they try to minimize the sum of a distance function between the vertices inside the dependences in this layer, while applying up to $2 * |Q| - 1$ swaps. If, after these swaps, the mapping does not satisfy all dependences from the layer, the algorithm fallbacks onto layers with one operation each. The distance function that ibmmapper tries to minimize is given by $dist_{q_0 q_1} = (1 + r) \cdot d(q_0, q_1)^2$, where $0 \leq r \leq 1$ is a random number and $d(q_0, q_1)$ is the number of control dependences between q_0 and q_1: The complete code of ibmmapper contains more components than just the solution of qubit allocation. In particular, ibmmapper contains optimizations to simplify quantum circuits, which are orthogonal to qubit allocation. In this paper, we only use the module of ibmmaper that solves qubit allocation (Definition 3.3).

We shall compare the different algorithms along two dimensions: the cost of the final circuit they produce – a metric that approximates the number of operations executed by the quantum computer; and the time necessary to solve qubit allocation. Notice that we have not implemented ibmmapper in our compiler; hence, its runtime serves only as a reference.

Runtime Environment: We have created a front-end compiler in C++ for the QASM language. All competing algorithms, except ibmmapper's allocator, have been implemented in this compilation infrastructure. Ibmmapper runs within the program qiskit-sdk-py r0.3, that IBM makes available on Github (downloaded on September, 1^{st} 2017). The machine in which we run all the allocators is an Intel Core i7-4700MQ computer with 8GB of RAM and a clock of 2.4GHz.

5.1 RQ1 – Effectiveness

An allocator A_1 is more effective than an allocator A_2, if, given the same inputs (coupling graph and dependences), A_1 produces a circuit that costs less than A_2. We define this cost as the sum of the costs of all transformations used by such allocator. Our cost function for each transformation θ is determined by the number of quantum gates necessary to implement θ. We use the following cost function: $C(\theta_c) = 0$ for CNOT gate; $C(\theta_r) = 4$ for Reversal; $C(\theta_s) = 7$ for Swap; and $C(\theta_b) = 10$ for Bridge.

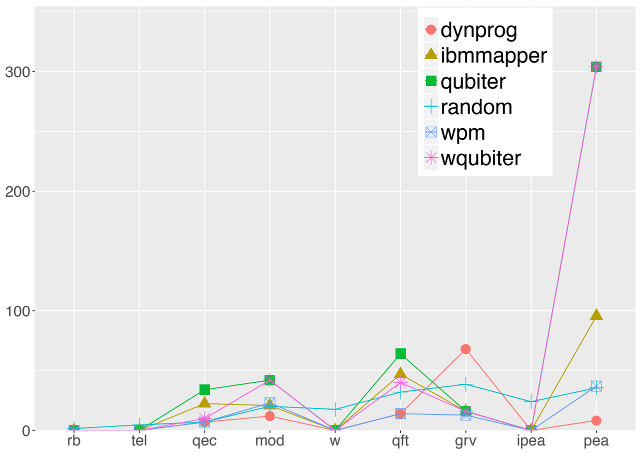


Figure 14. Cost (y-axis) of qubit allocation using each algorithm on each quantum program seen in Figure 13.

Figure 14 shows the costs produced by each algorithm for our actual quantum circuits. Our heuristic (wpm) has found the exact solution in 7, out of 9, cases. The other two cases (mod and pea) were within an 1.92 and a 2.64 factor of the exact solution. On these two benchmarks, the cost obtained by starting with a random configuration (random) was slightly worse than when we use wpm. Thus, the initial placement is useful to reduce allocation costs.

IBM's ibmmapper was, in general, outperformed by the other algorithms, even though it managed to find the best solution for 5 benchmarks. For mod, pea, qec and qft, the cost found by ibmmapper was 2.42, 9.42, 2.72 and 3.13 times worse than the exact solution. Qubiter and its variation, wqubiter, have obtained similar results, although wqubiter achieved small gains on qubiter. Their worst results happen in the same benchmarks where ibmmapper did not fare well. However, for pea, qubiter and wqubiter did more than three times worse than ibmmapper. Wqubiter did 3.4 and 1.6 times

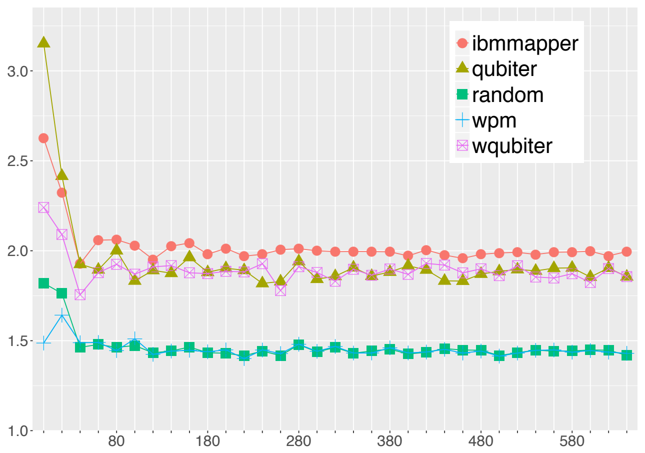


Figure 15. Relative cost of the solution compared to the optimal on synthetic circuits, as a function of the number of dependences. From left to right we have programs from 10 dependences to 640 dependences on the extreme right.

Algo.	Mean	σ	Algo.	Mean	σ
ibmmapper	1.99	0.08	qubiter	1.87	0.09
random	1.44	0.026	wpm	1.44	0.03
wqubiter	1.87	0.08			

Figure 16. Cost found for circuits with 640 dependences, compared against the optimal algorithm. The lower the mean, the closer to the optimum.

better than qubiter on qec and qft, which indicates that our initial mapping can improve other algorithms.

Figure 15 compares the allocators on synthetic circuits, showing how the heuristics fare in circuits of increasing complexity. The first stage of our heuristic is more beneficial when the number of dependences is low. As circuit complexity grows, the second phase of our heuristic starts to have more impact. Our heuristic, wpm, outperforms the other heuristic methods for the IBM qx2 architecture. Figure 16 shows the mean and standard deviation of the results obtained when executing each algorithm with quantum programs of 640 dependences. These are the largest programs we have, and the ones that give us the lowest standard deviation. Numbers compare each heuristic against the optimal algorithm. For these programs, our solution is 27% better than ibmmapper's and 22% better than qubiter's.

5.2 RQ2 – Runtime

Figure 17 compares the time spent by each of the algorithms when compiling the benchmarks in the IBM repository. Although the performance of dynprog is competitive with the Python-based ibmmapper, on the 5-qubit computer, its exponential complexity restricts it to configurations with fewer than a dozen qubits. The four heuristics implemented in

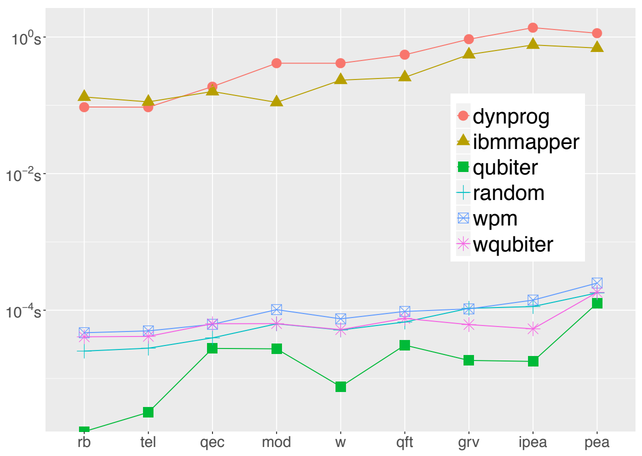


Figure 17. Time, in seconds scaled in log10 (y-axis), spent by the algorithms, when executed with each quantum program.

C++ presented similar running times. With 640 dependences, the difference between the fastest heuristic, qubiter, and the slowest, wpm, was less than 1.6ms.

5.3 RQ3 – The qx3 Computer

At the time of writing this paper, IBM released a new quantum architecture: the qx3, with 16 qubits. We have tested our heuristic (wpm) and ibmmapper on this architecture[4]. Figure 18 shows the result of this comparison. Usually, ibmmapper yields better results than wpm in the larger coupling graph, with an allocation cost on average 11% better. However, it runs very slowly in this setup. Some of this slowdown is an artifact of implementation: we are comparing Python against C++. However, the asymptotic growth of ibmmapper's runtime shows also worse behavior. For the smallest program, it is 15,730x slower; for the largest, it is 27,559x slower than wpm. The large graph benefits ibmmapper's ability to consider multiple dependences at the same time. After its initial placement phase, wpm takes decisions greedily, based on the current labeling, and the next dependence that must be satisfied. In qx2, physical qubits are so constrained that this simple approach outperforms ibmmapper's more holistic view.

6 Related Work

Quantum computing [4], and the notion of universal quantum computers [10] date back to the eighties. In the late nineties we saw the first quantum algorithms with practical purpose, such as integer factorization [38] and database search [18]. Programming languages that let developers interact with quantum machines came later [1, 15, 36, 39].

[4]We did not run qubiter or wqubiter on the qx3 computer, because they only work on qx2; and wpm consistently outperforms random.

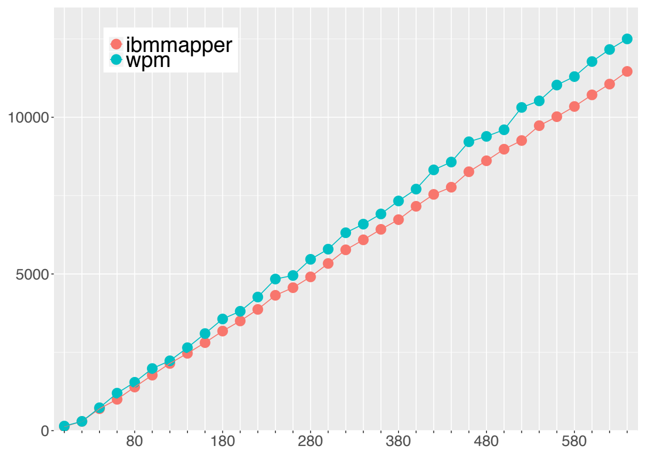


Figure 18. Costs of circuits produced for the qx3 architecture. X-axis shows random programs, varying from 10 to 640 dependences.

Because quantum computers are so recent, so is the interest on quantum compilers. Except for some early work [41], compiler frameworks that translate high-level languages to quantum gates have only been proposed in recent years [17, 19, 21, 28, 42]. Most of them involve solving the qubit allocation problem as part of the compilation flow when targeting partially-connected qubit machines like superconducting quantum computers. Therefore, the algorithms presented here are applicable to all these frameworks. We evaluated the qubit allocators of open-source projects in Section 5. Among classical architectures, clustered VLIW processors also have connectivity constraints between registers. However, the clustered VLIW register allocation problem is very different than the qubit allocation problem. The former is tightly linked with instruction scheduling [7], whereas this degree of freedom is not available in reversible circuits.

On Prior Solutions to Qubit Allocation. There has been previous attempts to solve qubit allocation [20, 26, 29, 32, 37]. The main difference between them and our work is the fact that they focus on particular topologies of coupling graphs, and use only swaps to implement the transitions between different logical-to-physical qubit mappings. In what follows, we shall discuss some earlier work, starting with Maslov *et al.* [29]. In 2008, they have formalized an instance of the problem similar to our Definition 3.2, and have presented an exponential-time heuristic to solve it. Similar to ibmmapper, this heuristic partitions CNOT gates into sets that can be solved without swaps. Maslov *et al.* find these partial solutions via graph isomorphism (between the coupling graph and a subset of dependences). They use a heuristic to insert swaps connecting different partitions of the quantum circuit.

In 2014, Shafaei *et al.* [37] have proposed a methodology to map logical into physical qubits based on Mixed Integer Programming (MIP) [43]. They focus on coupling graphs having a grid architecture, and rely on this assumption to provide a simple and elegant algorithm. In this paper, we assume a general topology for the coupling graph. Furthermore, like Maslov *et al*, Shafaei *et al.* restrict the set of allowed transformations to swaps. Finally, whereas we use dynamic programming to find an exact solution to qubit allocation, they employ MIP, a different method. Both these exact solutions are exponential in their worst case. Along similar lines, Pedram *et al* [32] use Minimum Linear Arrangement (MINLA)[5] to solve qubit allocation on 1D grid architectures, again, using only swaps to ensure the correct semantics of the implementation of the quantum circuit.

Like Shafaei *et al.*, Lin *et al.* [26] also present a solution to qubit allocation in 2D architectures. However, contrary to them, Lin *et al.* rely on heuristics to find said solution. Similar to the algorithm that we have discussed in Section 4.2, they split allocation into two phases, which they have called *placement* and *routing*. Placement fills a role similar to the algorithm in Figure 9. Routing, in turn, would have a purpose similar to the algorithm in 11. Nevertheless, our heuristics use different techniques, given that we deal with general coupling graphs, and resort to operations other than swaps, when transforming quantum circuits.

7 Conclusion

This paper has presented exact and heuristic solutions to the qubit allocation problem. Along this discussion, we have defined the problem, and presented complexity results for it. Our algorithms, including the exact solution, compare favourably against the implementations of qubit allocators that we were aware of. This paper is one more step in the path towards more mature compilers for quantum programs; however, much work is still left to do in the field. We leave as future work the design and implementation of qubit allocators that attempt to maximize the kind of gate sequence simplifications seen in Figure 4. Qubit allocation could also be improved in future work by taking advantage of static knowledge of quantum state and entanglement. For instance, some qubits used as *ancillae* are periodically reset to a known non-entangled state, and are thus interchangeable at these points. This information could be obtained either from high-level language constructs [19] or by static analysis [21, 33].

Acknowledgment

This research is part of the Prospiel Associate Team project, which is supported by a joint grant from Inria and FAPEMIG. Fernando Pereira and Vinícius Santos are supported by the Brazilian Research Council (CNPq). Marcos Siraichi is sponsored by the Brazilian Ministry of Education (through CAPES).

[5]For further details on MINLA, we recommend the introduction written by Jordi Petit [35]

References

[1] Steven Balensiefer, Lucas Kregor-Stickles, and Mark Oskin. 2005. An Evaluation Framework and Instruction Set Architecture for Ion-Trap Based Quantum Micro-Architectures. In *ISCA*. IEEE, Washington, DC, USA, 186–196.

[2] Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical review A* 52, 5 (1995), 3457.

[3] Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16 (1958), 87–90.

[4] Paul Benioff. 1980. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics* 22, 5 (1980), 563–591.

[5] Édouard Bonnet, Tillmann Miltzow, and Pawel Rzazewski. 2016. Complexity of Token Swapping and its Variants. *CoRR* arXiv:1607.07676, Article 2 (2016), 23 pages.

[6] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6 (1981), 47–57.

[7] Josep M Codina, Jesús Sánchez, and Antonio González. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA, USA, 175–184.

[8] Stephen A. Cook. 1971. The Complexity of Theorem-proving Procedures. In *STOC*. ACM, New York, NY, USA, 151–158.

[9] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. *Open Quantum Assembly Language*. IBM, Armonk, NY, USA.

[10] D. Deutsch. 1985. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 400, 1818 (1985), 97–117.

[11] Simon J. Devitt. 2016. Performing quantum computing experiments in the cloud. *Phys. Rev. A* 94, 3 (2016), 032329.

[12] Michel H Devoret, Andreas Wallraff, and John M Martinis. 2004. Superconducting qubits: A short review. *arXiv* cond-mat/0411174 (2004), 1–41.

[13] Martin Farach and Vincenzo Liberatore. 1998. On local register allocation. In *SODA*. ACM, New York, NY, USA, 564–573.

[14] Jay M Gambetta, Jerry M Chow, and Matthias Steffen. 2017. Building logical qubits in a superconducting quantum computing system. *NPJ Quantum Mechanics* 3, Article 2 (2017), 7 pages.

[15] Simon J Gay. 2006. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science* 16, 4 (2006), 581–600.

[16] Dario Gil. 2017. The Future of Computing: AI and Quantum. Online video.

[17] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *SIGPLAN Notices*, Vol. 48. ACM, New York, NY, USA, 333–342.

[18] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *STOC*. ACM, New York, NY, USA, 212–219.

[19] Thomas Häner, Damian S. Steiger, Krysta M. Svore, and Matthias Troyer. 2016. A Software Methodology for Compiling Quantum Programs. *CoRR* abs/1604.01401 (2016), 1–14.

[20] Ali Javadi-Abhari, Pranav Gokhale, Adam Holmes, Diana Franklin, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong. 2017. Optimized Surface Code Communication in Superconducting Quantum Computers. In *MICRO*. ACM, New York, NY, USA, 692–705.

[21] Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Computing Frontiers*. ACM, New York, NY, USA, 1.

[22] Jun Kawahara, Toshiki Saitoh, and Ryo Yoshinaka. 2017. The Time Complexity of the Token Swapping Problem and Its Parallel Variants. In *WALCOM*. Springer, Heidelberg, Germany, 448–459.

[23] Jens Koch, Terri M. Yu, Jay Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. 2007. Charge-insensitive qubit design derived from the Cooper pair box. *Phys. Rev. A* 76, 1 (2007), 04319.

[24] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. 2007. Aliased Register Allocation for Straight-Line Programs Is NP-Complete. In *ICALP*. Springer, Heidelberg, Germany, 258–273.

[25] Daniel A Lidar and Todd A Brun. 2013. *Quantum error correction*. Cambridge University Press, Cambridge, UK.

[26] C. C. Lin, S. Sur-Kolay, and N. K. Jha. 2015. PAQCS: Physical Design-Aware Fault-Tolerant Quantum Circuit Synthesis. *Transactions on Very Large Scale Integration (VLSI) Systems* 23, 7 (2015), 1221–1234.

[27] Chris Lomont. 2003. Quantum Circuit Identities. *CoRR* arXiv:quant-ph/0307111 (2003), 1–6.

[28] Dmitri Maslov. 2017. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics* 19, 2 (2017), 023035.

[29] D. Maslov, S. M. Falconer, and M. Mosca. 2008. Quantum Circuit Placement. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 4 (2008), 752–763.

[30] M Mohseni, P Read, H Neven, S Boixo, V Denchev, R Babbush, A Fowler, V Smelyanskiy, and J Martinis. 2017. Commercialize early quantum technologies. *Nature* 543, 7644 (2017), 171.

[31] Michael A Nielsen and Isaac Chuang. 2000. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, UK.

[32] M. Pedram and A. Shafaei. 2016. Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures. *Circuits and Systems Magazine* 16, 2 (2016), 62–74.

[33] Simon Perdrix. 2008. Quantum entanglement analysis based on abstract interpretation. In *SAS*. Springer, Heidelberg, Germany, 270–282.

[34] Fernando Magno Quintao Pereira and Jens Palsberg. 2006. Register Allocation after Classic SSA elimination is NP-complete. In *FOSSACS*. Springer, Heidelberg, Germany, 79–93.

[35] Jordi Petit. 2003. Experiments on the Minimum Linear Arrangement Problem. *J. Exp. Algorithmics* 8 (2003), 1–33.

[36] Peter Selinger. 2004. A brief survey of quantum programming languages. In *Functional and Logic Programming*. Springer, Heidelberg, Germany, 61–69.

[37] A. Shafaei, M. Saeedi, and M. Pedram. 2014. Qubit placement to minimize communication overhead in 2D quantum architectures. In *ASP-DAC*. IEEE, Washington, DC, USA, 495–500.

[38] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *Journal on Computing* 26, 5 (1997), 1484–1509.

[39] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2017. A Practical Quantum Instruction Set Architecture. *arXiv* arXiv:1608.03355 (2017), 1–15.

[40] Krysta M. Svore, Alfred V. Aho, Andrew W. Cross, Isaac Chuang, and Igor L. Markov. 2006. A Layered Software Architecture for Quantum Computing Design Tools. *Computer* 39, 1 (2006), 74–83.

[41] Robert R Tucci. 1999. A Rudimentary Quantum Compiler (2nd Ed.). *arXiv* quant-ph/9902062 (1999), 1–25.

[42] Dave Wecker and Krysta M Svore. 2014. LIQUi|⟩: A software design architecture and domain-specific language for quantum computing. *arXiv* quant-ph:1402.4467 (2014), 1–14.

[43] Laurence A. Wolsey. 2008. Mixed Integer Programming. *Encyclopedia of Computer Science and Engineering* Online, ecse244 (2008), –.

[44] Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. 2014. *Swapping Labeled Tokens on Graphs*. Springer, Heidelberg, Germany, 364–375.