Scalable Optimal Layout Synthesis for NISQ Quantum Processors

Wan-Hsuan Lin[1], Jason Kimko[1], Bochen Tan[1], Nikolaj Bjørner[2], Jason Cong[1]
[1]Computer Science Department, University of California, Los Angeles, CA, USA,
[2]Microsoft Research, One Microsoft Way, Redmond, WA, USA
{wanhsuanlin, kimko, bochentan}@ucla.edu; nbjorner@microsoft.com; cong@cs.ucla.edu

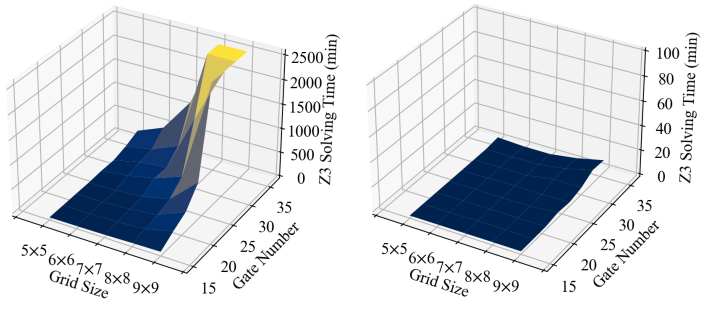
***Abstract*—Due to its effect on the success rate of a quantum circuit, quantum layout synthesis is a crucial step for circuit compilation. As such, having a layout synthesis tool that provides high solution quality is important to maximize circuit performance and fidelity for NISQ application. Previous heuristic approaches have been shown to be far from optimal when evaluated on known-optimal benchmarks. Alternatively, exact layout synthesis tools can generate optimal results with the aid of constraint solvers but generally suffer from scalability issues because of inefficient encodings and slow optimization methods. In this paper, we propose a scalable optimal layout synthesis tool that improves upon previous works, through a more succinct problem formulation as well as better encoding techniques. Additionally, we implement a depth and SWAP count optimization feature that performs iterative refinement under a fixed time budget. Experimental results show that for depth optimization, our tool can achieve a 692× speedup over the state-of-the-art optimal layout synthesis, and for SWAP optimization, we can obtain a 6,957× speedup on average. Compared to a leading heuristic-based synthesizer, for depth optimization, we can solve circuits consisting of 54 program qubits and 1726 gates within 11 hours with an 18× depth reduction and by 12× SWAP count reduction on average.**

I. Introduction

Quantum computing has increasingly been attracting more research interest due to its potential to achieve an exponential speedup over classical computing with a variety of problems such as factorization [1] and unstructured search [2]. Currently in the noisy intermediate-scale quantum (NISQ) era of quantum hardware development, executable programs are limited to small problem sizes (around one hundred qubits [3], [4]) and program outputs are extremely sensitive to system noise. A quantum computer can be realized using a variety of qubits [5], [6], [7]. The superconducting qubit is currently the most mature approach and appears promising for large-scale quantum computing. Different superconducting quantum computers support different quantum gate sets and qubit connectivity, so in order to execute a quantum program, we need to perform logic and layout synthesis to accommodate these hardware specifications. First, logic synthesis will translate gates in the circuit into those supported by the hardware. Then, layout synthesis will perform gate scheduling and placement while adapting to the hardware connectivity by inserting additional SWAP gates. In the NISQ era, the success rate of quantum programs suffers from short qubit coherence time, imperfect gate operations, and environmental noises. Thus, an effective layout synthesizer should minimize the number of inserted SWAP gates to avoid reducing the success rate via a prolonging of the circuit execution time, i.e., *circuit depth*, and an increase of the total gate count.

Layout synthesis has been proven to be NP-hard [8], [9]. Many heuristic approaches have been proposed to perform layout synthesis. Most works focus on minimizing the number of SWAP insertion. In [10], Zulehner *et al.* offer a depth-based partitioning and an A* search algorithm whose cost function is calculated based on qubit distance and SWAP gate count. However, this greedy partition algorithm may lead to a sub-optimal solution. In [11], Li *et al.* suggest an iterative SWAP selection method with a cost function parameterized only on local gate information, and thus may sacrifice global optimality. Other works [12], [13], [14], [15] have been shown to be far from optimal by [8].

Several exact approaches have been introduced to obtain optimal results in terms of SWAP insertion count by formulating layout synthesis as a constraint satisfaction problem. Wille *et al.* [16], [17] leverage a pseudo-Boolean optimizer and a satisfiability modulo theories (SMT) solver to minimize additional SWAP gate cost. Siraichi *et al.* [9] perform gate-by-gate processing and apply dynamic programming to solve the problem. Bhattacharjee *et al.* [18] and Nannicini *et al.* [19] both utilize a linear programming approach on layer-by-layer basis. Molavi *et al.* [20] encode layout synthesis to maximum satisfiability [21] and propose a constraint relaxation method by slicing a circuit into layers and solving them individually to improve scalability. However, Tan and Cong [22], [23] demonstrate that these gate-by-gate and layer-by-layer methods impose unnecessary constraints, and thus may lead to sub-optimality. To overcome this issue, they propose the state-of-the-art optimal layout synthesis tool, OLSQ, which constructs a gate-dependency graph and guarantee optimality for either depth or SWAP count objectives via an SMT solver. However, our analysis shows their method suffers from having redundant variables, a sub-optimal problem encoding, and a slow optimization module, leaving a large amount of untapped potential in the SMT solver. Fig. 1a illustrates the impact of the layout synthesis problem size on the SMT solving time (excluding optimization) for models generated by OLSQ. These SMT instances encode QAOA [24] applications with gate count ranging from 15 to 36 and a grid-based coupling graph varying from 5-by-5 to 9-by-9. According to Fig. 1a, compiling a 36-gate circuit on a 9-by-9 grid architecture using OLSQ's formulation takes more than 40 hours. Although testing a 9-by-9 grid architecture was sufficient at the time of OLSQ development, the leading superconducting quantum processors today have more than one hundred physical qubits, e.g., IBM Eagle has 127 qubits [4]. In addition, as qubit coherence time increases and gate fidelity improves, we can execute circuits with larger gate counts. Therefore, developing a scalable optimal layout synthesis tool is of urgent need.



(a) OLSQ. (b) Ours.

Fig. 1: Impact of coupling graph grid size and circuit gate count on SMT solving time using Z3 for the OLSQ layout synthesizer versus ours.

To overcome the aforementioned problems in OLSQ, we develop a scalable optimal layout synthesizer based on a more succinct SMT formulation along with a more efficient encoding method. Fig. 1b demonstrates that our tool can vastly improve time-to-solution compared to OLSQ and appears to scale into the near-term future. For the cases that take more than 40 hours with the models generated by OLSQ, our models take less than 10 minutes to solve. On average, our tool achieves a 387× speedup over OLSQ. In addition, our SWAP optimization method is designed to first obtain any valid solution and then perform an iterative refinement until an optimal solution is reached or a time budget is exhausted. The main contributions of this paper are:

- We present a succinct SMT formulation along with an efficient bit-vector SMT encoding for quantum layout synthesis.
- We propose a depth and SWAP optimization feature that iteratively refines a solution within a specified time budget.
- Our new formulation can achieve up to a 692× speedup over the leading optimal layout synthesis, OLSQ, for depth optimization, and an average of 6,957× speedup for SWAP optimization.
- Our tool demonstrates an average of 7× for depth reduction and an average of 12× for SWAP reduction compared to the leading heuristic layout synthesizer.

This work is partially supported by multiple industrial sponsors, including Amazon (under the Science Hub program) and NEC (under the CDSC industry partnership program).

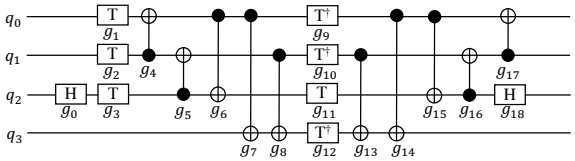


Fig. 2: A circuit implementing the Toffoli gate with one ancilla qubit.

II. Background

In this section, we first define the quantum layout synthesis problem, then introduce constraint solving techniques to facilitate SMT solving, and lastly, present the leading optimal layout synthesis tool, OLSQ.

A. Quantum Layout Synthesis

Quantum layout synthesis, or qubit mapping, is the process of mapping program qubits in a circuit to physical qubits on a quantum processor, followed by scheduling gate execution. First, we define terminology for the inputs:

- Quantum program: a sequence of gates G and their target program qubits. Since quantum processors only support one- or two-qubit gates, the gates to be scheduled are one of these two types. In this paper, we denote the set of program qubits by Q, the set of single-qubit gates by G_1, and the set of two-qubit gates by G_2. Fig. 2 shows an example quantum circuit implementing the Toffoli gate [25], where each horizontal wire represents one program, or logical, qubit. H, T, and T† gates are single-qubit gates, and CNOT gates are two-qubit gates.
- Coupling graph: a graph (P, E), where each vertex $p \in P$ is a physical qubit and each edge $e \in E$ is a connection between two qubits that captures the connectivity of a quantum processor. In this paper, we sometimes refer to an edge between qubits p and p' as $e(p, p')$. We will also occasionally access its qubits via $e.p$ and $e.p'$. Fig. 3 shows the coupling graph of IBM QX2, which consists of five physical qubits $p_0, p_1, \ldots, p_4$ and six edges $e_0, e_1, \ldots, e_5$.

The outputs of the layout synthesizer consist of a set of mapping $\{\pi_q^t \mid q \in Q, 0 \le t < T\}$, where t is a circuit time step and T is the circuit depth, and a schedule $S = \{t_g \mid g \in G\}$, where t_g is the execution time step for gate g. Fig. 4 exhibits a valid layout synthesis result for the Tofolli circuit on IBM QX2 with 19 time steps and two inserted SWAP gates. Each SWAP gate is decomposed to three CNOT gates marked by the dotted rectangle. The qubit mapping is depicted above each wire at the start of the circuit and after each SWAP insertion. For instance, q_0 is initially mapped to p_2, i.e., $\pi_0^0 = p_2$, and then updated to $\pi_0^9 = p_3$ after the first SWAP gate. As mentioned previously, in order to achieve a high success rate, the objective for the layout synthesis is to minimize the circuit execution time, i.e., circuit depth, and the number of inserted SWAP gates.

A valid layout synthesis result requires a valid qubit mapping and gate schedule that satisfies the following constraints:

1) Mapping injectivity: Two program qubits cannot be mapped to the same physical qubit at every time step, i.e., $\pi_q^t \neq \pi_{q'}^t$ if $q \neq q'$.
2) Gate dependency: If two gates g and g' act on the same program qubit and g appears before g' in G, then g should be executed before g'. For instance, in the Toffoli circuit (Fig. 2), g_3 should be executed before g_5 on q_2. According to the input circuit, we can build the dependency gate list D that consists of pairs of gates (g, g') where g should be executed before g'. Fig. 5 shows the gate dependencies extracted from the Tofolli circuit.
3) Valid two-qubit gate scheduling: Every two-qubit gate is scheduled on two adjacent physical qubits in the coupling graph. In Fig. 4, g_7 can be executed at time $t = 5$ because $\pi_0^5 = p_2$ and $\pi_3^5 = p_0$ are adjacent physical qubits in Fig. 3.
4) SWAP transformation: If a two-qubit gate acts on non-adjacent physical qubits, the synthesizer will modify the mapping by inserting SWAP gates so that the involved logical qubits are physically adjacent. For example, in Fig. 4, g_8 cannot be executed at time 6 because $\pi_1^6 = p_3$ and $\pi_3^6 = p_0$ are non-adjacent physical qubits. After the first SWAP gate finishes, π_0^9 and π_1^9 are mapped to adjancent physical qubits p_3 and p_2, and thus g_8 can be scheduled at time 9. On the other hand, if no SWAP gate is inserted, the mapping should be the same, e.g., $\pi_0^0 = \pi_0^1$.
5) Valid SWAP insertion: A SWAP insertion is valid if it does not overlap with other gates. For instance, in Fig. 4, g_8 cannot be executed until the SWAP gate is finished at $t = 8$.

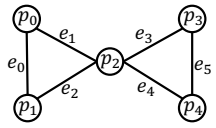


Fig. 3: Coupling graph of IBM QX2

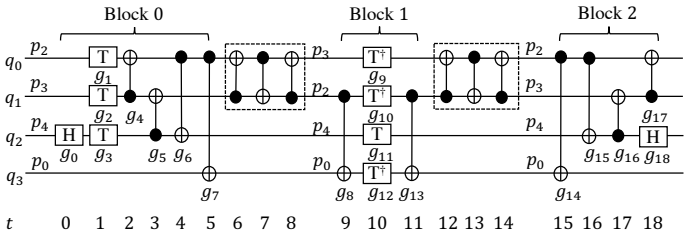


Fig. 4: Transition-based layout synthesis result for Toffoli circuit.

B. Constraint Solving

Many real-world problems can be formulated as a SAT problem and solved using a SAT solver. However, the performance of modern SAT solvers typically is heavily affected by the method of the encoding itself [26], [27]. Satisfiability modulo theories (SMT) augments SAT by enabling various logical domains, or theories, that extend beyond Boolean logic, such as the domains of equality and uninterpreted functions (EUF) and arithmetic. While this does allow for a higher-level and more expressive formulation, it also offers different means of encoding a problem, and similarly to SAT, the performance of SMT solvers is linked to the effectiveness of the encoding method. For example, Bjørner *et al.* [28] demonstrates a significant performance improvement for a finite domain combinatorial problem by switching to a bit-vector encoding from an arithmetic one. In this case, the use of fixed-sized bit-vectors encodes the finite domain problem more narrowly than integer arithmetic and also enables specialized solving operations such as bit-blasting.

C. OLSQ and TB-OLSQ

OLSQ is an optimal quantum layout synthesizer proposed by Tan and Cong [22]. It encodes the layout synthesis problem to SMT and applies the Z3 SMT solver [29] to solve the model. Given a quantum circuit and coupling graph, OLSQ defines mapping variables to capture qubit mapping constraints, space-time coordinate variables for gate scheduling, and SWAP variables for enabling SWAP insertions along edges. The SWAP variables are encoded as Boolean while all other variables are encoded as integers. To ensure valid layout synthesis results, its formulation includes constraints (1)–(5) as previously described in Section II-A. Following this problem formulation, OLSQ applies Z3's optimization module to perform either SWAP or depth optimization. To address concerns with scalability, Tan and Cong propose a transition-based version of OLSQ, called TB-OLSQ, which produces a near-optimal solution for SWAP minimization. TB-OLSQ augments OLSQ by introducing a coarse-grained approach that schedules gates into *blocks* between mapping transitions rather than individually, and there is no SWAP gate inside a block. For example, Fig. 4 demonstrates a circuit with three blocks and two layers of SWAP gates.

III. OLSQ2 Algorithm

In this section, we present OLSQ2, an optimal layout synthesizer that is not only based on OLSQ [22] but also our strategies for depth optimization and SWAP optimization. Then, we discuss the different encoding schemes in the SMT solver.

A. Improvement 1: Succinct Formulation

Our formulation is improved upon OLSQ by using fewer variables and constraints and is presented in the following sections.

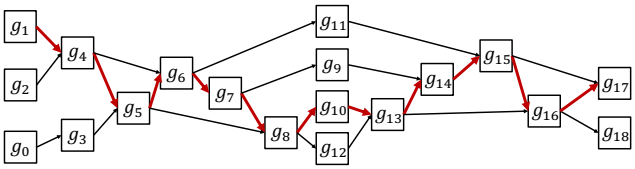


Fig. 5: Longest gate dependency chain (red) for the Toffoli circuit.

1) Variable Definition: Since we need to define variables that account for all time steps, we select an upper bound T_{UB} for the circuit depth, which can be trivially set to the total gate count. From our empirical results, we found $T_{UB} = 1.5 \times T_{LB}$ to be a sufficient upper bound, where T_{LB} is the longest dependency chain in the gate dependency graph. For example, Fig. 5 shows the longest chain of length 12. Terminology for our variables is defined as follows.

- Mapping variable $\pi_q^t = p$ if and only if program qubit q is mapped to physical qubit p at time t.
- Time variable $t_g = t$ if and only if gate g is scheduled to execute at time t.
- SWAP variable $\sigma_e^t = 1$ if and only if a SWAP gate is scheduled on edge e and finishes at time t.

OLSQ also uses a space variable x_g for each gate g to represent the gate position and enforces valid two-qubit gate scheduling via constraints containing space variables that represent the gate's target qubit(s). However, we conclude gate positions can be inferred from mapping variables and time variables, so space variables are redundant. Moreover, using space variables requires additional constraints for consistency checking between mapping, time, and space variables. As a result, our first proposed improvement is to eliminate space variables, thereby reducing the total number of variables by $|G|$ and the total number of constraints by $(|E|-1)\cdot|G_2|\cdot T + |G_1|\cdot T$. In general, reducing the total number of variables in a problem formulation can speed up the solving process as there is less work for a solver to perform. In the following, we describe the constraints employed in our formulation without using space variables.

2) Constraint Construction: After eliminating space variables, we need to reformulate constraints (3) and (5) described in II-A.[1]

Valid Two-Qubit Gate Scheduling A valid gate schedule requires the affected qubits of every two-qubit gate to be physically adjacent at execution time. Therefore, we have:

$$(t_g == t) \implies \bigvee_{e\in E} [((\pi_q^t == e.p) \wedge (\pi_{q'}^t == e.p')) \vee ((\pi_{q'}^t == e.p) \wedge (\pi_q^t == e.p'))] \text{ for every } 0 \le t < T_{UB} \text{ and } g(q,q') \in G_2. \tag{1}$$

Note that the length of each individual constraint is increased, but the experimental result shows it is well compensated by the reduction of variables and constraints.

Valid SWAP Insertion Because OLSQ represents the gate position by space variables, we need to rewrite the constraints that prevent SWAP gates from overlapping with other gates. Since a SWAP gate may be a composition of other gates, e.g., 3 CNOT gates, our formulation accounts for any arbitrary swap depth S_D. Additionally, while a SWAP is occurring, we must wait for S_D time steps before allowing other gates to be scheduled on the affected qubits. Recall that, even though a SWAP gate's execution time is defined by its last time step t, it actually starts at time step $t - S_D + 1$. Let $E_p = \{e \in E \mid e \text{ contains } p\}$ and $overlap(t,q,e) := (\pi_q^t == e.p) \vee (\pi_q^t == e.p')$ describe the condition where the mapping of logical qubit q at time t is on an endpoint of edge e. First, a SWAP gate should not overlap with single-qubit gates:

$$[(t_g == t') \wedge overlap(t,q,e)] \Rightarrow (\sigma_e^t == 0) \quad \text{for every } S_D \le t < T_{UB},\ t - S_D < t' \le t,\ g(q) \in G_1 \text{ and } e \in E. \tag{2}$$

Then, a SWAP gate should not overlap with two-qubit gates:

$$\{(t_g == t') \wedge [overlap(t,q,e) \vee overlap(t,q',e)]\} \Rightarrow (\sigma_e^t == 0) \quad \text{for every } S_D \le t < T_{UB},\ t - S_D < t' \le t,\ g(q,q') \in G_2 \text{ and } e \in E. \tag{3}$$

[1]Constraints (3) and (5) correspond to Eq. 3–4 and Eq. 7–8 in [22].

B. Improvement 2: Optimization Strategy

Using our formulation, we may choose to optimize circuit depth or SWAP count. For our experiments, we found that the optimization module in Z3 [30] was not very efficient for our problems. This led us to implement our own optimization routine by iteratively updating an additional constraint that limits the maximum value for either depth or SWAP count and checking for a satisfying solution. In each iteration, if no solution exists under the current bound, we relax the problem by increasing the bound; if a solution does exist, we decrease it. The optimization process is terminated after we reach the optimal value. Note that we are solving a set of similar models that only differ in the bound of the optimization objective. Thus, we are able to make use of *incremental solving* so that learned information [31] from the previous iteration can be reused in later ones to reduce the amount of duplicated work.

1) Depth Optimization: To optimize circuit depth, we add the constraint:

$$\bigwedge_{g\in G} t_g \le T_B, \tag{4}$$

where T_B is the bound for depth and is initialized to T_{LB}. During depth optimization, the size of the solution space grows as the depth bound increases. In order to initiate the optimization process from easier problems with small solution spaces, we start from a tight depth bound. In addition, according to our empirical results, the optimal depth is usually close to T_{LB}. Then, if the formulation is unsatisfiable, we increase T_B to rT_B with $r > 1$. By default, we set r to 1.3 if T_B is less than 100; otherwise, we set r to 1.1. After we find the first satisfying assignment, we decrease T_B by 1 iteratively until we get the unsatisfiable case. Then, the optimal depth is the minimal value that can have a satisfiable assignment. If no solution exists with $T_B < T_{UB}$, we regenerate a new formulation with a larger T_{UB}.

2) Iterative Refinement for SWAP Optimization: To optimize the SWAP count, we have the constraint:

$$\sum_{\substack{0\le t<T_B \\ e\in E}} \sigma_e^t \le S_B, \tag{5}$$

where S_B is the bound for the SWAP count. Since increasing the depth bound may reduce the SWAP count, we perform a two-dimensional search along the depth and SWAP count to generate the Pareto-optimal solutions. The SWAP optimization procedure begins with a depth-optimal solution as starting with a tight depth bound trims the solution space, and thus, accelerates the solving process. After obtaining the optimal SWAP count under the current depth bound, we relax the depth bound and attempt to reduce the SWAP count again. This process terminates under one of two conditions: (1) the time budget is exhausted or (2) there is no reduction in the SWAP count after increasing the depth bound. If the process is terminated by the second condition, the solution is Pareto-optimal in terms of SWAP count under the given depth.

Unlike the depth optimization strategy, we apply an *iterative descent* approach for SWAP count optimization exploiting the *monotone property* of the solution structure, because varying the SWAP bound will not change the solution space but only affect the number of feasible assignments in the solution space. That is, let ϕ be the set of satisfying assignments for the model with SWAP bound S. For two SWAP bounds S and S' with $S < S'$, a satisfying assignment for S will also satisfy the model with SWAP bound S', i.e., $\phi \subseteq \phi'$. As a result, obtaining a satisfying assignment from the model with S would be more difficult than that with S. Therefore, S_B is initialized to the upper bound S_{UB} and lessened by one until reaching the optimal value. Note that the upper bound S_{UB} can alternatively be determined by other heuristic layout synthesizers. In this manner, we can first obtain a valid solution and then perform iterative descent refinement. The optimal SWAP count under a certain depth T is known when the first unsatisfiable case occurs, i.e., if we can find a satisfiable solution with $S_B = S$ at depth T but none with $S_B = S - 1$, then the optimal SWAP count for depth T is S.

C. Improvement 3: Exploration of Different Encoding Schemes

Because variable and constraint encoding has a big impact on solver performance, in this section, we explore different encoding schemes for our formulation. Depending on the chosen encoding, an SMT solver like Z3 can invoke specialized theory solvers during the solving process, each of which has its strength and weakness for certain problem domains. In the context of

our problem, we have the following choices for our variable types: integer, bit-vector, and Boolean. Recall that we already use Boolean for SWAP variables since this value is either 0 or 1. Because the ranges of our mapping and time variables are bounded integers, we can encode them using either the integer or bit-vector type; utilizing integers will trigger an arithmetic theory solver while employing bit-vectors will lower the problem into propositional logic (SAT) via a process called bit-blasting. Note that bit-vectors support basic arithmetic operations, so changing the underlying encoding only affects variable definitions and not their usage in constraints. As exhibited in our experimental results in Section IV-A, using the SAT solver in Z3 is more efficient than the arithmetic theory solver for our problems. Therefore, we choose to encode each mapping variable and time variable by an unsigned bit-vector of length $\lceil \log_2 |P| \rceil$ and $\lceil \log_2 (T_{UB} - 1) \rceil$, respectively.

Constraints can also be encoded in different ways. For example, utilizing the logic of *equality and uninterpreted functions (EUF)*, we can avoid generating pairwise constraints for mapping injectivity. More specifically, to specify an injective function f, we can generate a set of constraints $\{f(x) \neq f(y) \mid x \neq y\}$. Another approach is to define a left inverse function $g = f^{-1}$ and add the constraint $g(f(x)) = x$. To apply this idea to our formulation, we redefine mapping variables to be a mapping function $\pi(q, t) = p$. Then, we define another function $\pi_{inv}(p, t)$ and capture its inverse property through the constraint $\pi_{inv}(\pi(q, t), t) = q$ for $0 \leq t < T_{UB}$. The results in Section IV-A reveal that using EUF to encode injectivity can produce a better performance over employing pairwise constraints with integer type variables.

Boolean cardinality constraints can also be encoded in different ways. As an example, Eq. 5 is a Boolean cardinality constraint that specifies at most S_B Boolean variables can be `True`. In Z3, such a cardinality constraint can be directly encoded by the `AtMost` function. However, if we encode Eq. 5 using the `AtMost` function, Z3 will employ a pseudo-Boolean theory solver. This solver finds an assignment that satisfies a collection of linear pseudo-Boolean constraints, also known as integer linear programming. When comparing the solving time for the formulation with and without the `AtMost` function, we observed that the appearance of the `AtMost` function nullified all/some of the performance gained from utilizing a bit-vector representation. We posit that the pseudo-Boolean theory solver is not as efficient as the SAT solver. As a result, our final encoding of Eq. 5 is a sequential counter circuit [32] in the *conjunctive normal form (CNF)*, so that we can trigger the SAT solver to achieve the best performance.

D. Improvement 4: Use of Coarse-Grained Circuit Model

Inspired by TB-OLSQ, our work also includes TB-OLSQ2, a transition-based version of OLSQ2. In TB-OLSQ2, the gate dependency constraints need to be modified because two dependent gates can still be scheduled in the same block. Furthermore, since the SWAP gates will be scheduled at the end of a block and thus will not overlap with other gates, Eq. 2 and Eq. 3 are removed. The objective of TB-OLSQ2 is to minimize either the block count or the SWAP gate count and can be optimized by the same strategies used in OLSQ2. When optimizing the block count, we adopt the depth optimization strategy described above with T_B initialized to 1. If the formulation is unsatisfiable, T_B will be increased by 1. For the SWAP count optimization, the depth optimization step is changed to block count optimization. Then, we can detect the optimal solution when either the first unsatisfiable case occurs or, additionally, when S_B is equal to the block count because each transition contains at least one SWAP gate. With TB-OLSQ2, we can obtain near-optimal results for the SWAP gate count.

IV. Experimental Results

In this section, we present a comparison of different encoding strategies for our problem and the layout synthesis results using OLSQ2 and TB-OLSQ2. We implemented our proposed algorithm in Python 3.6 and used the Z3's Python API (v4.8.15.0) [29] for the SMT solving and pySAT (v0.1.7) [33] for CNF generation. All experiments were conducted on an Intel Xeon E5-2680 CPU at 2.40GHz and 64 GB of RAM.

To measure the effectiveness of our encoding strategy, we design two experiments. Section IV-A presents the speedup achieved from employing the reduced SMT formulation and different variable encoding methods. Section IV-B shows the advantage of utilizing CNF to encode cardinality constraints. We use the QAOA [34] phase-splitting operator for random 3-regular graphs generated by *networkx* (v2.4) [35] as our benchmark circuits and various grid architectures for the coupling graphs. For QAOA circuits, the SWAP duration is set to 1.

Then, to evaluate the solution quality and scalability of our approach, we test our tool with coupling graphs from existing quantum devices and NISQ quantum circuits; more specifically, those describing Rigetti's Aspen-4 processor with 16 qubits [36], Google's Sycamore processor with 54 qubits [3], and IBM's Eagle processor with 127 qubits [4]. We present these results in Section IV-C. Our benchmark suite includes QAOA circuits, arithmetic circuits from IBM Qiskit, and QUEKO circuits [8], which is used for evaluating layout synthesizers. For non-QAOA circuits, the SWAP duration is set to 3. We compare the performance of our tool against the leading exact layout synthesizer, OLSQ [22] and the state-of-the-art heuristic layout synthesizers, SABRE [11] and SATMap [20].

A. Comparing SMT Encodings

To measure the quality of each SMT encoding technique, we set up six experiments:

1) OLSQ(int): original OLSQ formulation, which uses integer variables.
2) OLSQ(bv): OLSQ formulation with bit-vector variables.
3) OLSQ2(int): our proposed formulation with integer variables.
4) OLSQ2(EUF+int): our proposed formulation with EUF for mapping constraints and integer time variables.
5) OLSQ2(EUF+bv): our proposed formulation with EUF for mapping constraints and bit-vector time variables.
6) OLSQ2(bv): our proposed formulation with bit-vector variables.

For each method, we collect the Z3 solving time for the respective instances generated by `Solver.sexpr()`. Each SMT instance encodes the layout synthesis problem for a QAOA circuit on a 7-by-7 or 8-by-8 grid architecture with T_{UB} fixed to 21 and without the constraint for the SWAP count. This value for T_{UB} is large enough to ensure that the generated instances are all satisfiable.

Table I displays our runtime results. The speedup of each technique against the baseline, OLSQ(int), can be seen in the "Ratio" column. Among the six configurations, the performance of OLSQ(int) is consistently the worst. With the bit-vector encoding, OLSQ(bv) can achieve up to a 32.90× speedup and an average of a 18.87× speedup when compared to OLSQ(int). On the other hand, OLSQ2(int) can outperform OLSQ(int) for all cases with a 3.59× speedup on average, which is in accordance with our hypothesis that reducing the variable count can facilitate the solving process. By using EUF to reduce the number of constraints, OLSQ2(EUF+int) shows a 44.56× speedup on average. Changing to bit-vector variables in OLSQ2(EUF+bv) reduces the performance, which we hypothesize to be due to the presence of EUF logic inhibiting the bit-vector engine, or vice versa. Although performance degrades when employing bit-vector variables along with EUF, utilizing only bit-vector variables for our proposed formulation demonstrates an impressive runtime improvement: OLSQ2(bv) can achieve at least a hundred times speedup on the smaller cases and can reach up to a 2327.41× speedup over OLSQ(int) for larger cases with an average of 692× speedup. In conclusion, our proposed formulation along with a bit-vector variable encoding reaps the greatest boost in performance for our layout synthesis problems.

B. Comparing Cardinality Constraint Encodings

To quantify the benefit of using CNF to encode cardinality constraints, we compare the runtimes of OLSQ, TB-OLSQ, OLSQ2 with `AtMost`, OLSQ2 with CNF, and TB-OLSQ2 with CNF. Here, we use the original implementation of OLSQ and TB-OLSQ. Similarly to the previous section, we measure the Z3 solving time for the SMT instances generated by `Solver.sexpr()`. Each SMT instance encodes the layout synthesis problem for a QAOA circuit on a 5-by-5 grid architecture with T_{UB} fixed to 21 and S_B fixed to 30. For TB-OLSQ and TB-OLSQ2, setting the upper bound on block count to 5 is sufficient to get satisfiable instances. [2]

Our results are shown in Table II. First, we note that the OLSQ2(CNF) encoding can solve all instances within the time limit while also providing

[2]Empirically, T_{UB} for the transition-based model is four times smaller than the non-transition-based model.

TABLE I: Runtime comparison for integer, bit-vector, and EUF formulations. The instances are layout synthesis problems for QAOA circuits on grid architectures with a depth limit of 21 and with an unconstrained SWAP count. An entry "TO" indicates that the case cannot be solved within the 24-hour time limit.

Grid	Qubit/ Gate	OLSQ(int)		OLSQ(bv)		OLSQ2(int)		OLSQ2(EUF+int)		OLSQ2(EUF+bv)		OLSQ2(bv)	
		Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio
7×7	16/24	4352.07	1.00	443.16	9.82	2219.11	1.96	215.58	20.19	840.00	5.18	16.76	259.67
	18/27	13367.83	1.00	638.07	20.95	3729.63	3.58	320.38	41.72	2439.04	5.48	61.93	215.85
	20/30	13004.06	1.00	974.71	13.34	5018.15	2.59	559.57	23.24	3718.45	3.50	29.66	438.44
	22/33	27073.60	1.00	1404.28	19.28	7683.02	3.52	413.36	65.50	4307.89	6.28	40.15	674.31
	24/36	69496.54	1.00	2112.46	32.90	17304.09	4.02	629.96	110.32	10090.08	6.89	29.86	2327.41
8×8	16/24	11879.47	1.00	1502.06	7.91	4098.26	2.90	532.28	22.32	2106.05	5.64	22.14	536.56
	18/27	25830.72	1.00	1748.26	14.78	7857.68	3.29	1102.92	23.42	3825.52	6.75	34.60	746.55
	20/30	84290.19	1.00	2634.05	32.00	12264.60	6.87	1694.44	49.75	5325.02	15.83	248.14	339.69
	22/33	TO	–	2867.26	–	21257.85	–	1647.89	–	15254.48	–	221.73	–
	24/36	TO	–	4829.89	–	28548.72	–	3063.81	–	17555.59	–	213.48	–
	Avg.		1.00		18.87		3.59		44.56		6.94		692.31

a faster time-to-solution across the board. For the instances generated by OLSQ, TB-OLSQ, and OLSQ2(`AtMost`), Z3 can only solve four out of five cases within the time limit. Although OLSQ2(`AtMost`) can achieve a 6.40× speedup over OLSQ on average, it does not outperform OLSQ for all cases. Because the `AtMost` function invokes a pseudo-Boolean solver, we reason that the cost of not using a SAT solver sometimes outweighs the benefit of eliminating space variables and employing bit-vectors. On the other hand, OLSQ2(CNF) leverages the faster SAT engine, and thus achieves up to a 20.86× speedup and an average of 11.71× speedup over OLSQ. With the coarse-grained circuit model, our proposed formulation and encoding demonstrate a significant runtime improvement. Overall, TB-OLSQ2 achieves an average of 6,956.75× speedup over OLSQ and 149.98× speedup over TB-OLSQ for SWAP optimization. In addition, we observe that the solving times were less sensitive to the size of the problem.

C. Depth and SWAP Optimization

In this section, we evaluate the scalability and solution quality of OLSQ2 when optimizing depth or SWAP. First, we demonstrate that our tool is more scalable than the leading exact layout synthesizer, OLSQ [22], by compiling 22 quantum circuits with qubit counts ranging from 7 to 54, gate counts ranging from 24 to 1726, and targeting the Rigetti Aspen-4, Google Sycamore, and IBM Eagle. We summarize the results as follows: OLSQ only managed to solve five cases within the time budget while OLSQ2 was able to handle all cases and even outperformed OLSQ with up to a 157.48× speedup and a 64.25× speedup on average for depth optimization.

Next, we demonstrate our solution quality in terms of depth compared to that of SABRE [11]. Table III shows our results. On average, OLSQ2 can reduce the circuit depth by 6.66×. For the same category of benchmark, the solution quality of SABRE declines as the quantum processor size increases. Using QAOA(16,24) as an example, SABRE reported 27 inserted SWAP operations for the smaller Sycamore architecture, but 64 for the larger Eagle architecture. We observe the same trend when compiling QUEKO circuits for Aspen-4 and Sycamore processors. In addition, for QUEKO benchmarks, the circuit depth of the results generated by OLSQ2 matches the known-optimal depth, showing that OLSQ2 can generate depth-optimal results.

For SWAP optimization, we compare the solution quality of TB-OLSQ2 with those produced by SABRE and SATMap [20]. Table IV contains our evaluation results. We observe that SATMap fails to produce a solution for eight of our twenty experiments, either due to a timeout or out-of-memory error. For the memory error, SATMap reaches the 64GB limit on QAOA(20/30) after solving for 14 hours while TB-OLSQ2 uses less than only 5GB memory for the same problem. On average, TB-OLSQ2 can reduce the SWAP count by 109.65× compared to SABRE and by 12.42× for SATMap.

V. Conclusion and Future Direction

In this paper, we propose an efficient and optimal layout synthesis tool, OLSQ2, that improves upon previous work through a more succinct problem formulation and better encoding techniques. Moreover, we design a SWAP count optimization feature with the ability to iteratively refine a solution under a fixed time budget. To achieve better scalability, we also develop a near-optimal layout synthesis tool, TB-OLSQ2, using a transition-based model. Our experimental results show that our approach can achieve more than a 6,000× speedup over the state-of-the-art optimal layout synthesis tool and exhibits higher solution quality with a 7× depth reduction and 12× SWAP count reduction compared to leading heuristic solvers.

According to our observations, the solving time of OLSQ2 is affected by both the type of coupling graph and the input circuit. For instances that do not require many SWAP gates, e.g., QUEKO benchmarks, we can scale up to a thousand gates and tens of qubits. However, for those requiring many SWAP gates, e.g., QAOA circuits, TB-OLSQ2 cannot return a result within the 24-hour limit for circuits with more than 40 program qubits. In the future, we plan to explore different optimization techniques. One potential strategy is to search neighboring solutions based on the current satisfiable assignment. This strategy can be realized using an anytime MaxSAT solver as in [20]. In addition, we aim to support parallel layout synthesis by solving multiple instances simultaneously. Since each instance is independent of one another, we can build a portfolio of instances by generating configurations for a wide range of objective bounds. This could also include instances containing different encoding methods for cardinality constraints, as there does not appear to be a single best-in-class method with respect to solving time. Another possible direction is to help guide the SAT solving process using application-specific, heuristic algorithms. For example, our SAT solver currently uses a generic variable ordering for the search process, but we may be able to provide a better ordering based on our domain knowledge. In conclusion, although we were able to demonstrate significant advantages over several state-of-the-art layout synthesis tools in this paper, we believe that there is still further room for improvement for the modern layout synthesis tool and leave parallelization as one potential area for future exploration.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

TABLE II: Runtime comparison for using `AtMost` and CNF to encode cardinality constraints. Instances represent the layout synthesis problem for QAOA circuits on grid architectures with a SWAP count limit of 30. The depth limit is 21 for OLSQ and OLSQ2 and 5 for TB-OLSQ and TB-OLSQ2. An entry "TO" indicates that the case cannot be solved within the 24-hour time limit.

Grid	Qubit/ Gate	OLSQ		TB-OLSQ		OLSQ2(`AtMost`)		OLSQ2(CNF)		TB-OLSQ2(CNF)	
		Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio	Runtime (s)	Ratio
5×5	16/24	1221.84	1.00	123.19	9.92	404.98	3.02	65.03	18.79	2.59	471.75
	18/27	14665.26	1.00	90.47	162.10	944.71	15.52	703.20	20.86	2.73	5371.19
	20/30	5353.61	1.00	276.23	19.38	8308.12	0.644	1493.32	3.59	3.08	1738.19
	22/33	68833.65	1.00	1227.77	56.06	TO	–	19037.63	3.62	3.40	20245.19
	24/36	TO	–	827.73	–	78555.91	–	10776.14	–	3.79	–
	Avg.		1.00		61.87		6.40		11.71		6956.75

TABLE III: Depth optimization comparison between SABRE [11] and OLSQ2. The name of the benchmark circuit is followed by the number of program qubits and the gate number. For example, QFT(8/106) represents the QFT circuit with 8 qubits and 106 gates.

Device	Benchmark	SABRE	OLSQ2	Ratio
Sycamore	QFT(8/106)	120	70	1.71
	tof_4(7,55)	80	58	1.38
	barenco_tof_4(7,72)	115	73	1.58
	tof_5(9,75)	106	66	1.61
	barenco_tof_5(9,104)	120	66	1.82
	QAOA(16/24)	27	9	3.00
	QAOA(20/30)	34	10	3.40
	QAOA(24/36)	33	10	3.30
	QAOA(28/42)	49	11	4.45
	QUEKO(54/192)	87	5	17.40
	QUEKO(54/576)	208	15	13.87
	QUEKO(54/959)	423	25	16.92
	QUEKO(54/1342)	520	35	14.86
	QUEKO(54/1726)	789	45	17.53
Aspen-4	QUEKO(16/37)	43	5	8.60
	QUEKO(16/109)	73	15	4.87
	QUEKO(16/180)	112	25	4.48
	QUEKO(16/253)	157	35	4.49
	QUEKO(16/324)	193	45	4.29
Eagle	QAOA(16/24)	63	10	6.30
	QAOA(20/30)	56	14	4.00
			Avg.	6.66

TABLE IV: SWAP optimization comparison of SABRE [11], SATMap [20], and TB-OLSQ2. If the result has no SWAP gate, we count it as 1 when calculating the average ratio. An entry "OOM" indicates that the case cannot be solved due to the run-out-of-memory issue, and an entry "TO" indicates that the case cannot be solved within the timeout limit of 86400 seconds, or 24 hours.

Device	Benchmark	SABRE	SATMAP	TB-OLSQ2
Sycamore	QFT(8/106)	30	20	9
	tof_4(7,55)	24	1	1
	barenco_tof_4(7,72)	29	6	4
	tof_5(9,75)	17	1	1
	barenco_tof_5(9,104)	24	8	6
	ising_10(10,480)	33	9	0
	QAOA(16/24)	37	15	5
	QAOA(20/30)	59	OOM	7
	QAOA(24/36)	69	TO	13
	QAOA(28/42)	78	TO	18
	QUEKO(54/192)	111	TO	0
	QUEKO(54/576)	226	TO	0
	QUEKO(54/959)	416	TO	0
	QUEKO(54/1342)	460	TO	0
	QUEKO(54/1726)	649	TO	0
Aspen-4	QUEKO(16/37)	17	0	0
	QUEKO(16/109)	31	4	0
	QUEKO(16/180)	44	19	0
	QUEKO(16/253)	60	20	0
	QUEKO(16/324)	65	45	0
	Avg. Ratio	109.65	12.42	1.00

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, 1996, pp. 212–219.

[3] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019.

[4] J. Chow, O. Dial, and J. Gambetta, "IBM quantum breaks the 100-qubit processor barrier," *IBM Research Blog*, 2021.

[5] C. Monroe and J. Kim, "Scaling the ion trap quantum processor," *Science*, vol. 339, no. 6124, pp. 1164–1169, 2013.

[6] D. Schrader *et al.*, "Neutral atom quantum register," *Physical Review Letters*, vol. 93, no. 15, p. 150501, 2004.

[7] J. Clarke and F. K. Wilhelm, "Superconducting quantum bits," *Nature*, vol. 453, no. 7198, pp. 1031–1042, 2008.

[8] B. Tan and J. Cong, "Optimality study of existing quantum computing layout synthesis tools," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1363–1373, 2020.

[9] M. Y. Siraichi *et al.*, "Qubit allocation," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 113–125.

[10] A. Zulehner and R. Wille, "Compiling SU(4) quantum circuits to IBM QX architectures," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 185–190.

[11] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for NISQ-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1001–1014.

[12] A. Ho and D. Bacon, "Announcing Cirq: an open source framework for NISQ algorithms," *Google AI Blog*, vol. 18, 2018.

[13] A. Zulehner *et al.*, "An efficient methodology for mapping quantum circuits to the IBM QX architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1226–1236, 2018.

[14] S. Sivarajah *et al.*, "t|ket⟩: a retargetable compiler for NISQ devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014003, 2020.

[15] IBM. (2018) Qiskit. [Online]. Available: https://qiskit.org/

[16] R. Wille *et al.*, "Optimal SWAP gate insertion for nearest neighbor quantum circuits," in *2014 19th Asia and South Pacific Design Automation Conference*. IEEE, 2014, pp. 489–494.

[17] ——, "Mapping quantum circuits to IBM QX architectures using the minimal number of swap and H operations," in *2019 56th ACM/IEEE Design Automation Conference*. IEEE, 2019, pp. 1–6.

[18] D. Bhattacharjee *et al.*, "MUQUT: Multi-constraint quantum circuit mapping on NISQ computers," in *2019 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2019, pp. 1–7.

[19] G. Nannicini, L. S. Bishop, O. Günlük, and P. Jurcevic, "Optimal qubit assignment and routing via integer programming," *ACM Transactions on Quantum Computing*, vol. 4, no. 1, pp. 1–31, 2022.

[20] A. Molavi *et al.*, "Qubit mapping and routing via MaxSAT," in *2022 55th IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2022, pp. 1078–1091.

[21] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[22] B. Tan and J. Cong, "Optimal layout synthesis for quantum computing," in *2020 IEEE/ACM International Conference On Computer Aided Design*. IEEE, 2020, pp. 1–9.

[23] ——, "Optimal qubit mapping with simultaneous gate absorption iccad special session paper," in *2021 IEEE/ACM International Conference On Computer Aided Design*. IEEE, 2021, pp. 1–8.

[24] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[25] M. Amy *et al.*, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 818–830, 2013.

[26] H. Gorjiara *et al.*, "Satune: synthesizing efficient SAT encoders," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–32, 2020.

[27] M. Björk, "Successful SAT encoding techniques," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 4, pp. 189–201, 2011.

[28] N. Bjørner *et al.*, "Supercharging plant configurations using Z3," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2021, pp. 1–25.

[29] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[30] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ–an optimizing SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 194–199.

[31] J. Marques Silva and K. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 220–227.

[32] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2005, pp. 827–831.

[33] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26

[34] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[35] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab., Los Alamos, NM, US, Tech. Rep., 2008.

[36] Rigetti computing. [Online]. Available: https://www.rigetti.com