A Synthesis Framework for Stitching Surface Code with Superconducting Quantum Devices

Anbang Wu
University of California
Santa Barbara, USA
anbang@ucsb.edu

Gushu Li
University of California
Santa Barbara, USA
gushuli@ece.ucsb.edu

Hezi Zhang
University of California
Santa Barbara, USA
hezi@ucsb.edu

Gian Giacomo Guerreschi
Intel Labs
Santa Clara, USA
gian.giacomo.guerreschi@intel.com

Yufei Ding
University of California
Santa Barbara, USA
yufeiding@cs.ucsb.edu

Yuan Xie
University of California
Santa Barbara, USA
yuanxie@ece.ucsb.edu

ABSTRACT

Quantum error correction (QEC) is the central building block of fault-tolerant quantum computation but the design of QEC codes may not always match the underlying hardware. To tackle the discrepancy between the quantum hardware and QEC codes, we propose a synthesis framework that can implement and optimize the surface code onto superconducting quantum architectures. In particular, we divide the surface code synthesis into three key subroutines. The first two optimize the mapping of data qubits and ancillary qubits including syndrome qubits on the connectivity-constrained superconducting architecture, while the last subroutine optimizes the surface code execution by rescheduling syndrome measurements. Our experiments on mainstream superconducting architectures demonstrate the effectiveness of the proposed synthesis framework. Especially, the surface codes synthesized by the proposed automatic synthesis framework can achieve comparable or even better error correction capability than manually designed QEC codes.

CCS CONCEPTS

• **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**.

KEYWORDS

quantum computing, quantum error correction, compiler

ACM Reference Format:
Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, and Yuan Xie. 2022. A Synthesis Framework for Stitching Surface Code with Superconducting Quantum Devices. In *Proceedings of The 49th Annual International Symposium on Computer Architecture (ISCA '22).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527381

ISCA '22, June 18–22, 2022, New York City, NY

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527381

1 INTRODUCTION

Quantum hardware has made significant progress over the past decade, with the first demonstration of *quantum supremacy* in 2020 [2]. Among various quantum hardware technologies [3, 16, 26, 31], the superconducting (SC) qubit is currently one of the most promising candidates for building quantum processors [13, 38] due to its low error rate, single qubit addressability, manufacturing scalability, etc. Many of the latest quantum computers adopt SC technology, such as IBM's 65-qubit heavy-hexagon-architecture chip [49], Rigetti's 32-qubit octagonal-architecture device [21], and Google's 54-qubit square-architecture processor [2].

The low error rate of SC quantum processors makes them ideal platforms for quantum error correction (QEC) [5, 7, 19, 41, 43, 44], thus enabling fault-tolerant (FT) quantum computation. Among various QEC codes, the surface code [19] is a popular choice due to its high tolerance of physical error rates (up to 1%). This makes surface codes one of the most viable QEC options for demonstrating near-term FT quantum computation.

With off-the-shelf surface code arrays, many recent research efforts have been devoted to improving the efficiency of FT quantum computation, ranging from compilation [15, 39], communication scheduling [25, 27], to micro-controller design [46]. All these studies are based on a nontrivial assumption: we have found a scalable way to build logical qubits with the surface code family on existing quantum devices, in particular SC devices.

However, implementing surface codes on SC devices is complex in itself as error detection relies on sophisticated circuits. Surface codes divide physical qubits into *data qubits* and *syndrome qubits*, with syndrome qubits detecting the errors of neighboring data qubits through *measurement circuits* [19]. In surface code, the implementation of measurement circuits requires a qubit structure of 2D-lattice, with each qubit coupled to four neighbors [19]. Such an architecture is not readily available on many latest SC quantum processors [21, 49]. This is because dense architectures like the 2D-lattice would lead to a high physical error rate and low yield rate [35].

Previous works attempt to address the connectivity gap between surface codes and sparsely connected SC devices either by adapting architectures with tunable couplings [26] or by designing device-dedicated QEC codes [10]. Nevertheless, the former method is expensive and may introduce additional device noise, while the latter

method is not automated. A third attempt is to treat the measurement circuits of the surface code as ordinary quantum circuits and compile them to sparse SC architectures with existing quantum compilers [17, 34, 36, 42, 45, 47, 48, 50]. Unfortunately, generic compilers are not suitable for compiling surface codes. Firstly, they don't distinguish data qubits from other qubits. Those compilers may move data qubits frequently, making it hard to apply logical operations which assume a fixed data qubit layout [19]. Secondly, the SWAP gates they use to overcome the connectivity gap make the compiled measurement circuits more error-prone, compared to specialized measurement circuits [10, 32] which do not use SWAP gates. Finally, they only focus on gate-level optimization and do not account for the parallelism between measurement circuits enabled by a specific execution order [19].

To address problems of existing methods, we propose the first automatic synthesis framework *Surf-Stitch* which specializes in stitching the surface code family to various SC quantum devices. With specialized measurement circuits [10, 32] as the backend, our framework overcomes three key challenges of the surface code synthesis, which remain unexplored by existing works. The first is *the allocation of data qubits*. If the data qubits of a measurement circuit are far apart from each other, we would need many ancillary qubits to help detect their errors. Conversely, if they are too close, there will not be enough room for the syndrome qubit. The second is *the construction of measurement circuits*. Measurement circuits should be small as large circuits are error-prone and hurt the error detection accuracy of the surface code. Besides, large measurement circuits may contend for ancillary qubits. Such resource conflicts would destroy the parallelism of error detection. The third is *the execution order of measurement circuits*. We should exploit parallelism between measurement circuits as much as possible, to shorten the error detection cycle and reduce the decoherence error.

Our framework decouples the solution space of the identified key challenges with a modular optimization scheme that includes three stages. Firstly, we optimize the allocation of data qubits as they are the key to gluing measurement circuits together. We search for data qubits over rectangular device blocks since the measurement circuit is exactly shaped by a rectangle [19]. We require each rectangular block to be the smallest possible, for a compact data qubit layout and potential small measurement circuits. Secondly, we optimize measurement circuits for the allocated data qubits. The goal is to keep them small and minimize the conflict between them if possible. To achieve the goals, we constrict each measurement circuit within rectangular blocks of zero overlapping areas and then adopt two heuristics to find small circuits. Finally, we optimize the execution order of measurement circuits as the conflict between them is sometimes inevitable. We observe that the error detection cycle can be reduced by executing large measurement circuits together. Therefore we propose a procedure to find and execute large circuits that do not have resource conflicts in parallel.

We evaluate the proposed synthesis framework by comparing it with manually-designed QEC codes [10]. The results show that the surface codes synthesized by our framework can achieve equivalent or even better error correction capability. This result is inspiring as it unveils the possibility that automated synthesis can surpass manual QEC code design by experienced theorists. We also investigate our framework on various mainstream SC quantum architectures to demonstrate its wide applicability. Surf-Stitch would be of great interest to both QEC researchers and quantum hardware designers. Theorists will have a baseline to compare with when designing novel QEC codes. Hardware researchers can identify inefficient architecture designs for the surface code with Surf-Stitch.

Our contributions in this paper are summarized as follows:

- We systematically formulate the surface code synthesis problem on SC quantum devices for the first time and identify three key challenges: data qubit allocation, measurement circuit construction, and syndrome measurement schedule.
- We propose the first automatic synthesis framework that addresses the identified challenges step by step, with insights extracted from surface codes and SC quantum architectures.
- Our evaluation demonstrates the effectiveness of the proposed framework by the comparison to manually designed QEC codes and a comprehensive investigation of Surf-Stitch on various mainstream SC quantum architectures.

2 BACKGROUND

In this section, we introduce key concepts for understanding the implementation requirements of surface codes [4, 5, 14]. We do not cover the basics of quantum computing but recommend [37] for reference.

2.1 Surface code basics

Quantum computation is fragile without error correction. Information in qubits can be easily distorted by the decoherence error [37]. The imprecise quantum operation and erroneous quantum measurement further worsen the situation [37]. To ensure fault-tolerant quantum computation, various QEC codes [5, 7, 19, 41, 43, 44] are proposed. In these QEC codes, the surface code is among the most popular ones due to its excellent error correction capability [19]. We introduce the basics of surface codes as follows.

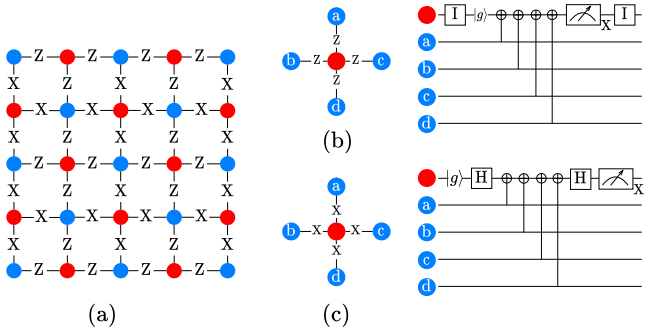


Figure 1: Common components of the surface code. (a) The surface code lattice with data qubits (blue dots) and syndrome qubits (red dots). (b) Z-type syndrome extraction and its circuit. (c) X-type syndrome extraction and its circuit.

Data and syndrome qubits: The surface code encodes a logical qubit in a 2D lattice of physical qubits, as shown in Figure 1(a). The physical qubits in the code lattice can be divided into two types: data qubits and syndrome qubits, denoted as blue and red dots, respectively in Figure 1(a). The encoded logical information is stored

in data qubits. Error information on data qubits can be extracted by measuring the syndrome qubit (a.k.a measurement qubit). Each syndrome qubit is coupled with its (up to) four neighboring data qubits, using the syndrome extraction circuit (a.k.a measurement circuit) shown in Figure 1(b)(c) to gather the error information on data qubits.

Pauli operator and stabilizer: In surface codes, the relationship between a syndrome qubit and its neighboring data qubits is represented by the product of Pauli operators (a.k.a Pauli string [37]), as shown in Figure 1(a) where Pauli operators are labeled on the edges between data qubits and syndrome qubits. For each syndrome qubit, the Pauli string on its edges can be in one of two possible patterns. The first one (Z-type) is shown in Figure 1(b). The connections between the center syndrome qubit and the four data qubits are all labeled by the operator Z, and together they are represented by the Pauli string $Z_aZ_bZ_cZ_d$. The second one (X-type) as shown in Figure 1(c) is similar, except that all connections are labeled by the operator X, together represented by the Pauli string $X_aX_bX_cX_d$.

For these two different patterns, we would have corresponding syndrome extraction circuits to detect errors on data qubits (shown on the right side of Figure 1(b)(c)). Syndrome extraction circuits in Figure 1(b)(c) project the state of data qubits $\{a, b, c, d\}$ onto the eigenstates of corresponding Pauli strings, which are also referenced by *stabilizers* [22] in the context of QEC. Syndrome extraction is thus known as the *stabilizer measurement* [6]. Without ambiguity, we use the stabilizer notation to represent the syndrome extraction circuit. We denote the stabilizer $Z_aZ_bZ_cZ_d$ ($X_aX_bX_cX_d$) with Z_{abcd} (X_{abcd}) for simplicity.

Error detection: Surface codes can detect Pauli X- and Z-errors on data qubits with Z- and X-type stabilizer measurement circuits, respectively. Errors on a data qubit can affect the measurement results of stabilizers associated with it. In an *error detection cycle*, the surface code would run all stabilizer measurements once and collect the measurement results. With these results, a surface code error correction protocol can infer what errors have occurred in the code lattice and apply corrections accordingly. For more details, please refer to [19].

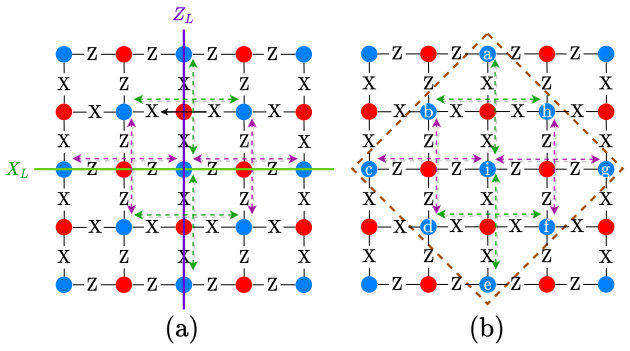


Figure 2: Code distance and the compact lattice. (a) Logical operations of the distance-3 surface code. (b) Inside the rotated rectangle (dashed brown line) is a compact surface code lattice with the same code distance as in (a).

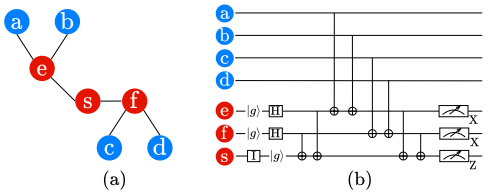


Figure 3: Z-type stabilizer measurement circuit synthesis. (a) The connected graph of data qubits (blue) and ancillary qubits (red). Qubit s is the syndrome qubit. (b) The synthesized stabilizer measurement circuit that is executable on the connected graph in (a).

Code distance: The error correction capability of the surface code is related to the code distance [7, 30], which is defined as the minimum number of physical qubits that support the logical X or Z operation on the encoded logical qubit (denoted by X_L or Z_L in Figure 2). Usually, surface codes with larger code distances can correct more complex errors, but their implementation overhead is also higher. Figure 2(a) shows the logical operations in a distance-3 surface code. Figure 2(b) indicates that a more compact surface code lattice can be obtained without changing the code distance. In this paper, we focus on the rotated surface code [24] in Figure 2(b).

2.2 SC hardware and stabilizer measurement

The low error rates and fast operation speed of SC qubits make it an ideal platform for implementing surface codes that require high-fidelity quantum operations and short stabilizer measurement latency to detect data qubit errors in time. However, the average node (qubit) degree of the SC quantum device is usually kept low (< 3) to reduce frequency collisions and crosstalk errors in the device [10]. This means many SC processors have limited connectivity between qubits and we may not have enough four-degree qubits to synthesize the surface code lattice in Figure 2(b).

The first step of the surface code synthesis is to implement each stabilizer measurement. To map a single stabilizer measurement circuit to the sparsely-connected SC device, various methods have been proposed, such as the degree-deduction technique [10], and the flag-bridge circuit [8, 9, 12, 32]. Once data qubits and the required ancillary qubits (including the syndrome qubit) for the stabilizer measurement are specified, those methods can generate corresponding measurement circuits executable on SC devices. Figure 3 shows the generated flag-bridge circuit for the stabilizer Z_{abcd} with a tree of ancillary qubits $\{e, s, f\}$. These ancillary qubits are also called *bridge qubits*, and the tree they form is called the *bridge tree*. Here we set qubit s as the tree root, which acts as the syndrome qubit for Z_{abcd}, collecting error information from data qubits $\{a, b, c, d\}$. The construction of the flag-bridge circuit consists of five components:

1) *Initialization:* bridge qubits except the tree root s are initialized to $|+\rangle$ while s is initialized to $|0\rangle$.

2) *Encoding circuit:* Starting from $k = 1$ (i.e., from the root s), for each node at the k-th level of the bridge tree, we add one CNOT gate from this node to its parent node in the encoding circuit.

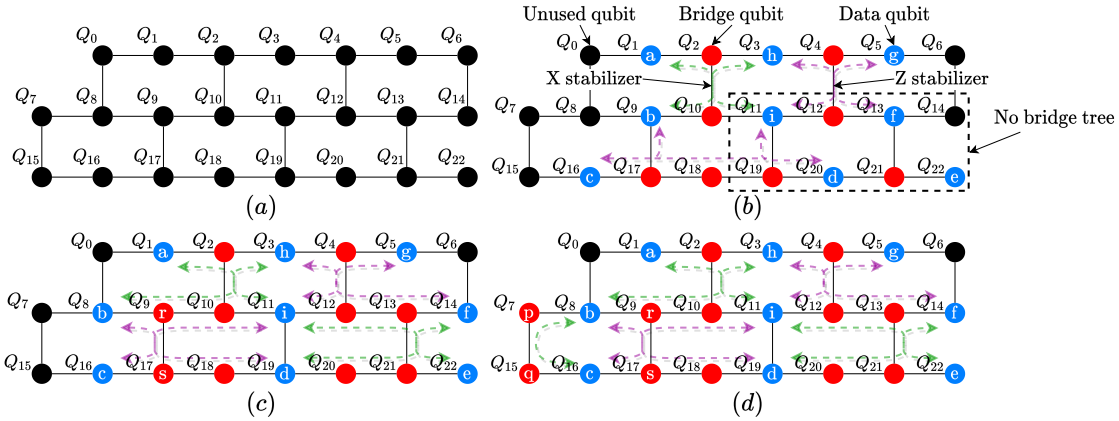


Figure 4: A motivating example for synthesizing a (rotated) distance-3 surface code. (a) An SC device based on the hexagon structure. (b) A bad data qubit layout where the stabilizer X_{idfe} cannot be measured. (c) A promising data qubit layout that ensures all stabilizer measurements. (d) An example of resolving the bridge tree conflict.

3) *Coupling circuit for data qubits:* We couple data qubits in a zigzag way with leaf bridge qubits instead of the tree root to respect the device connectivity limitation.

4) *Decoding circuit:* The decoding circuit is the mirror of the encoding circuit. Those two circuits together ensure the fault tolerance of the stabilizer measurement.

5) *Measurements:* The tree root is measured on the Z basis while other ancillary qubits are measured on the X basis. The measurement results on the X basis can be used to detect Pauli Z errors on ancillary qubits.

Other stabilizer measurement circuits can be constructed in a similar way. For more details, please refer to Lao et al. [32].

Methods discussed in this section only solve the low-level circuit generation problem of one stabilizer measurement, far from tackling the overall surface code synthesis, where we need to answer the following questions: *how* to allocate data qubits of the surface code on physical devices? *how* to decide the ancillary qubits for each stabilizer measurement? *how* to determine the order of stabilizer measurements when some measurement circuits need to be executed sequentially?

Our framework aims exactly at the overall surface code synthesis and uses the low-level measurement circuit synthesis methods in this section as backends.

3 PROBLEM FORMULATION

In this section, we first formulate the problem of synthesizing surface codes onto SC quantum processors and then introduce the optimization opportunities.

We consider implementing the (rotated) surface code in Figure 2(b) on a quantum device with the hexagon architecture (Figure 4(a)) [10]. In this hexagon device, each qubit connects to at most three other qubits. This imposes a challenge to synthesizing stabilizer measurement circuits of the surface code since a syndrome qubit in either an X- or Z- type stabilizer measurement should connect to four data qubits (see Figure 1(b)(c)). As in Figure 3, we can overcome the connectivity limitation of this SC device with specialized measurement circuits [10, 32] as long as the data qubits and the bridge tree for each stabilizer measurement are determined. While deciding the data qubits and the bridge tree for one stabilizer measurement is easy, deciding them for all stabilizer measurements and making the implemented measurement circuits work together are challenging tasks in the overall surface code synthesis.

In this section, we formulate the surface code synthesis problem into three key stages: data qubit allocation, measurement circuit construction, and stabilizer measurement schedule. Since the measurement circuit is determined once the bridge tree is selected, we would refer to the second stage as bridge tree construction in the rest of the paper.

We briefly introduce the objectives and the design considerations of each stage as follows.

Data qubit allocation: We choose to allocate and fix the position of data qubits first as data qubits are the key to gluing stabilizer measurement circuits together. Once allocated, the location of data qubits should not be changed, otherwise, the logical operations designed for a fixed data qubit layout [19] would be invalidated. The layout of data qubits affects the execution of stabilizer measurements. As an example, we synthesize the distance-3 surface code in Figure 2(b) with two data qubit layouts in Figure 4(b) and Figure 4(c). In Figure 4(b), the stabilizer X_{idfe} cannot be measured without moving data qubits and inserting SWAP gates, which are not allowed to avoid error proliferation. In contrast, all stabilizer measurements $(X_{abhi}, X_{idfe}, X_{fg}, X_{bc}, Z_{bcid}, Z_{higf}, Z_{ah}, Z_{de})$ can be executed on Figure 4(c) by using the depicted bridge trees.

Bridge tree construction: After the data qubits are placed, the next step is to select bridge qubits and construct bridge trees for stabilizer measurements. The first constraint in this stage is that we should minimize the number of bridge qubits for each stabilizer measurement since using more physical qubits would result in larger measurement circuits which are naturally more error-prone. Besides, the construction of bridge trees affects the efficiency of error detection because two stabilizers can be simultaneously measured only if their bridge trees do not share qubits (i.e., no resource conflict). For instance, referring to Figure 4(c), if we measure X_{bc}

with bridge qubits $\{r, s\}$, these two qubits then cannot be used in the measurement circuit of X_{abhi} at the same time because the bridge qubits need to be reset at the beginning of any measurement circuit. However, if we measure X_{bc} with bridge qubits $\{p, q\}$ in Figure 4(d), we can measure X_{bc} and X_{abhi} in parallel. An efficient bridge tree construction should enable the concurrent measurement of as many stabilizers as possible.

Stabilizer measurement scheduling: The third stage is to schedule the execution of stabilizer measurements. It would be desirable to execute stabilizer measurements in parallel as much as possible since it can reduce the error detection latency and mitigate the decoherence error. However, stabilizer measurement circuits with overlapped bridge qubits cannot be executed simultaneously. For example in Figure 4(c), the measurement circuit of X_{abhi} and Z_{bcid} cannot be measured together since they share bridge qubits $\{q_9, q_{10}\}$. One possibility is to measure X_{abhi} and X_{idhe} first, then measures Z_{hgif} and Z_{bcid}. This schedule may seem promising, but it is not optimal as these two groups of stabilizer measurements take 20 operation steps in total, using the flag-bridge circuit [32](see Figure 3) as the backend. As a comparison, if we measure X_{abhi} and Z_{hgif} first and measure X_{idfe} and Z_{bcid} second, the total number of operation steps is only 18. Our objective is to identify the potential parallelism in stabilizer measurements and generate efficient scheduling to shorten the overall error detection latency.

4 SYNTHESIS ALGORITHM DESIGN

In this section, we introduce the surface code synthesis flow of Surf-Stitch. As discussed above, we will introduce three key stages: data qubit allocation, bridge tree construction and stabilizer measurement scheduling.

4.1 Data qubit allocator

We start by allocating data qubits. Once allocated, the positions of data qubits should not be changed. This is because the logical operations on surface codes [19] assume a fixed data qubit layout, and moving data qubits would invalidate those high-level operations. Moreover, moving data qubits would involve a series of SWAP gates which are noisy and could destroy the logical information stored in data qubits.

A fundamental requirement for data qubit allocation is to ensure that a bridge tree exists for each stabilizer. The following proposition about the degree of nodes (qubits) provides a necessary condition to guarantee this.

Proposition 1. *Any bridge tree for a stabilizer with support on four data qubits must have at least one four-degree node or two three-degree nodes.*

Proof. For any graph $G(V, E)$ where V is the vertex set and E is the edge set, we have $\sum_{v \in V} deg(v) = 2|E|$. An n-vertex tree always has $n-1$ edges. A bridge tree with four data qubits has four 1-degree leaf nodes and all other nodes should have degrees of at least 2. Therefore we have $4 + \sum_{v \in V \setminus \text{data qubits}} deg(v) = 2n - 2$ and $\sum_{v \in V \setminus \text{data qubits}} deg(v) = 2n - 6 = 2(n-4) + 2$. We only have $n-4$ vertices after removing the four leaf nodes. So we must have at least one four-degree node or two three-degree nodes. □

Algorithm 1: Data qubit allocation

Input: Device architecture graph G.
Output: Data qubit layout *data_layout*.

L_h = all three- and four-degree nodes in G;
bridge_rects = [] ; `// the set of bridge rectangles`
for n_a *in* L_h **do**
  if $deg(n_a) == 3$ **then**
    n_b = the nearest high-degree node of n_a;
    rect = the minimal rectangle containing n_a, n_b and their neighboring qubits;
  else
    rect = the minimal rectangle containing n_a and its neighboring qubits;
  end
  bridge_rects.append(*rect*);
end
r_0 = the bridge rectangle at the top left corner of G;
bridge_rect_tuple = []; `// compatible bridge rects;`
repeat
  for $(r_1, r_2, r_3) \in \otimes^3 bridge\_rects$ **do**
    if r_0, r_1, r_2, r_3 *are mutually compatible* **then**
      potent_dqbits = qubits enclosed by r_0, r_1, r_2, r_3; `// potential data area;`
      if *potent_dqbits* $\neq \emptyset$ **then**
        bridge_rect_tuple.append((r_0, r_1, r_2, r_3));
        break;
  end
  set r_0 to r_1, r_2, r_3 in turn to find new tuples of r_0, r_1, r_2, r_3 that has non-empty *potent_dqbits*;
until *bridge_rect_tuple converges*;
data_layout = [];
for r_0, r_1, r_2, r_3 *in bridge_rect_tuple* **do**
  dqb = the qubit at the center of *potent_dqbits* of r_0, r_1, r_2, r_3;
  data_layout.append(*dqb*);
end

From Proposition 1, we see that a feasible layout should ensure that each data qubit has enough three-degree or four-degree qubits around it so that it can form a stabilizer with nearby data qubits. To achieve that, we introduce a data qubit layout that ensures the existence of *local bridge trees*, whose bridge qubits lie within the region bounded by the corresponding stabilizer's data qubits. The benefit of this strategy comes from the fact that local bridge trees often lead to shallow measurement circuits. For illustration, we adopt the SC quantum architecture shown in Figure 5(a). We embed the coupling graph of that architecture into a 2D grid so that all qubits can be referred to by the spatial coordinates on the grid. Such an embedding is always possible for the latest SC processors as they are usually designed in a modular structure.

Now we state the data qubit allocation algorithm, as shown in Algorithm 1. We keep a list (denoted as L_h) for all the three- and four-degree nodes in the grid and record their coordinates since

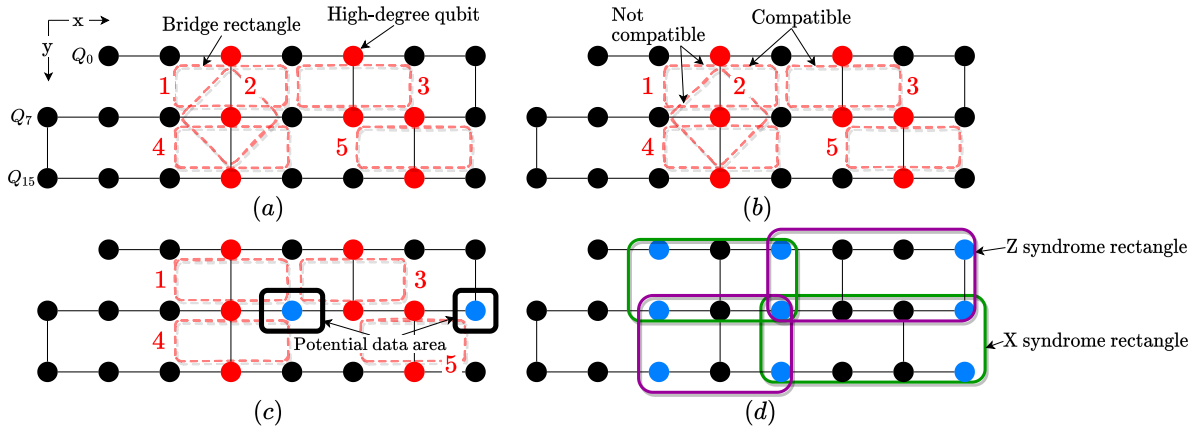


Figure 5: Data qubit allocation example. (a) A modified device from Figure 4(a). Red circles indicate physical qubits with a high degree of connectivity (i.e. with 3 or more edges). (b) Finding compatible bridge rectangles. (c) Locating data qubits. (d) The final data qubit layout and syndrome rectangles.

high-degree nodes are critical to constructing the data qubit layout. In Figure 5(a), $L_h = \{Q_2, Q_4, Q_{10}, Q_{12}, Q_{13}, Q_{18}, Q_{21}\}$. Then we process the nodes in the list sequentially. For each node n_a in L_h, if it is a three-degree node, we search for its nearest high-degree node n_b and create a minimal rectangle containing n_a, n_b, and their neighbors. If n_a is of degree ≥ 4, then we create a rectangle containing n_a and its neighbors. Such a rectangle is called a *bridge rectangle*. Figure 5(a) depicts five bridge rectangles resulted from $\{Q_2, Q_{10}\}$, $\{Q_{10}\}$, $\{Q_4, Q_{12}\}$, $\{Q_{13}, Q_{21}\}$ and $\{Q_{18}, Q_{10}\}$, indexed from 1 to 5. We omit other bridge rectangles here for simplicity.

Based on those bridge rectangles, we can then determine the positions of data qubits. As shown in Figure 1(a), each data qubit of the surface code should be shared by four stabilizers. Likewise, we can determine the position of a data qubit by four bridge rectangles. We search for *compatible* bridge rectangles starting from rectangle 1. (We can also start from rectangle 2 which is created from a four-degree qubit. We will discuss this possibility in Section 5.) Two bridge rectangles are said to be compatible if their overlapping area is zero. For example in Figure 5(b), rectangle 2 is not compatible with rectangle 1 and rectangle 4, while rectangles 1, 3, 4, and 5 are mutually compatible. We avoid using incompatible rectangles as they may not allow a feasible data qubit layout. When four compatible bridge rectangles are found, we search for the data qubit in the potential data area (the black rectangle in the center of Figure 5(c)), which is enclosed by those compatible bridge rectangles, as shown in Figure 5(c). If the potential data area is empty, we select another four compatible bridge rectangles. Otherwise, we select the qubit at the center of the potential data area as a data qubit.

On the boundary, we may not have enough bridge rectangles to locate the data qubits. For example, the bottom right corner of rectangle 3 is only neighbored by rectangle 5. In this case, we have to locate the data qubit based on only those two bridge rectangles. Specifically, a potential data qubit should satisfy: A) its x coordinate $\geq$ the largest x coordinate in rectangles 3 and 5; B) its y coordinate should lie between the largest y coordinate of rectangle 3 and the smallest y coordinate of rectangle 5. With these spatial constraints, the only qubit we can find is Q_{14}, as shown in the black rectangle on the right of Figure 5(c). Positions of other data qubits are determined in a similar way.

The final layout of data qubits and their associated syndrome rectangles are shown in Figure 5(d). A syndrome rectangle is an extension of the bridge rectangle which includes the allocated data qubits. We can assign a stabilizer to each syndrome rectangle and synthesize the corresponding measurement circuits locally (using qubits inside each syndrome rectangle). In the next section, we will discuss how to find a short bridge tree for each stabilizer.

4.2 Bridge tree finder

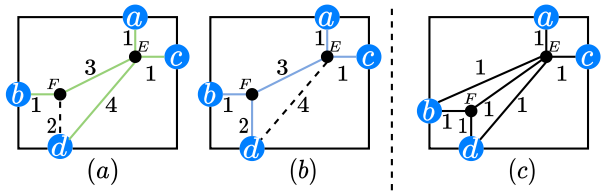


Figure 6: Finding bridge trees in a syndrome rectangle with data qubits $\{a, b, c, d\}$. (a)(b) shows the case where path merge is efficient, while (c) shows the case when path merging incurs extra overhead. (a) Green edges denote the shortest paths from qubit E to data qubits and they form a bridge tree with length 10. (b) Blue edges form a bridge tree with length 8. (c) An example where data qubits are close to each other.

Based on allocated data qubits and syndrome rectangles, we search for bridge trees that satisfy two requirements: small in size and local in position. The reason why they are required to be small is that large bridge trees would compromise the fidelity of stabilizer measurements. This is because the error correction capability of the synthesized surface codes is sensitive to the length of bridge trees, as each additional edge in a bridge tree results in two more

Algorithm 2: Bridge tree construction

Input: A syndrome rectangle R with data qubits $\{a, b, c, d\}$.
Output: Candidate bridge trees.
// bridge trees by the star-tree method;
$star\_trees$ = [];
// bridge trees by the branching-tree method;
$branching\_trees$ = [];
for qb *in* R **do**
  T = the bridge tree by connecting qubit qb to data qubits $\{a, b, c, d\}$ with shortest paths;
  insert T to $star\_trees$ and remove trees larger than T from $star\_trees$;
end
let $\{a', b', c', d'\}$ be an arrangement of $\{a, b, c, d\}$ s.t. $l_{a'b'} + l_{c'd'} = \min\{l_{ab} + l_{cd}, l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$;
// l_{ab} is the distance of $a \to b$;
connect a' and b', c' and d' with shortest paths;
for qb_1 *in* $a' \to b'$, qb_2 *in* $c' \to d'$ **do**
  T = the resulting bridge tree by connecting qb_1 and qb_2 with shortest paths;
  insert T to $branching\_trees$ and remove trees larger than T from $branching\_trees$;
end
merge $star\_trees$ and $branching\_trees$ to find a list of small local bridge trees;

CNOT gates in the measurement circuit, increasing the probability of correlated errors which are hard to detect and correct [19]. The reason why bridge trees should be local is to guarantee the parallelism of bridge trees, which also affects the fidelity of stabilizer measurements. For bridge trees that share bridge qubits, i.e., incompatible bridge trees, their corresponding stabilizers must be measured sequentially, resulting in a longer error detection cycle, which means more decoherence errors. To reduce potential conflicts between bridge trees, we only search for bridge trees inside each syndrome rectangle. Such *local bridge trees*, whose qubits lie completely within the syndrome rectangles, naturally facilitate the concurrent measurement of stabilizers.

A natural way to find small local bridge trees is to first locate the bridge tree root within the syndrome rectangle, and then connect the tree root to data qubits by the shortest paths. We denote this method as the *star-tree method*. A disadvantage of this method is that it may miss opportunities for path merging. For example, in the syndrome rectangle in Figure 6(a), the length of the bridge tree produced by the star-tree method is 10 (green edges). In contrast, by merging paths $E \to F \to b$ and $E \to d$, we can get a bridge tree of length 8 (blue edges in Figure 6(b)), which reduces the number of CNOT gates in the resulting stabilizer measurement circuit by at least 4.

To remedy the above shortcoming, we propose the *branching-tree method*, which first connects close data qubit pairs by shortest paths, and then connects those shortest paths to build a complete bridge tree. As an example, suppose we are constructing a bridge tree for the syndrome rectangle in Figure 6(a). We first find the shortest paths $a \to c$ and $b \to d$, since $l_{ac} + l_{bd}$ (l_{ac} is the length of the shortest path from a to c) is smaller than $l_{ab} + l_{cd}$ and $l_{ad} + l_{bc}$. Then by connecting paths $a \to c$ and $b \to d$ with path $E \to F$, we immediately obtain the small bridge tree (blue edges) in Figure 6(b). The following proposition bounds the length of the bridge tree generated by the branching-tree method:

Proposition 2. *Let the total edge length of the bridge tree T generated by the branching-tree method be $E(T)$, then,*

$$E(T) \le \frac{1}{2}(l_{ab} + l_{ac} + l_{ad} + l_{bc} + l_{bd} + l_{cd}).$$

Proof. W.l.o.g., we assume $l_{ab} + l_{cd} \le \min\{l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$. Then in T, we first connect a and b, c and d, respectively. On the other hand, the distance between shortest paths $a \to b$ and $c \to d$ is smaller than $\min\{l_{ac}, l_{ad}, l_{bc}, l_{bd}\}$. This proposition then can be proved by combining these two inequalities. □

Generally, the branching-tree method is more efficient if $\min\{l_{ab}+l_{cd}, l_{ac}+l_{bd}, l_{ad}+l_{bc}\}$ is small, as shown in Figure 6(a)(b). In this case, the length of the resulting bridge tree is very close to $\frac{1}{2}(l_{ad} + l_{bc})$. In contrast, the length of the bridge tree by the star-tree method is at least $\max\{l_{ad}, l_{bc}\}+2$, which is larger than that by the branching-tree method. However, if $\max\{l_{ab} + l_{cd}, l_{ac} + l_{bd}, l_{ad} + l_{bc}\}$ is small too, the benefit of path merging may not outweigh the overhead of not using shortest paths. Figure 6(c) shows an example where the star-tree method produces a shorter bridge tree. In practice, we will run both the star-tree method and the branching-tree method, then find small bridge trees by merging their results, as shown in Algorithm 2. Once the bridge tree is determined, we can assign the syndrome qubit to the center node of the bridge tree.

In general, our bridge tree finder can generate small bridge trees that approximate optimal bridge trees as long as the distances between data qubits are small (by Proposition 2).

4.3 Stabilizer measurement scheduler

With all stabilizers and their measurement circuits allocated to the physical device, the next goal is to minimize the runtime of stabilizer measurements by maximizing parallelism, which naturally reduces the effect of decoherence. Two stabilizers can be measured in parallel if and only if their measurement circuits do not share bridge qubits. Such stabilizers are said to be compatible with each other. To exploit the parallelism of compatible stabilizers while not allowing incompatible stabilizers to be measured concurrently, we propose a heuristic scheduling approach in Algorithm 3, which consists of two steps: schedule initialization and refinement loop.

Schedule initialization: The proposed data qubit allocation ensures that syndrome rectangles of the same type do not have bridge tree conflicts, i.e., the stabilizer measurements of the same type are compatible with each other. With this guarantee, we initialize the stabilizer measurement schedule with two sets, S_1 and S_2 which contain X- and Z-type stabilizers, respectively.

Refinement loop: The core idea of the refinement loop is to move stabilizers with large measurement circuits into one set. The motivation for this refinement is that the execution time of a set of stabilizer measurements is determined by the stabilizer with the deepest measurement circuit. After the refinement loop, we end up

Algorithm 3: Iterative stabilizer measurement scheduling

Input: Binary tuples of stabilizer and syndrome rectangle: $\{(s, R)\}$.
Output: A schedule of binary tuples of stabilizer and bridge trees.

```
// Schedule initialization;
S_1 = tuples of X-stabilizers and syndrome rectangles;
S_2 = tuples of Z-stabilizers and syndrome rectangles;
// Iterative refinement;
repeat
    if exec_time(S_1) < exec_time(S_2) then
        swap(S_1, S_2);
    r_2 = (s, R) in S_2 that has longest execution time;
    swap_list = [r_2]; for i in [0 : k] do
        S = S_{i%2+1};
        for r in swap_list do
            swap_list.remove(r);
            for r_1 in S in descending order do
                if r_1 and r does not have compatible bridge trees then
                    if exec_time(r_1) > exec_time(r) then
                        terminate the refinement loop;
                    swap_list.append(r_1);
                    S.remove(r_1);
            end
            if swap_list == ∅ then
                break;
        end
    end
    if swap_list ≠ ∅ then
        recover S_1 and S_2 to the values before this iteration;
        break;
until S_1 converges;
generate the finalized stabilizer measurement schedule from S_1 and S_2;
```

with one set containing the stabilizers with large measurement circuits, and the other set containing the remaining stabilizers which have small measurement circuits and can be measured in a short time.

To illustrate how the refinement loop works, suppose we are given stabilizers and syndrome rectangles shown in Figure 7. Initially, we have $S_1 = \{(s_1, R_1), (s_4, R_4), (s_5, R_5)\}$ and $S_2 = \{(s_2, R_2), (s_3, R_3), (s_6, R_6)\}$. We then send the largest element in S_2, which is (s_2, R_2) in this case, to the *swap_list* and swap it into S_1. Since (s_4, R_4) and (s_2, R_2) do not have compatible bridge trees, we will move (s_4, R_4) to S_2. In S_2, (s_6, R_6) is not compatible with (s_4, R_4), so it will be swapped into S_1. After this swap, the refinement loop will stop since the *swap_list* is empty and every stabilizer in S_1 has a larger bridge tree than the stabilizer in S_2. The finalized stabilizer measurement schedule is shown in Figure 7(b). Compared to the initial schedule, the refined schedule in Figure 7(b) reduces the error detection cycle by one time step, and reduces the CNOT gate number by two.

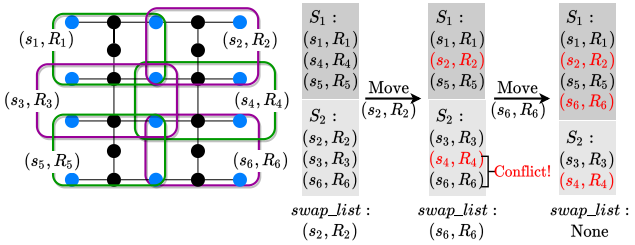


Figure 7: An example of stabilizer measurement scheduling.

5 EVALUATION

In this section, we first evaluate the proposed synthesis framework *Surf-Stitch* by comparing its generated surface codes with state-of-the-art manually designed QEC codes. We then demonstrate the effectiveness of Surf-Stitch on mainstream SC architectures by analyzing the error correction capability and resource overhead of the synthesized codes.

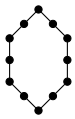
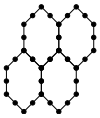
5.1 Experiment Setup

Evaluation setting: We use the flag-bridge circuit [32] as the backend for instantiating stabilizer measurement circuits as it provides the extra feature of fault-tolerant error detection [11]. We implement all numerical simulations with stim v1.5.0, a fast stabilizer circuit simulator [20]. We use PyMatching v0.4.0 [23] for error decoding with measurement signals from bridge qubits. The PyMatching decoder is the implementation of the well-studied *Minimum Weight Perfect Matching* (MWPM) algorithm [10, 19]. Error rates are computed by performing 10^5 simulations with $3d$ (d is the code distance) error detection rounds, on a Ubuntu 18.04 server with a 6-core Intel E5-2603v4 CPU and 32GB RAM.

Metrics: We evaluate the *error threshold* of the synthesized surface codes to demonstrate their error correction capability. Error threshold indicates the error rate below which hardware errors can be tolerated [19]. Hence, a higher error threshold is preferred. The time-step count determines the execution speed of the surface code and its logical operations. A large time-step count would also introduce more decoherence errors. Thus, a small time-step count is preferred. Finally, We evaluate the resource overhead of the synthesized surface codes with the *CNOT count* and the *qubit count*. A resource-efficient synthesis should use fewer CNOT gates and bridge qubits.

Device architectures: we use two categories of device architectures, as shown in Table 1. The first category of architectures built from tiled polygons is the basic structure of many SC quantum devices, e.g. Google's Sycamore [2] and IBM's latest machines [28]. The second category of architectures is mainly used by IBM devices [28]. It consists of *heavy architectures* with an extra qubit inserted into each edge of the polygons. Edges with the extra qubit in the middle are called *heavy edges*. Compared to polygon architectures, the average qubit connectivity of heavy architectures is

Table 1: Overview of device architectures.

Type	Name	Building blocks	Tilling Example	Remark
Polygon Architectures	Square			Each square can have at most four neighboring squares for tiling.
	Hexagon			Each hexagon can have at most six neighboring hexagons for tiling.
	Octagon			Each octagon can have at most four neighboring octagons for tiling.
Heavy Architectures	Heavy Square			Heavy squares are tiled like squares.
	Heavy Hexagon			Heavy hexagons are tiled like hexagons.

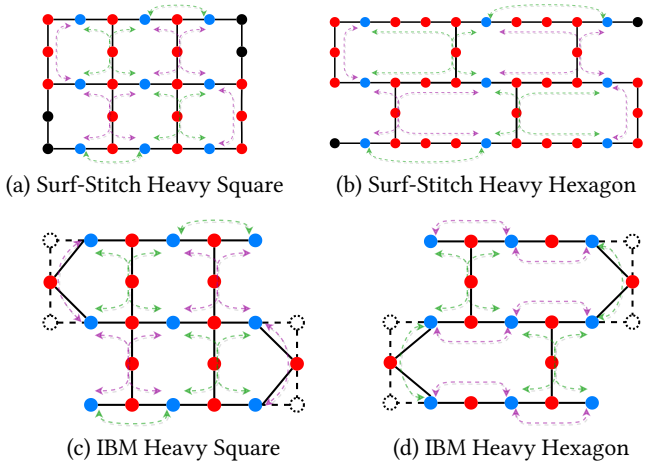
lower due to the inserted two-degree qubits. All architectures in Table 1 can be easily embedded into a 2D grid.

Error model: In all simulations, we assume a similar circuit-level error model as in [10, 19]. For the gate error, we assume an error probability p_e for the single-qubit depolarizing error channel on single-qubit gates, the two-qubit depolarizing error channel on two-qubit gates, and the Pauli-X error channel on measurement and reset operations. For the idle error induced by decoherence, we assume each idle qubit is followed by a single-qubit depolarizing error channel per gate duration with the error probability 0.0002, which is estimated by the decoherence error formula $1 - e^{-\frac{t}{T}} \approx 0.0002$, with the gate duration $t = 20\,ns$ and the relaxation or dephasing time $T = 100\,\mu s$ [1]. These errors happen on all qubits, including data qubits and bridge qubits.

5.2 Compared to manually designed QEC codes

We first compare the synthesized surface codes by Surf-Stitch to the two manually designed QEC codes by Chamberland et al. [10] on heavy architectures. Figure 8(a)(b) show the qubit layouts and stabilizer measurements of our synthesized surface codes on the heavy square architecture ('Surf-Stitch Heavy Square') and the heavy hexagon architecture ('Surf-Stitch Heavy Hexagon'). Figure 8(c)(d) show the manually designed QEC codes on the heavy square architecture ('IBM Heavy Square') and the heavy hexagon architecture ('IBM Heavy Hexagon'). The error thresholds of these codes are in Figure 9. The error thresholds are computed with respect to Pauli X errors.

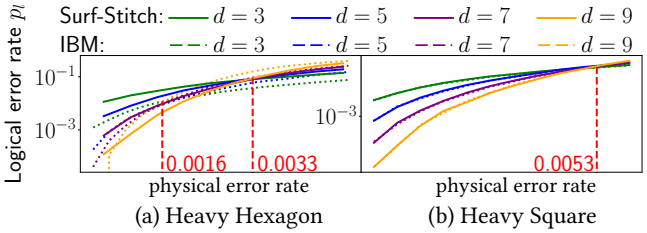
Overall, compared with the manually designed codes on the heavy architectures, the surface codes synthesized by Surf-Stitch can have comparable or even better error correction capability. On



(a) Surf-Stitch Heavy Square (b) Surf-Stitch Heavy Hexagon

(c) IBM Heavy Square (d) IBM Heavy Hexagon

Figure 8: The synthesized distance-3 surface codes by Surf-Stitch and the two manually designed QEC codes by IBM [10].



(a) Heavy Hexagon (b) Heavy Square

Figure 9: The error threshold is the physical error rate where code curves of different distances meet. (a) The error thresholds are 0.16% and 0.33% for codes by IBM and Surf-Stitch, respectively. (b) The error threshold is 0.53% for both codes.

the heavy hexagon architecture, the error threshold of 'Surf-Stitch Heavy Hexagon' is 0.33% which is 106% higher than that of 'IBM Heavy Hexagon' (0.16%), as shown in Figure 9(a). This significant discrepancy comes from the fact that 'IBM Heavy Hexagon' measures gauge operators instead of the stabilizers for the Pauli-X error detection. Besides, 'IBM Heavy Hexagon' does not guarantee the fault tolerance of the Pauli-X error detection procedure. On the heavy square architecture, the error threshold of 'Surf-Stitch Heavy Square' is the same as that of 'IBM Heavy Square', as shown in Figure 9(b). This is because the code synthesized by Surf-Stitch is almost identical to 'IBM Heavy Square' except for stabilizers on boundaries, as shown in Figure 8(a)(c). The only difference is that 'IBM Heavy Square' removes some boundary nodes (dotted) and edges (dotted) for better efficiency of stabilizer measurements on the borderline.

In summary, Surf-Stitch can automatically generate surface codes that have similar or even better error correction capability compared with manually designed QEC codes on the two studied architectures.

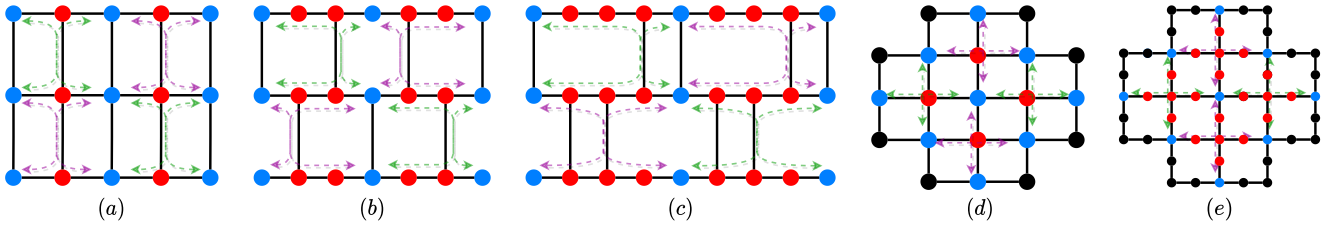


Figure 10: First four stabilizers of the synthesized surface codes by Surf-Stitch. (a)(b)(c) syntheses on the square, hexagon, and octagon architectures. (d)(e) syntheses on the square and heavy square architectures by using syndrome rectangles induced by 4-degree qubits.

Table 2: Metrics of the synthesized surface codes by Surf-Stitch. The average numbers of bridge qubits, CNOT gates, and time steps are computed over all X-type stabilizers.

Code	Avg. bridge qubit #	Avg. CNOT #	Avg. time-step #	Tot. time-step #	Error threshold
Surf-Stitch Heavy Sqaure	3	8	12	24	0.53%
Surf-Stitch Heavy Hexagon	7	19	20	40	0.33%
Surf-Stitch Sqaure	2	6	10	20	0.63%
Surf-Stitch Hexagon	4	10	13	26	0.47%
Surf-Stitch Octagon	6	14	14	28	0.38%
Surf-Stitch Sqaure-4	1	4	8	8	0.70%
Surf-Stitch Heavy Sqaure-4	5	12	13	13	0.45%
IBM Heavy Sqaure	3	8	12	24	0.53%
IBM Heavy Hexagon	3	8	12	24	0.16%

Table 3: Qubit utilization of the distance-5 surface codes synthesized by Surf-Stitch on different architectures.

Code	data qubit %	bridge qubit %	unused qubit %	Tot. qubit #
Surf-Stitch Heavy Sqaure	31.7%	45.6%	22.8%	79
Surf-Stitch Heavy Hexagon	18.8%	59.4%	21.8%	133
Surf-Stitch Square	55.6%	44.4%	0.0%	45
Surf-Stitch Hexagon	30.5%	48.8%	20.7%	82
Surf-Stitch Octagon	13.8%	75.8%	10.4%	116
Surf-Stitch Square-4	43.9%	56.1%	0.0%	57
Surf-Stitch Heavy Sqaure-4	16.3%	83.7%	0.0%	153
IBM Heavy Sqaure	31.7%	45.6%	22.8%	79
IBM Heavy Hexagon	17.4%	63.0%	19.6%	92

5.3 Synthesis on various SC architectures

We further apply Surf-Stitch to other architectures in Table 1 to demonstrate the general applicability of Surf-Stitch. Figure 10(a)-(c) presents the synthesis results of Surf-Stitch on the square, hexagon, and octagon architectures. Besides syntheses enabled by a pair of three-degree bridge qubits as in Figure 10(a)-(c), we also include another two surface codes generated by using syndrome rectangles centering around four-degree qubits, as shown in Figure 10(d)(e). These two codes have the suffix '-4' in the code name in Table 2 and Table 3. Table 2 summarizes the characteristics of stabilizer measurements in the synthesized surface codes by Surf-Stitch. Table 3 shows the resource requirements of these synthesized surface codes and is obtained by finding the smallest tiling of building blocks in Table 1 that is able to support the distance-5 surface code and then computing the ratios of different types of qubits.

The effect of architectures: High-degree architectures are more effective for surface code synthesis than low-degree architectures. Compared to polygon architectures, heavy architectures increase the bridge qubit number by 114% on average, up to 400%. Heavy architectures also increase the average time-step number by 40.7% on average. Fortunately, in Surf-Stitch, such significant resource differences do not lead to great error correction capability degradation. Compared to polygon architectures, heavy architectures reduce the error threshold by 26.7% on average. Besides, low-degree devices have a much lower hardware error rate and are easier to fabricate than high-degree devices [35].

Table 4: Resource scalability of Surf-Stitch on different architectures. d is the code distance. The 'Ideal' rows denotes the ideal surface code on a 2D lattice [19].

Code	bridge qubit #	bridge/data	2-qubit gate #	1-qubit gate #
Surf-Stitch Heavy Sqaure	$2(d^2-1)$	2	$8d(d-1)$	$2(d-1)(3d+1)$
Surf-Stitch Heavy Hexagon	$2(2d+1)(d-1)$	4	$4(4d-1)(d-1)$	$2(d-1)(7d-1)$
Surf-Stitch Square	$(d-1)(d+2)$	1	$6d(d-1)$	$2(d-1)(2d+1)$
Surf-Stitch Hexagon	$(2d+3)(d-1)$	2	$10d(d-1)$	$2(5d-1)(d-1)$
Surf-Stitch Octagon	$(d-1)(3d+7)$	3	$14d(d-1)$	$2(d-1)(6d+1)$
Ideal	d^2-1	1	$4(d^2-1)$	$2(d^2-1)$

The effect of the synthesis design: The synthesized codes centering on four-degree qubits have higher resource requirements than the synthesized codes induced by a pair of three-degree qubits. In Table 3, 26.7% and 93.7% more qubits are required for 'Surf-Stitch Square-4' and 'Surf-Stitch Heavy Square-4' than 'Surf-Stitch Square' and 'Surf-Stitch Heavy Square', respectively. The synthesis by four-degree qubits may also have a lower error threshold. Compared to 'Surf-Stitch Heavy Square', 'Surf-Stitch Heavy Square-4' decreases the error threshold by 15.1%.

Overall, not only the architectural design but also the synthesis design have a critical impact on the resource overhead and error correction capability of the synthesized codes. By optimizing the three key stages in the surface code synthesis, *Surf-Stitch* achieves reasonable error thresholds on various mainstream SC quantum architectures. In fact, IBM recently demonstrates a CNOT gate with a fidelity of 99.77% [29], whose physical error rate of 0.23% is lower than the worst error threshold by Surf-Stitch in Table 2 (0.33% on the heavy hexagon architecture).

Being effective in synthesizing surface codes, Surf-Stitch also has good scalability on quantum resources. Table 4 reports the bridge qubit number and quantum gate number by Surf-Stitch, with respect to the code distance d. These data are obtained by analyzing the patterns of the synthesized surface codes. As shown in Table 4, the required number of bridge qubits (also, two-qubit gates and single-qubit gates) in Surf-Stitch scales almost linearly with the number of data qubits, i.e., d^2. This indicates that no matter how large the code distance is, for a given architecture, Surf-Stitch

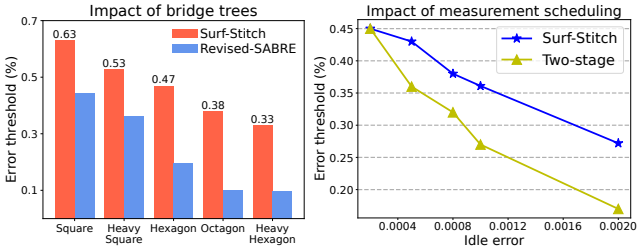


Figure 11: Sub-component analysis. (a) The impact of different bridge trees. (b) The impact of different measurement scheduling.

only needs a constant number of bridge qubits to measure one stabilizer. Such good scalability comes from the fact that Surf-Stitch always uses small syndrome rectangles and only considers *local* bridge trees that have limited sizes and do not grow as the code distance increases.

5.4 Analysis on sub-components

In this section, we study the effect of Surf-Stitch's three optimization stages.

Data qubit allocation: To demonstrate the necessity of a specialized data qubit allocation algorithm, we compare the data qubit allocation pass of Surf-Stitch to Qiskit [1] and the random sampling method on device architectures in Table 1. For Qiskit, we try two different qubit layouts: SABRE [34] and NoiseAdaptive [36]. The random sampling method tries to find data qubits by sampling the device nodes uniformly. During 100000 trials, Qiskit and the random sampling method do not produce any *valid* data qubit layout that is able to execute all stabilizer measurement circuits without moving data qubits. The reason for the failure of these methods is that they do not consider the constraints of the surface code synthesis: (a) once allocated, data qubits should not be moved; (b) there should be high-degree qubits between data qubits. In contrast, Surf-Stitch always produces valid data qubit layouts.

Bridge tree construction: Keeping other optimization steps fixed, we compare the bridge tree construction algorithm of Surf-Stitch to a *revised SABRE* routing algorithm. We use a revised SABRE here because the original SABRE cannot distinguish bridge qubits and data qubits. To make a CNOT gate executable, SABRE may move data qubits towards syndrome qubits. Such behavior is prohibited in the surface code because moving data qubits will invalidate the logical operations of the surface code. Besides, SABRE cannot guarantee the correct execution of stabilizer measurements. When we measure an X- and Z-stabilizer together, the order of the CNOT gates between syndrome qubits (or bridge qubits) and data qubits must follow the zig-zag ordering [19], as shown in Figure 3. Unfortunately, SABRE does not obey this constraint. Therefore, we revise the SABRE algorithm to make it applicable for implementing stabilizer measurements. As in Figure 11(a), the SABRE method is also significantly worse than Surf-Stitch due to the large number of extra CNOT gates induced by using SWAP gates. This result illustrates the necessity of a specialized bridge tree optimization pass.

Stabilizer measurement scheduling: Keeping the first two optimization stages fixed, we compare the stabilizer measurement scheduling of Surf-Stitch to the two-stage measurement scheme [32] which first measures all X-stabilizers and then measures Z-stabilizers. For the comparison, we consider the synthesized surface code in Figure 10(e). As shown in Figure 11(b), the stabilizer measurement schedule of Surf-Stitch achieves a higher error threshold than the two-stage schedule, and the advantage of Surf-Stitch increases as the idle error grows. Such a result demonstrates the effectiveness of Surf-Stitch's stabilizer measurement scheduling, especially for SC quantum devices in the near future.

6 DISCUSSION

To the best of our knowledge, this paper is the first attempt that formalizes the surface code synthesis problem on SC quantum devices. In particular, we identify three key challenges of the surface code synthesis: data qubit allocation, bridge tree construction, and stabilizer measurement schedule. We propose a modular framework to tackle these challenges one by one. The proposed framework is the first automated solution to the surface code synthesis problem, as far as we know. Our paper unveils the opportunities for the QEC code optimization on SC devices and would potentially inspire a series of works for closing the gap between QEC codes and SC quantum architectures.

Although we show that the surface code synthesized by our framework can achieve comparable or even better error correction capability than manually designed QEC codes, there is still much space left for potential improvements.

Improving data qubit allocation: For the concern of computational efficiency, this paper adopts a greedy strategy when constructing syndrome rectangles for stabilizer measurements. However, when the device architecture becomes complicated, our framework may be stuck at local minima and unable to generate the optimal data qubit layout. There are two ways to solve this problem: the first one is to incorporate more device information such as topological symmetries into the data qubit layout design; the second way is to use advanced optimization algorithms like simulated annealing or neural network to discover better data qubit layouts.

Exploring more design space: The proposed framework is modular, which makes it very extensible. However, such a solution cannot explore the entire solution space and may miss some optimization opportunities. For example, we may generate better bridge trees by co-optimizing the bridge tree finder and the stabilizer measurement scheduler. Or we may improve the data qubit layout by tweaking the data qubit allocator and the bridge tree finder jointly.

Adapting to other QEC codes: This paper is specifically designed for the surface code. Extra efforts may be needed to extend the proposed framework to other QEC codes. For example, the idea of using the bridge rectangle in our framework originates from the stabilizer shape of the surface code. Our data qubit allocator should be modified to process QEC codes with quite different stabilizer shapes, e.g., the color code [18] whose stabilizers can be of any shape.

7 RELATED WORK

General quantum compilers: Compiling a general quantum circuit onto an SC quantum device with limited qubit connectivity has been widely studied [17, 34, 36, 42, 45, 47, 48, 50]. However, these general quantum compilers are not suitable for compiling the surface code. Firstly, they don't distinguish data qubits from other qubits. They may move data qubits frequently, invalidating logical operations designed for a fixed data qubit layout [19]. Secondly, the SWAP gates they use to overcome the connectivity issue of the SC device make the compiled measurement circuits too error-prone for practical error correction. Finally, they do not account for the constraints on measurement circuits, e.g., the order of CNOT gates between syndrome qubits and data qubits [19].

Circuit compilation over the surface code: Most circuit compilation works on the surface code are at the higher logical circuit level. Javadi et al. [27] and Hua et al. [25] studied the routing congestion in circuits over the surface code. Ding et al. [15] and Paler et al. [39] studied the compilation of magic state distillation circuits with existing surface code logical operations for realizing a universal quantum gate set. Lao et al. [33] proposed a mapping process to execute lattice surgery-based quantum circuits on surface code architectures. These works assume that the ideal surface code architecture is already available and do not consider the problem of synthesizing surface codes on hardware. In contrast, this paper focuses on optimizing the surface code synthesis on various SC quantum architectures.

QEC code and architecture: Most efforts on QEC code synthesis are still on looking for an architecture that is suitable for the target code. Reichardt [40] proposed three possible planar qubit layouts for synthesizing the seven-qubit color code. Chamberland et al. [9] proposed a trivalent architecture where it is straightforward to allocate data qubits of triangular color codes. Chamberland et al. [10] introduced heavy architectures which reduce qubit frequency collisions while still providing support for the surface code synthesis. Instead, the synthesis framework in this paper can automatically synthesize the surface code onto various SC architectures and avoid manually redesigning code protocols for the ever-changing quantum architectures. Another line of research targets compiling stabilizer measurement circuits to existing SC architectures. Lao and Almudever [32] proposed the flag-bridge circuit which can measure the stabilizer of the Steane code on the IBM-20 device. However, their work relies on manually appointed data qubits and bridge qubits, and focuses on the IBM-20 device. Methods in this category are orthogonal to our work and can be easily merged into our framework.

8 CONCLUSION

In this paper, we formalize the three challenges of the surface code synthesis on SC quantum architectures and present an automatic synthesis framework to overcome these challenges. The proposed framework consists of three optimizations. Firstly, we adopt a geometrical method to allocate data qubits in a way that ensures the existence of shallow measurement circuits. Secondly, we select two heuristics to reduce the size of local bridge trees, which are enclosed by data qubits. Thirdly, we propose an iterative procedure to arrange the execution of measurement circuits based on a good initial schedule enabled by the proposed data qubit allocation. Our comparative evaluation with manually designed QEC codes demonstrates that, with good optimization, automated synthesis can even surpass manual QEC code design by experienced theorists.

ACKNOWLEDGMENTS

We thank the anonymous reviews for their constructive feedback. This work was supported in part by NSF 2048144. G. L. was in part funded by NSF QISE-NET fellowship under the award DMR1747426.

REFERENCES

[1] MD SAJID ANIS, Abby-Mitchell, Héctor Abraham, AduOffei, Rochisha Agarwal, Gabriele Agliardi, Merav Aharoni, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Sashwat Anagolum, Anthony-Gandon, Eli Arbel, Abraham Asfaw, Anish Athalye, Artur Avkhadiev, Carlos Azaustre, PRATHAMESH BHOLE, Abhik Banerjee, Santanu Banerjee, Will Bang, Aman Bansal, Panagiotis Barkoutsos, Ashish Barnawal, George Barron, George S. Barron, Luciano Bello, Yael Ben-Haim, M. Chandler Bennett, Daniel Bevenius, Dhruv Bhatnagar, Arjun Bhobe, Paolo Bianchini, Lev S. Bishop, Carsten Blank, Sorin Bolos, Soham Bopardikar, Samuel Bosch, Sebastian Brandhofer, Brandon, Sergey Bravyi, Nick Bronn, Bryce-Fuller, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adam Carriker, Ivan Carvalho, Adrian Chen, Chun-Fu Chen, Edward Chen, Jielun (Chris) Chen, Richard Chen, Franck Chevallier, Kartik Chinda, Rathish Cholarajan, Jerry M. Chow, Spencer Churchill, CisterMoke, Christian Claus, Christian Clauss, Caleb Clothier, Romilly Cocking, Ryan Cocuzzo, Jordan Connor, Filipe Correa, Zachary Crockett, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Navaneeth D, Sean Dague, Tareq El Dandachi, Animesh N Dangwal, Jonathan Daniel, Marcus Daniels, Matthieu Dartiailh, Abdón Rodríguez Davila, Faisal Debouni, Anton Dekusar, Amol Deshmukh, Mohit Deshpande, Delton Ding, Jun Doi, Eli M. Dow, Eric Drechsler, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem EL-Safty, Eric Eastman, Grant Eberle, Amir Ebrahimi, Pieter Eendebak, Daniel Egger, ElePT, Emilio, Alberto Espiricueta, Mark Everitt, Davide Facoetti, Farida, Paco Martín Fernández, Samuele Ferracin, Davide Ferrari, Axel Hernández Ferrera, Romain Fouilland, Albert Frisch, Andreas Fuhrer, Bryce Fuller, MELVIN GEORGE, Julien Gacon, Borja Godoy Gago, Claudio Gambella, Jay M. Gambetta, Adhisha Gammanpila, Luis Garcia, Tanya Garg, Shelly Garion, James R. Garrison, Jim Garrison, Tim Gates, Leron Gil, Austin Gilliam, Aditya Giridharan, Juan Gomez-Mosquera, Gonzalo, Salvador de la Puente González, Jesse Gorzinski, Ian Gould, Donny Greenberg, Dmitry Grinko, Wen Guan, Dani Guijo, John A. Gunnels, Harshit Gupta, Naman Gupta, Jakob M. Günther, Mikael Haglund, Isabel Haide, Ikko Hamamura, Omar Costa Hamido, Frank Harkins, Kevin Hartman, Areeq Hasan, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Stefan Hillmich, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Junye Huang, Rolf Huisman, Haruki Imai, Takashi Imamichi, Kazuaki Ishizaki, Ishwor, Raban Iten, Toshinari Itoko, Alexander Ivrii, Ali Javadi, Ali Javadi-Abhari, Wahaj Javed, Qian Jianhua, Madhav Jivrajani, Kiran Johns, Scott Johnstun, Jonathan-Shoemaker, JosDenmark, JoshDumo, John Judge, Tal Kachmann, Akshay Kale, Naoki Kanazawa, Jessica Kane, Kang-Bae, Annanay Kapila, Anton Karazeev, Paul Kassebaum, Tobias Kehrer, Josh Kelso, Scott Kelso, Vismai Khanderao, Spencer King, Yuri Kobayashi, Kovi11Day, Arseny Kovyrshin, Rajiv Krishnakumar, Vivek Krishnan, Kevin Krsulich, Prasad Kumkar, Gawel Kus, Ryan LaRose, Enrique Lacal, Raphaël Lambert, Haggai Landa, John Lapeyre, Joe Latone, Scott Lawrence, Christina Lee, Gushu Li, Jake Lishman, Dennis Liu, Peng Liu, Lolcroc, Abhishek K M, Liam Madden, Yunho Maeng, Saurav Maheshkar, Kahan Majmudar, Aleksei Malyshev, Mohamed El Mandouh, Joshua Manela, Manjula, Jakub Marecek, Manoel Marques, Kunal Marwaha, Dmitri Maslov, Paweł Maszota, Dolph Mathews, Atsushi Matsuo, Farai Mazhandu, Doug McClure, Maureen McElaney, Cameron McGarry, David McKay, Dan McPherson, Srujan Meesala, Dekel Meirom, Corey Mendell, Thomas Metcalfe, Martin Mevissen, Andrew Meyer, Antonio Mezzacapo, Rohit Midha, Daniel Miller, Zlatko Minev, Abby Mitchell, Nikolaj Moll, Alejandro Montanez, Gabriel Monteiro, Michael Duane Mooring, Renier Morales, Niall Moran, David Morcuende, Seif Mostafa, Mario Motta, Romain Moyard, Prakash Murali, Jan Müggenburg, Tristan NEMOZ, David Nadlinger, Ken Nakanishi, Giacomo Nannicini, Paul Nation, Edwin Navarro, Yehuda Naveh, Scott Wyman Neagle, Patrick Neuweiler, Aziz Ngoueya, Thien Nguyen, Johan Nicander, Nick-Singstock, Pradeep Niroula, Hassi Norlen, NuoWenLei, Lee James O'Riordan, Oluwatobi Ogunbayo, Pauline Ollitrault, Tamiya Onodera, Raul Otaolea, Steven Oud, Dan Padilha, Hanhee Paik, Soham Pal, Yuchen Pang, Ashish Panigrahi, Vincent R. Pascuzzi, Simone Perriello, Eric Peterson, Anna Phan, Kuba Pilch, Francesco Piro, Marco Pistoia, Christophe Piveteau, Julia Plewa, Pierre Pocreau, Alejandro Pozas-Kerstjens, Rafał Pracht, Milos Prokop, Viktor Prutyanov, Sumit

Puri, Daniel Puzzuoli, Jesús Pérez, Quant02, Quintiii, Rafey Iqbal Rahman, Arun Raja, Roshan Rajeev, Isha Rajput, Nipun Ramagiri, Anirudh Rao, Rudy Raymond, Oliver Reardon-Smith, Rafael Martín-Cuevas Redondo, Max Reuter, Julia Rice, Matt Riedemann, Rietesh, Drew Risinger, Marcello La Rocca, Diego M. Rodríguez, RohithKarur, Ben Rosand, Max Rossmannek, Mingi Ryu, Tharrmashastha SAPV, Nahum Rosa Cruz Sa, Arijit Saha, Abdullah Ash-Saki, Sankalp Sanand, Martin Sandberg, Hirmay Sandesara, Ritvik Sapra, Hayk Sargsyan, Aniruddha Sarkar, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Zachary Schoenfeld, Travis L. Scholten, Eddie Schoute, Mark Schulterbrandt, Joachim Schwarm, James Seaward, Sergi, Ismael Faro Sertage, Kanav Setia, Freya Shah, Nathan Shammah, Rohan Sharma, Yunong Shi, Jonathan Shoemaker, Adenilton Silva, Andrea Simonetto, Deeksha Singh, Divyanshu Singh, Parmeet Singh, Phattharaporn Singkanipa, Yukio Siraichi, Siri, Jesús Sistos, Iskandar Sitdikov, Seyon Sivarajah, Magnus Berg Sletfjerding, John A. Smolin, Mathias Soeken, Igor Olegovich Sokolov, Igor Sokolov, Vicente P. Soloviev, SooluThomas, Starfish, Dominik Steenken, Matt Stypulkoski, Adrien Suau, Shaojun Sun, Kevin J. Sung, Makoto Suwama, Oskar Słowik, Hitomi Takahashi, Tanvesh Takawale, Ivano Tavernelli, Charles Taylor, Pete Taylour, Soolu Thomas, Kevin Tian, Mathieu Tillet, Maddy Tod, Miroslav Tomasik, Caroline Tornow, Enrique de la Torre, Juan Luis Sánchez Toural, Kenso Trabing, Matthew Treinish, Dimitar Trenev, TrishaPe, Felix Truger, Georgios Tsilimigkounakis, Davindra Tulsi, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Adish Vartak, Almudena Carrera Vazquez, Prajjwal Vijaywargiya, Victor Villar, Bhargav Vishnu, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Johannes Weidenfeller, Rafal Wieczorek, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, WinterSoldier, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Steve Wood, James Wootton, Matt Wright, Lucy Xing, Jintao YU, Bo Yang, Unchun Yang, Jimmy Yao, Daniyar Yeralin, Ryota Yonekura, David Yonge-Mallo, Ryuhei Yoshida, Richard Young, Jessie Yu, Lebin Yu, Christopher Zachow, Laura Zdanski, Helena Zhang, Iulia Zidaru, Christa Zoufal, aeddins ibm, alexzhang13, b63, bartek bartlomiej, bcamorrison, brandhsn, charmerDark, deeplokhande, dekel.meirom, dime10, dlasecki, ehchen, fanizzamarco, fs1132429, gadial, galeinston, georgezhou20, georgios ts, gruu, hhorii, hykavitha, itoko, jeppevinkel, jessica angel7, jezerjojo14, jliu45, jscott2, klinvill, krutik2966, ma5x, michelle4654, msuwama, nico lgrs, ntgiwsvp, ordmoj, sagar pahwa, pritamsinha2304, ryancocuzzo, saktar unr, saswati qiskit, septembrr, sethmerkel, sg495, shaashwat, smturro2, sternparky, strickroman, tigerjack, tsura crisaldo, upsideon, vadebayo49, welien, willhbang, wmurphy collabstar, yang.luh, and Mantas Čepulkovskis. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

[2] F. Arute, K. Arya, R. Babbush, D. Bacon, J. Bardin, R. Barends, R. Biswas, S. Boixo, F. Brandão, D. Buell, B. Burkett, Y. Chen, Zijun Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, Keith Guerin, Steve Habegger, M. Harrigan, M. Hartmann, A. Ho, M. Hoffmann, Trent Huang, T. Humble, S. Isakov, E. Jeffrey, Zhang Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, Mike Lindmark, E. Lucero, Dmitry I. Lyakh, Salvatore Mandrà, J. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Niu, E. Ostby, A. Petukhov, John C. Platt, C. Quintana, E. Rieffel, P. Roushan, N. Rubin, D. Sank, K. Satzinger, V. Smelyanskiy, Kevin J. Sung, M. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. Yao, P. Yeh, Adam Zalcman, H. Neven, and J. Martinis. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574 (2019), 505–510.

[3] Torsten Asselmeyer-Maluga. 2021. 3D topological quantum computing. *International Journal of Quantum Information* (2021).

[4] Rami Barends, Julian Kelly, Anthony Megrant, Andrzej Veitia, Daniel Sank, Evan Jeffrey, Ted C White, Josh Mutus, Austin G Fowler, Brooks Campbell, et al. 2014. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature* 508, 7497 (2014), 500–503.

[5] Sergey B Bravyi and A Yu Kitaev. 1998. Quantum codes on a lattice with boundary. *arXiv preprint quant-ph/9811052* (1998).

[6] A Robert Calderbank, Eric M Rains, Peter W Shor, and Neil JA Sloane. 1997. Quantum error correction and orthogonal geometry. *Physical Review Letters* 78, 3 (1997), 405.

[7] A Robert Calderbank and Peter W Shor. 1996. Good quantum error-correcting codes exist. *Physical Review A* 54, 2 (1996), 1098.

[8] C. Chamberland and M. Beverland. 2017. FLAG FAULT-TOLERANT ERROR CORRECTION WITH ARBITRARY DISTANCE CODES. *arXiv: Quantum Physics* 2 (2017), 53.

[9] C. Chamberland, Aleksander Kubica, Theodore J. Yoder, and Guanyu Zhu. 2019. Triangular color codes on trivalent graphs with flag qubits. *arXiv: Quantum Physics* (2019).

[10] C. Chamberland, Guanyu Zhu, Theodore J. Yoder, J. Hertzberg, and A. Cross. 2020. Topological and Subsystem Codes on Low-Degree Graphs with Flag Qubits. *Physical Review X* 10 (2020).

[11] R. Chao and B. Reichardt. 2017. Fault-tolerant quantum computation with few qubits. *npj Quantum Information* 4 (2017), 1–8.

[12] R. Chao and B. Reichardt. 2019. Flag fault-tolerant error correction for any stabilizer code. *arXiv: Quantum Physics* (2019).

[13] Yu Chen, Charles J. Neill, Pedram Roushan, Nelson Leung, Michael Fang, Rami Barends, Julian Kelly, Brooks Campbell, Z Chen, Benjamin Chiaro, Andrew Dunsworth, Evan Jeffrey, Anthony Megrant, Josh Mutus, P. J. J. O'Malley, Chris Quintana, Daniel Thomas Sank, Amit Vainsencher, J. Wenner, Theodore White, Michael R. Geller, Andrew N Cleland, and John M. Martinis. 2014. Qubit Architecture with High Coherence and Fast Tunable Coupling. *Physical review letters* 113 22 (2014), 220502.

[14] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (2002), 4452–4505.

[15] Yongshan Ding, Adam Holmes, Ali JavadiAbhari, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. 2018. Magic-State Functional Units: Mapping and Scheduling Multi-Level Distillation Circuits for Fault-Tolerant Quantum Architectures. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), 828–840.

[16] David P. DiVincenzo and Ibm. 2000. The Physical Implementation of Quantum Computation. *Protein Science* 48 (2000), 771–783.

[17] Will Finigan, Michael Cubeddu, Thomas Lively, Johannes Flick, and Prineha Narang. 2018. Qubit allocation for noisy intermediate-scale quantum computers. *arXiv preprint arXiv:1810.08291* (2018).

[18] Austin G. Fowler. 2011. Two-dimensional color-code quantum computation. *Physical Review A* 83 (2011), 042310.

[19] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A* 86, 3 (2012), 032324.

[20] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (July 2021), 497. https://doi.org/10.22331/q-2021-07-06-497

[21] Alysson Gold, Anna Stockklauser, Matt Reagor, Jean-Philip Paquette, Andrew Bestwick, Cody James Winkleblack, Ben Scharmann, Feyza Oruc, and Brandon Langley. 2021. Experimental demonstration of entangling gates across chips in a multi-core QPU. *Bulletin of the American Physical Society* (2021).

[22] Daniel Gottesman. 1996. Class of quantum error-correcting codes saturating the quantum Hamming bound. *Physical Review A* 54, 3 (1996), 1862.

[23] Oscar Higgott. 2021. PyMatching: A Python package for decoding quantum codes with minimum-weight perfect matching. *arXiv preprint arXiv:2105.13082* (2021).

[24] Clare Horsman, Austin G. Fowler, Simon J. Devitt, and Rodney Van Meter. 2012. Surface code quantum computing by lattice surgery. *New Journal of Physics* 14 (2012), 123011.

[25] Fei Hua, Yan-Hao Chen, Yuwei Jin, Chi Zhang, Ari B. Hayes, Youtao Zhang, and Eddy Z. Zhang. 2021. AutoBraid: A Framework for Enabling Efficient Surface Code Communication in Quantum Computing. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021).

[26] J. Kelly. 2017. A Preview of Bristlecone, Google's New Quantum Processor. https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

[27] Ali JavadiAbhari, Pranav Gokhale, Adam Holmes, Diana Franklin, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong. 2017. Optimized Surface Code Communication in Superconducting Quantum Computers. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), 692–705.

[28] P. Jurcevic, Ali Javadi-Abhari, L. Bishop, I. Lauer, D. Bogorin, M. Brink, L. Capelluto, O. Günlük, Toshinari Itoko, Naoki Kanazawa, A. Kandala, G. Keefe, Kevin D Krsulich, W. Landers, E. Lewandowski, D. McClure, G. Nannicini, Adinath Narasgond, H. Nayfeh, E. Pritchett, M. Rothwell, S. Srinivasan, N. Sundaresan, Cindy Wang, K. X. Wei, C. J. Wood, J. Yau, E. Zhang, O. Dial, J. Chow, and J. Gambetta. 2020. Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science & Technology* 6 (2020).

[29] A. Kandala, K. X. Wei, S. Srinivasan, E. Magesan, S. Carnevale, G. A. Keefe, D. Klaus, O. Dial, and D. C. McKay. 2021. Demonstration of a High-Fidelity cnot Gate for Fixed-Frequency Transmons with Engineered ZZ Suppression. *Phys. Rev. Lett.* 127 (Sep 2021), 130501. Issue 13. https://doi.org/10.1103/PhysRevLett.127.130501

[30] Emanuel Knill, Raymond Laflamme, and Lorenza Viola. 2000. Theory of quantum error correction for general noise. *Physical Review Letters* 84, 11 (2000), 2525.

[31] Pieter Kok, William J. Munro, Kae Nemoto, Timothy C. Ralph, Jonathan P. Dowling, and Gerard J. Milburn. 2007. Linear optical quantum computing with photonic qubits. *Reviews of Modern Physics* 79 (2007), 135–174.

[32] L. Lao and C. G. Almudéver. 2020. Fault-tolerant quantum error correction on near-term quantum processors using flag and bridge qubits. *Physical Review A* 101 (2020), 032333.

[33] Lingling Lao, Bert van Wee, Imran Ashraf, J. van Someren, Nader Khammassi, Koen Bertels, and Carmen Garcia Almudever. 2018. Mapping of lattice surgery-based quantum circuits on surface code architectures. *Quantum Science and Technology* (2018).

[34] Gushu Li, Yufei Ding, and Yuan Xie. 2019. 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*. 1001–1014.

[35] Gushu Li, Yufei Ding, and Yuan Xie. 2020. Towards Efficient Superconducting Quantum Processor Architecture Design. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[36] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1015–1029.

[37] Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information.

[38] Hanhee Paik, David I. Schuster, Lev Bishop, Gerhard Kirchmair, Gianluigi Catelani, Adam P. Sears, B. R. Johnson, Matthew Reagor, Luigi Frunzio, Leonid I. Glazman, Steven M. Girvin, Michel H. Devoret, and Robert J. Schoelkopf. 2011. Observation of high coherence in Josephson junction qubits measured in a three-dimensional circuit QED architecture. *Physical review letters* 107 24 (2011), 240501.

[39] Alexandru Paler. 2019. SurfBraid: A concept tool for preparing and resource estimating quantum circuits protected by the surface code. *ArXiv* abs/1902.02417 (2019).

[40] B. Reichardt. 2018. Fault-tolerant quantum error correction for Steane's seven-qubit color code with few or no extra qubits. *arXiv: Quantum Physics* (2018).

[41] Peter W Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Physical review A* 52, 4 (1995), R2493.

[42] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. 2018. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 113–125.

[43] Andrew Steane. 1996. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 452, 1954 (1996), 2551–2577.

[44] Andrew M Steane. 1996. Error correcting codes in quantum theory. *Physical Review Letters* 77, 5 (1996), 793.

[45] Bochen Tan and Jason Cong. 2020. Optimal layout synthesis for quantum computing. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[46] Swamit S. Tannu, Zachary Myers, Prashant J. Nair, Douglas M. Carmean, and Moinuddin K. Qureshi. 2017. Taming the Instruction Bandwidth of Quantum Computers via Hardware-Managed Error Correction. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), 679–691.

[47] Swamit S Tannu and Moinuddin K Qureshi. 2019. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 279–290.

[48] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[49] Eric J Zhang, Srikanth Srinivasan, Neereja Sundaresan, Daniela F Bogorin, Yves Martin, Jared B Hertzberg, John Timmerwilke, Emily J Pritchett, Jeng-Bang Yau, Cindy Wang, et al. 2020. High-fidelity superconducting quantum processors via laser-annealing of transmon qubits. *arXiv preprint arXiv:2012.08475* (2020).

[50] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2018), 1226–1236.