Looped Pipelines Enabling Effective 3D Qubit Lattices in a Strictly 2D Device

Zhenyu Cai,[1,2,*] Adam Siegel,[1,2] and Simon Benjamin[1,2,†]

[1] *Quantum Motion, 9 Sterling Way, London N7 9HJ, United Kingdom*
[2] *Department of Materials, University of Oxford, Parks Road, Oxford OX1 3PH, United Kingdom*

(Received 25 March 2022; accepted 19 April 2023; published 16 June 2023)

Many quantum computing platforms are based on a two-dimensional (2D) physical layout. Here we explore a concept called *looped pipelines*, which permits one to obtain many of the advantages of a three-dimensional (3D) lattice while operating a strictly 2D device. The concept leverages qubit shuttling, a well-established feature in platforms like semiconductor spin qubits and trapped-ion qubits. The looped-pipeline architecture has similar hardware requirements to other shuttling approaches, but can process *a stack of qubit arrays* instead of just one. Even a stack of limited height is enabling for diverse schemes ranging from NISQ-era error mitigation through to fault-tolerant codes. For the former, protocols involving multiple states can be implemented with a space-time resource cost comparable to preparing *one* noisy copy. For the latter, one can realize a far broader variety of code structures; as an example we consider layered 2D codes within which transversal CNOTs are available. Under reasonable assumptions this approach can reduce the space-time cost of magic state distillation by 2 orders of magnitude. Numerical modeling using experimentally motivated noise models verifies that the architecture provides this benefit without significant reduction to the code's threshold.

DOI: 10.1103/PRXQuantum.4.020345

I. INTRODUCTION

Many platforms that are being explored for quantum computing have the property that qubits can only interact with physically proximal partners. For such systems, one may ask about the importance of the dimensionality of the qubit array: would one dimension (1D), two dimensions (2D), or three dimensions (3D) be required to achieve a given task efficiently? There are a number of studies related to this question. For example, within a certain framework it is possible to achieve quantum advantage using a noisy circuit of constant depth on a 3D qubit lattice [1], but it is not possible on a 2D qubit array if we are considering tasks like variational quantum algorithms [2]. When considering quantum error correction (QEC), moving from 2D qubit arrays to 3D qubit lattices enables 2D topological codes with transversal CNOT gates [3] or 3D topological codes with transversal non-Clifford gates [4,5]. In other applications like quantum error mitigation (QEM) [6,7] and quantum annealing [8], 3D qubit lattices can lessen or remove the demand for long-range gates, which are usually much noisier. However, despite the advantages provided by the 3D qubit lattices mentioned above, most hardware platforms nowadays are still confined to a 2D layout due to technological challenges. Hence, it is interesting to explore methods to efficiently implement an effective 3D qubit lattice on a physical 2D hardware substrate.

In many state-of-the-art hardware platforms, besides the qubit connectivity limitations, another key challenge is the conflict between providing spaces for classical control and maintaining a high qubit density. In hardware platforms where native entangling operations are short range, like silicon spin systems and trapped ions, we might expect to have an array of closely packed qubits. However, addressing qubits in such a closely packed array might lead to crosstalk. Moreover, the number of classical control elements required can scale with the area of the array (number of qubits) while the available space for these control elements may scale only with the perimeter of the array, leading to challenges in the wiring routing and heat dissipation [9]. To tackle the dichotomy of short-range interactions versus the desire for well-spaced structures, a natural solution is expanding the qubit array using shuttling tracks: qubits that are scheduled to interact are brought together through qubit shuttling. As a result, qubit shuttling forms the basis for many of the most promising scalable architectures for platforms like semiconductor spin qubits [10–12]

[*] cai.zhenyu.physics@gmail.com
[†] simon.benjamin@materials.ox.ac.uk

and trapped-ion qubits [13–17]. The related question of how to efficiently exploit the shuttling capabilities using suitable control sequences has also been considered in Refs. [18–21]. However, as we space out the qubit array using shuttling tracks, it appears that the qubit density will decrease accordingly, reducing the number of qubits that we can fit onto a single integrated platform (e.g., a silicon chip).

In this paper, we study a pipelining architecture, which is obtained by replacing the commonly considered linear shuttling tracks with shuttling loops. In this way, instead of storing and processing *a single 2D qubit array* in the shuttling-based architecture, we can effectively store and process *a stack of 2D qubit arrays* using a similar amount of physical resources, i.e., we aim to increase the qubit density for shuttling-based architecture without compromising any of the advantages brought by shuttling. Then, with the simple generalization of enabling (even limited) interactions between qubits in the same shuttling loop, we can perform transversal interactions between different layers in the stack of qubit arrays. This transforms the stack of qubit arrays into an effective 3D qubit lattice and enables computing tasks that are impossible before in a strictly 2D lattice of static, locally interacting qubits. Inevitably there are practical constraints on the height of the effective 3D lattice one can achieve, depending on the hardware and the circuits we try to run. However, as we see later, despite such limitations, very significant advantages can still emerge in various applications.

Our interest is in performing circuit-based quantum computation on matter qubits like silicon spin and trapped-ion qubits. However, before we proceed, we also want to take note of the extensive literature for building 3D cluster states in the photonic platform, which can be used for performing fault-tolerant measurement-based quantum computation to overcome challenges like nondeterministic two-qubit gates and photon losses [22–24]. Due to the “flying” nature of photonic qubits, in order to perform a given sequence of operations in a photonic processor, one must physically arrange a series of optical components along the optical path; similarly, in conventional computers, the classical circuits we want to perform are physically printed onto the classical processors. Due to such similarities, classical concepts like pipelining can be naturally adopted into photonic processors, turning into techniques like time-domain multiplexing and optical loops, which are then used in a range of proposed schemes for building 3D photonic cluster states [25–30]. On the other hand, typical architectures for *stationary* matter qubits operate on a completely different paradigm with the target circuit emerging from the time direction rather than physically appearing “baked into” the processor layout. Hence, the adoption of classical computing techniques for matter qubits is much less straightforward. In this paper, we construct a hybrid approach between flying and stationary qubits. This approach allows one to apply the classical pipelining concept to matter qubits and support features native to *3D circuit-based* scalable quantum computation, whereas existing schemes for performing *measurement-based* quantum computation using 3D photonic cluster states are effectively equivalent to *2D circuit-based* scalable quantum computation.

We start by looking at the general construction of the pipelining architecture in Sec. II. It is then used to implement a stack of 2D topological codes in Sec. III, and more specifically for the task of magic state distillation in Sec. IV. We consider the specific case of semiconductor platforms in Sec. V, where we present numerical simulations to determine thresholds for tolerable shuttling errors. In Sec. VI, we look at a near-term application of the looped-pipelining architecture in QEM. We conclude with some general remarks and a discussion of possible future directions in Sec. VII.

II. PIPELINING 2D QUBIT ARRAYS

A. Classical linear pipeline

Let us suppose we want to implement a classical data-processing circuit on many datasets. To convert the circuit into a *data-processing pipeline*, we divide it into multiple *steps* (also called *stages*) with each step able to operate only on *one dataset at a time* as shown in Fig. 1. Now instead of inputting the second dataset only after the first dataset completed the whole circuit, we can input the second dataset into the first data-processing step as soon as the first dataset leaves the first step, similarly for all the subsequent datasets and the subsequent steps. In this way, all data-processing steps can be working on different datasets in parallel, increasing the throughput of the system. This is the concept of *pipelining*. In fact, it is often the case that each step can actually perform a range of different operations with the same efficiency when using

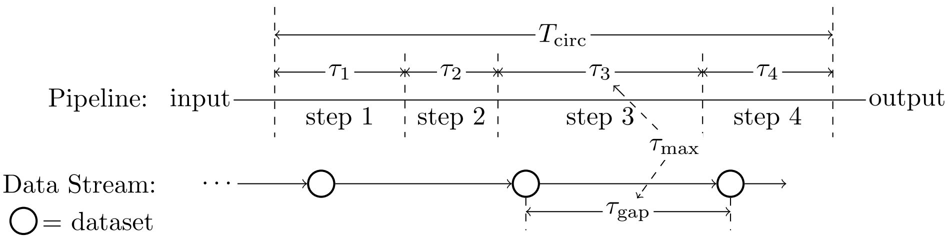


FIG. 1. Example of a classical data pipeline consists of 4 steps. Here we have a steady flow of datasets with a time gap $\tau_{\max}$ between consecutive datasets.

different control parameters. Hence, we can implement different circuits on the different datasets using the same data-processing pipeline.

The processing time needed for the mth step in the pipeline is denoted as τ_m. Suppose there are M steps in total, then the time taken for *one* dataset to flow through the entire pipeline and execute the whole circuit is simply

$$T_{\text{circ}} = \sum_{m=1}^{M} \tau_m. \tag{1}$$

This is the processing time required for *every* dataset without pipelining, and it is also the time required to process the *first* dataset in the pipeline. The additional time required to process the second dataset is determined by the slowest (rate-limiting) step in the pipeline with a processing time of

$$\tau_{\max} = \max_i \tau_i, \tag{2}$$

which is called *the rate-limiting step time*. This is because the second dataset can enter only the slowest step when the first dataset exits, similarly for any additional datasets. Hence, the total time needed to process k datasets using such a pipelining scheme is at least

$$T_{\text{pipe}}(k) = T_{\text{circ}} + (k-1)\tau_{\max}. \tag{3}$$

We call this the *steady-flow pipelining scheme* since the data stream can flow through the entire pipeline without needing to modify the time gap between adjacent datasets or be put on hold anywhere along the pipeline.

As we deviate from the steady-flow scheme, we often need to temporarily put the datasets on hold within the pipeline. To achieve this, we need to add *buffer*s in between the steps, which can hold multiple datasets temporarily and require no processing time.

B. Looped qubit pipeline

Similar to the classical pipeline, we can define a linear qubit pipeline as shown in Fig. 2(a) [31], with the datasets being qubits and the processing steps being shuttling steps, quantum gate devices, initialization and measurement devices. Each shuttling step here is a small section of shuttling track that can hold one and only one qubit at a time. We can also use the steady-flow scheme for the qubit pipeline, for which the time required for processing k qubits is simply given by Eq. (3). Note that in practice, each shuttling step is usually a very short shuttling section that performs relatively quickly compared to the other processes (see, e.g., Sec. V), thus the rate-limiting step is often not shuttling. We assume without loss of generality that shuttling tracks can hold the qubits without pushing them forward, thus they can also act as buffering regions for the pipeline when needed.

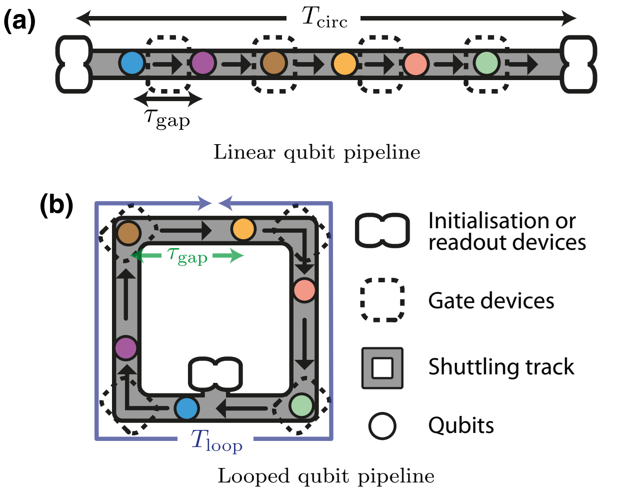


FIG. 2. Pipelines for applying single-qubit circuits on a stream of qubits.

Going beyond a linear pipeline, we can have a looped pipeline, which can be implemented using the structure shown in Fig. 2(b). This piece of hardware, which we simply refer to as a "*loop*," is made out of initialization and measurement devices connecting to an outer shuttling loop with gate devices. Unlike the linear qubit pipeline, in which the circuit depth we can perform is limited by the number of gate devices in the hardware, by allowing the qubits to go around the loop and reuse the gate devices, we can now effectively perform circuits of *any depth* even though we have only a small number of gate devices in the hardware.

Let us focus on the first qubit in the qubit stream, the time needed for it to wrap around the loop is called the *cycling period* and is denoted as T_{loop}. Assume that all additional qubits in the looped pipeline follow behind with a constant time gap of τ_{gap} between the consecutive qubits. When the qubit stream wraps around the loop, the first qubit in the qubit stream may collide with the last qubit. To avoid such *qubit collision*, we need to ensure the cycle period T_{loop} is larger than the time gap between the first and the last qubit, which is $(k-1)\tau_{\text{gap}}$ with k being the number of qubits:

$$T_{\text{loop}} \geq (k-1)\tau_{\text{gap}}.$$

The cycling period T_{loop} may change from one round to another since different steps on the loop can be activated at different rounds. We might want to apply three gates in this round, but only two gates in the next round; moreover, certain rounds may not involve measurement or initialization. Let us denote the *minimum cycling period* throughout the whole pipeline process as $T_{\text{loop}}^{\min}$, then to avoid qubit collision throughout the pipeline, we need to have

$$T_{\text{loop}}^{\text{min}} \geq (k-1)\tau_{\text{gap}},$$
$$k \leq T_{\text{loop}}^{\text{min}}/\tau_{\text{gap}} + 1 = K_{\text{loop}}, \tag{4}$$

where K_{loop} is the maximum number of qubits we can fit on the loop.

In the steady-flow scheme, we have $\tau_{\text{gap}} = \tau_{\text{max}}$. Therefore, the maximum number of qubits we can fit in the loop in this case is simply

$$K_{\text{loop}} = T_{\text{loop}}^{\text{min}}/\tau_{\text{max}} + 1. \tag{5}$$

If K_{loop} is too small and we want to fit in more qubits into the pipeline, we can try to increase $T_{\text{loop}}^{\text{min}}$ by adding buffering times, which we simply call *collision buffering*. In this paper, in most of the cases, we try to keep the qubit number in the loop low enough to satisfy Eq. (5) so that no collision buffering is needed.

We can also try to increase the number of pipelined qubits by reducing τ_{max}. When the rate-limiting step is measurement or initialization, we can reduce their effective processing time by adding more initialization or measurement devices, for example, as shown in Fig. 3, and operating them in parallel. With m times more initialization or measurement devices in the pipeline, the effective measurement and initialization time in the pipeline, and thus rate-limiting step time τ_{max}, will be reduced by a factor of m as long as we are operating on more than m qubits: $\tau_{\text{max}} = \tau_{\text{init/meas}}/m$. It is also possible to reduce the measurement or initialization time to the point that they are not the rate-limiting step anymore. It is worth noting that *without pipelining*, naively putting in more initialization or measurement devices will not reduce the processing time.

In the case that the rate-limiting step is the gate operation, we may be able to reduce its effective processing time by, for example, applying the gate operation while the qubit(s) involved traveling along a substantial portion of the loop. Then, the gate operation can be carried out alongside the shuttling process, instead of localizing the gate operation at a corner. As is further discussed in Sec. II D, this effectively decomposes the gate operation into many small gate steps to reduce the rate-limiting step time τ_{max}.

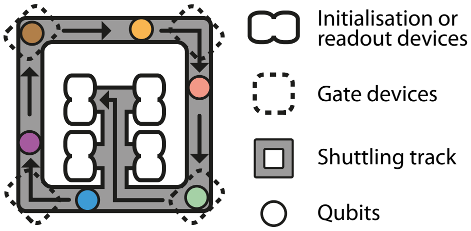


FIG. 3. Example of a looped qubit pipeline with multiple initialization or measurement devices.

One other possibility to fit more qubits in the loop is by adding bypasses to allow the qubits at the front of the qubit stream to lap the qubits at the back, so that the length of the whole qubit stream can wrap around the outer loop more than once. However, this could lead to more complex scheduling of the pipeline and possibly additional time cost.

It is also possible to fit in more qubits *without making any modification to the hardware* and instead by simply reducing the pipelining time gap between the qubits to below τ_{max} as mentioned in Sec. E. As we see later in our discussion in Sec. V A, this can be one of the most practical ways for solving the qubit collision problem.

C. Qubit-array pipeline

Let us consider a five-qubit array with a central qubit interacting with the four neighboring qubits. In order to avoid crosstalk among them and to provide additional spaces for the wiring of the classical controls, we can space out the qubits using shuttling loops as shown in Fig. 4(a), where the qubit array is now stored in a *loop array* formed from the loop elements mentioned in the last section. By synchronizing the movement of qubits in different loops, if the central qubit moves through one round of the loop, it will come into contact with all of the surrounding qubits along the way and interact with them, mimicking the connectivity of the five-qubit array as shown in Fig. 4(a). A method for synchronizing more complex qubit movements is outlined in Sec. A. Such an architecture can be easily extended to a large qubit array.

Now instead of one qubit per loop, we can fit multiple qubits in each loop as shown in Fig. 4(b), essentially loading multiple qubit arrays (denoted by different colors) into the loop array *without adding any additional components*. As shown in Fig. 4(b), by synchronizing the movement of qubits among different loops [as in the pure shuttling case in Fig. 4(a)] and allowing the various devices on the loops to operate in parallel, we can process a stack of qubit arrays from the top layer to the bottom layer in a pipelining manner. Hence, this loop array can be used to implement a *qubit-array pipeline* and thus we also refer to it as a *pipelining architecture*. Even in the simple case that the various layers of the virtual stack do not interact, this is already interesting for applications such as VQAs where one needs to repeatedly prepare and measure the same quantum state (to adequately learn a set of observables).

Let A denote the *number of (entire) loop arrays* we need to store and process all qubit arrays, which is proportional to the *space overhead* needed. When the number of qubit arrays is n, we can store them in $A = n$ different loop arrays [n copies of arrays similar to Fig. 4(a)] and process them all in *parallel*. Alternatively, we can store them all in a single loop array ($A = 1$) and process them using *pipelining*, leading to *a factor of n saving in the spatial overhead*,

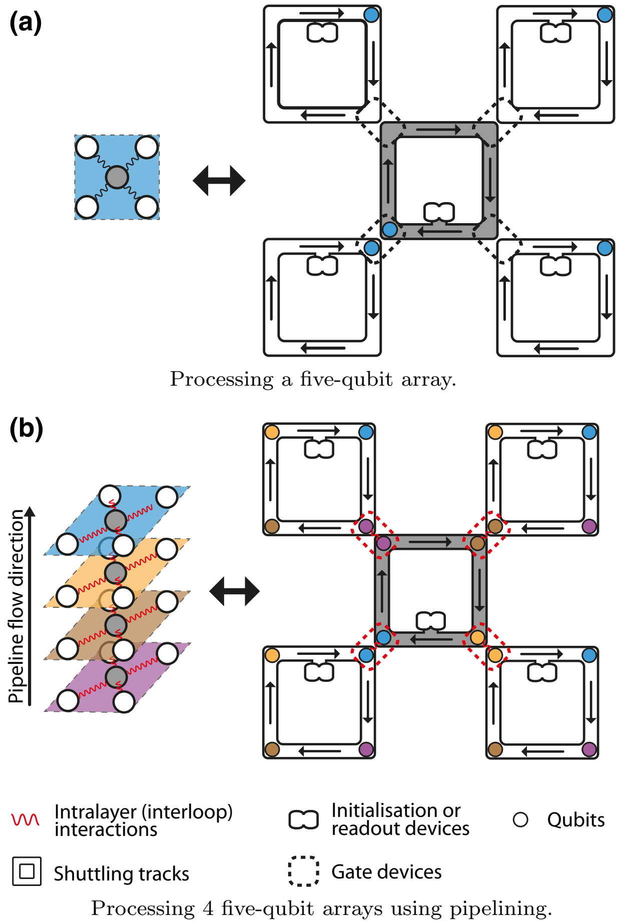

(a)
Processing a five-qubit array.
(b)
Pipeline flow direction
Intralayer (interloop) interactions
Initialisation or readout devices
Qubits
Shuttling tracks
Gate devices
Processing 4 five-qubit arrays using pipelining.

FIG. 4. Five-qubit arrays stored in a loop array. Multiple qubits can be stored on the same loop through pipelining, which corresponds to multiple layers of five-qubit arrays.

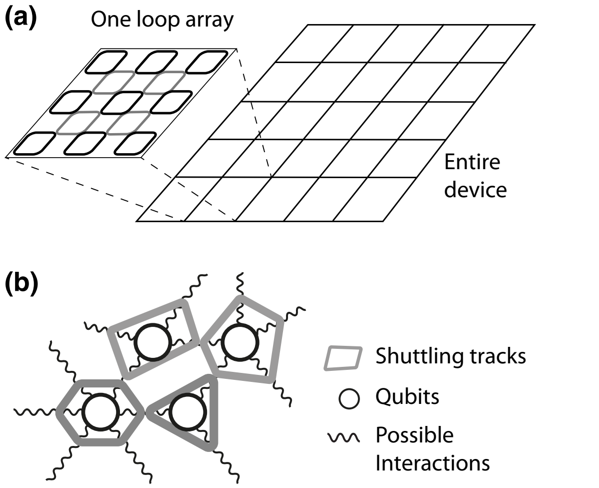


FIG. 5. (a) An entire array of loops, sufficient for a given role (a NISQ calculation, or representing one logical qubit) may be only part of the entire device. (b) Transforming a given qubit connectivity graph to a pipelining architecture. The shuttling loop of a given qubit is simply constructed by connecting up the midpoints of all of its interaction edges.

i.e., a factor of n increase in the qubit density. As given by Eq. (3), comparing the pipelining scheme to the parallel scheme, the time needed for the main computation is the same, which is simply T_{circ}, but processing each additional qubit using pipelining will increment the total time required by an amount $\tau_{\max}$ as in Eq. (3).

In an application, such as measuring observables by repeated sampling, it is likely that the number of qubit arrays n we want to process exceeds the maximum number of qubits we can fit in the pipeline K_{loop} given by Eq. (4). Then the need to distribute the n qubit arrays into at least $A = \lceil n/K_{\text{loop}} \rceil$ different loop arrays so that the number of qubit arrays stored in each loop array will not exceed K_{loop}. Nevertheless, we compress the spatial overhead by a factor of K_{loop}. This is further explored in the context of surface-code computation in Sec. III E.

The qubit-array pipeline can also be implemented on 2D qubit layouts beyond valence-4 connectivity as illustrated in Fig. 5. We can see that since the interloop interactions are carried out at the corners of the loop, the qubits with n neighboring qubits will simply be transformed into n-gon loops, whose n corners are connected to the corners of the n neighboring loops for interaction. There are of course other pipelining architectures possible as discussed in Sec. II D.

Generalizing the architecture described above, we can add components to allow interactions between specific qubits within the same loop; in the stack picture this corresponds to interactions between qubits in different layers. If the same pattern of intraloop interaction is repeated over a complete set of loops (as in the brown-purple pairing within the white loops in the lower panel of Fig. 6) then we implement "transversal" interactions between layers. In this way, the qubit connectivity is effectively extended beyond a 2D array to a 3D lattice. The "height" of this 3D lattice cannot be increased indefinitely due to the limitation on the number of qubits in each loop as discussed in Sec. II B. Nonetheless, the increased connectivity can expand the computation power of the device by, for example, enabling transversal CNOT gates in QEC codes and an efficient implementation of purification-based QEM as discussed later.

The simplest way of scheduling these intraloop interactions is just halting the pipelining flow, effectively halting all interloop interactions, and then bringing the corresponding qubits in the loop together for interaction. We can have a circuit structure of alternating layers of intralayer (interloop) and interlayer (intraloop) interactions just like the standard gate scheduling for a 3D qubit lattice. In this paper, we consider applications with mostly intralayer (interloop) operations and rare interlayer (intraloop) operations. Hence, we ignore the time cost for

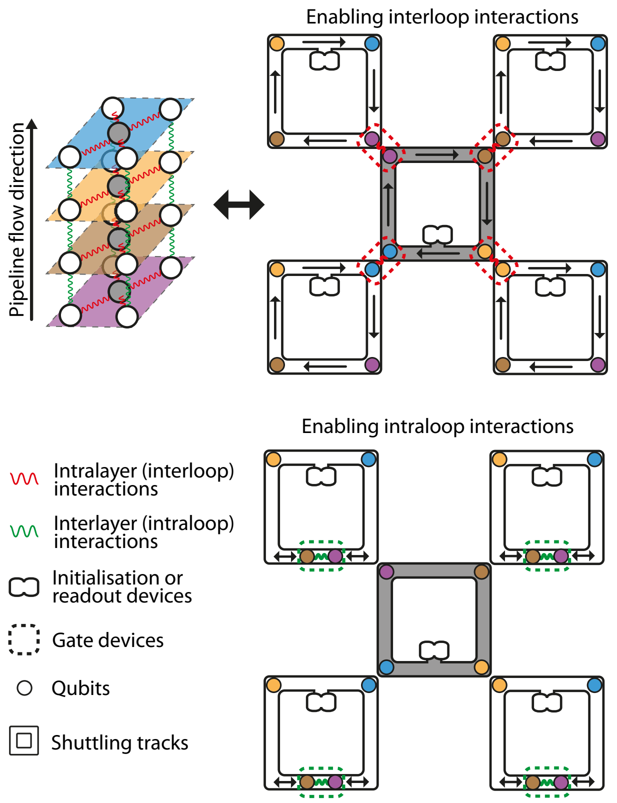

Enabling interloop interactions
Pipeline flow direction
Enabling intraloop interactions
Intralayer (interloop) interactions
Interlayer (intraloop) interactions
Initialisation or readout devices
Gate devices
Qubits
Shuttling tracks

FIG. 6. Interactions within layers are enabled by interloop (red) interactions while transversal interaction in between layers are enabled by intraloop (green) interactions.

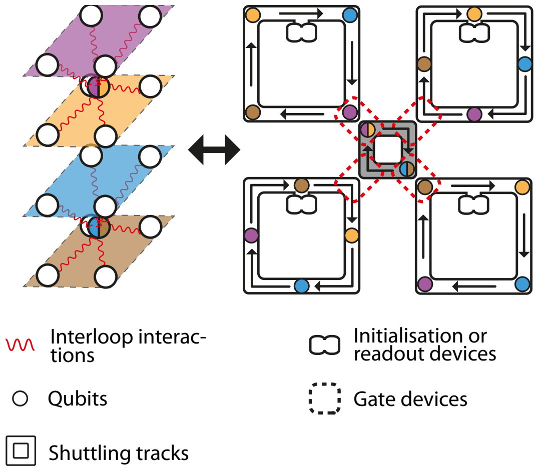


FIG. 7. Constructing 3D qubit connectivity without using intraloop interactions. Here the cycling frequency of the center (gray) loop is twice as fast as that of the surrounding (white) loops. Interactions between the middle two layers can be enabled if we allow qubit swaps within the same loops as shown in Fig. 8.

interlayer (intraloop) operations in the analysis for those applications.

We can also have different numbers of qubits and/or different cycling frequencies in different loops. In such a way, it is possible to construct 3D qubit structures beyond a stack of aligned layers, even without using intraloop interaction. One such example is shown in Fig. 7. If indeed we do also have intraloop interactions available, then we can construct qubit structures that are yet more sophisticated. One such example is shown in Fig. 8.

D. Alternative qubit-array pipeline implementations

The shape of the loop we construct in Fig. 5 is merely one of the more simple implementation possibilities. Instead of a simple loop of the shape of an n-gon, we can have more complicated shapes like, e.g., a “8” shape with a crossing at the middle (i.e., instead of having the shape of a graph *cycle*, we can have the shape of a graph *circuit*).

In Fig. 4, the corner interaction is illustrated as two separate tracks coming into proximity. In practice, such corner interaction can also be implemented using shutting junctions connecting the two loops as shown in Fig. 9, utilizing Y junctions that have been extensively considered for trapped-ion devices [19–21]. It is also possible to extend the interaction regions between the loops beyond the corners by increasing the overlaps between the edges of different loops as shown in Fig. 10(a). Pushing this to the extreme, we then have complete overlaps between the edges of different loops as shown in Fig. 10(b), which we simply call the *edge-interacting pipeline*.

Here we see that the qubit interactions are now extending over the whole edge of the loop instead of simply at the corner. It is now possible to operate on multiple qubit pairs simultaneously along a given edge. Such an extended gate-operation node along the whole edge can be viewed as many consecutive small gate-operation steps, with each small gate-operation step only able to operate on one qubit at a time. If the gate step is the rate-limiting step in the corner-interacting pipeline, by decomposing each gate operation into the m small gate steps in the edge-interacting pipeline, we can reduce the time required for the rate-limiting step by a factor of m. Of course, this assumes that the time needed for the gate operation for both the corner-interacting and edge-interacting pipelines are the same. In the edge-interacting pipeline, since we are carrying out shuttling and qubit interaction in parallel, there is a time saving on removing the shuttling time cost. The density of the loops in the loop array may also increase compared to the corner-interacting case. On the other hand, a qubit-qubit interaction realized while both qubits are transiting along the edge may not be possible in certain platforms, or may have lower fidelity than stationary qubit interaction at the corner.

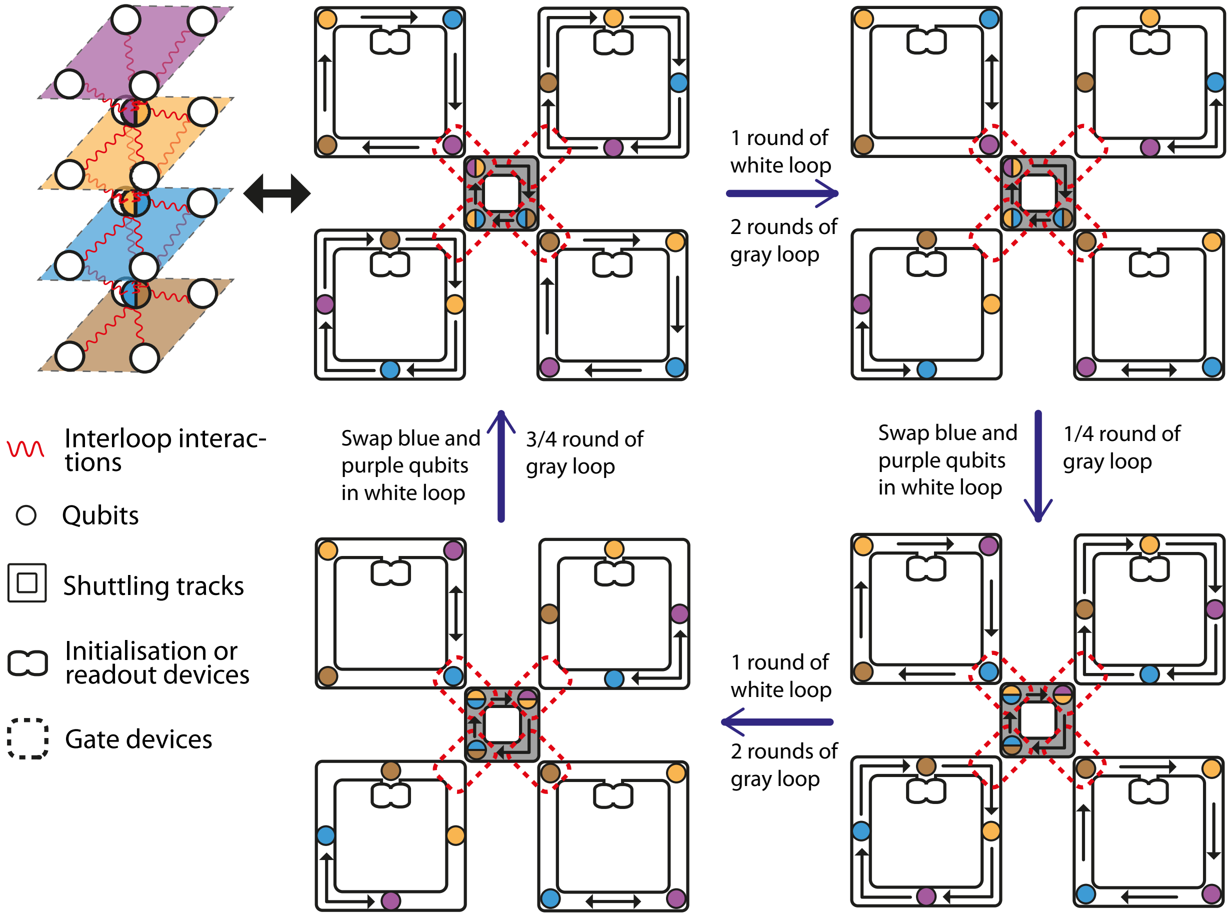


FIG. 8. A qubit pipelining scheme to achieve the qubit connectivity shown on the left using loops of different cycling frequencies and a different number of qubits. Here the only intraloop operation we need is swapping the qubits in the same loop, which can be carried out using swap gates or via qubit shuttling if we allow one qubit to bypass another qubit in the loop. Qubit bypass in the loop can be achieved by adding structures to allow one qubit to move off the loop to give way to another qubit.

III. STACKING 2D TOPOLOGICAL CODES

A class of QEC codes that are particularly relevant to physical implementation is *2D (planar) topological codes*. These codes can naturally match to the geometric layouts of physical qubits to ensure *local* stabilizer checks of *constant weights* [32,33]. When performing fault-tolerant quantum computation using 2D topological codes, we have a set of 2D qubit arrays (storing different logical qubits) that need to be processed in the exact same way to perform stabilizer checks for quantum memory. This is a perfect setting for applying the qubit-array pipeline discussed in Sec. II C, which will *increase the density of the logical qubits*. Furthermore, when we enable intraloop operations (or equivalent extensions), then we are able to perform transversal operations between the logical qubits for *faster fault-tolerant quantum computation*, or indeed to realize 3D codes.

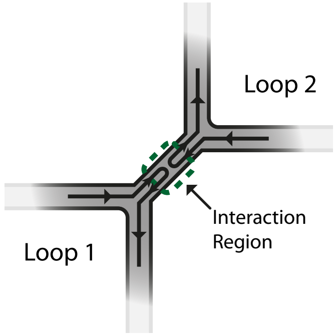
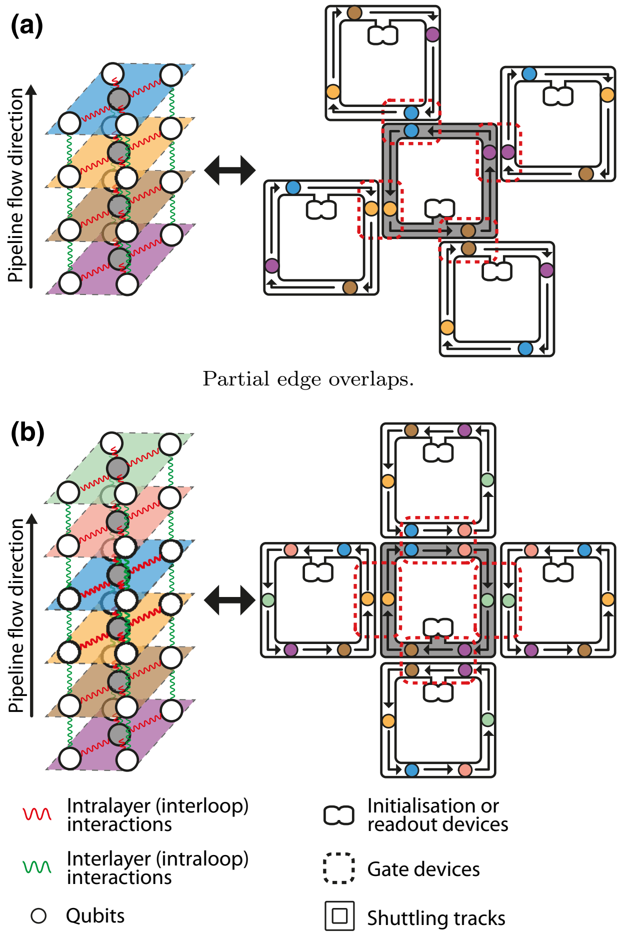


FIG. 9. One possible way to implement the interactions between two loops using shuttling junctions.

A. Pipelining time cost for quantum error correction

We assume the entire pipeline of QEC consists of D code cycles and ignore the initialization of all qubits at the beginning and the measurement of all qubits at the end since the associated time costs are negligible compared to


(a)
Pipeline flow direction
(b)
Pipeline flow direction
Intralayer (interloop) interactions
Interlayer (intraloop) interactions
Qubits
Initialisation or readout devices
Gate devices
Shuttling tracks

Partial edge overlaps.

Complete edge overlaps.

FIG. 10. Edge-interacting qubit-array pipelines. Intraloop gate devices are not explicitly shown for visual clarity. We see that multiple qubit pairs can interact simultaneously along a given edge.

the main code cycles. The time required for *one* logical qubit to run through *one* code cycle is denoted as T_{cycle} and called the *code cycle time*. Following Eq. (3), the time needed to pipeline k logical qubits through D code cycles is simply

$$\begin{aligned} T_{\text{pipe}}(D,k) &= DT_{\text{cycle}} + (k-1)\tau_{\max} \\ &= (D + (k-1)f\,)\,T_{\text{cycle}}, \end{aligned} \tag{6}$$

with $f\, = \tau_{\max}/T_{\text{cycle}}$ being the fraction of the code cycle time taken by the rate-limiting step.

The depth of the logical circuit D_{circ} is the number of layers of logical operations we need to perform, which is usually much smaller than the number of code cycles D. For most of the interesting applications, we usually find circuit depth scales more than linearly with the number of logical qubits needed n: $D_{\text{circ}} \gtrsim n$. Now using the fact that the total number of logical qubits n is larger than the number of logical qubits in each qubit-array pipeline k, and $f\, \leq 1$, we have $D \gg D_{\text{circ}} \gtrsim n > k > (k-1)f\,$. When applied to Eq. (6), this implies that *in many fault-tolerant applications, the pipelining time cost is negligible.*

B. Surface codes

Surface codes and color codes are two of the most promising topological codes at the moment due to their 2D planar layout, high thresholds, and the existence of efficient decoding algorithms [34,35]. Let us first use surface codes as an example. In surface codes we need to perform local X/Z checks, which are represented using the red and green plaquettes in Fig. 11(a). For each X check, we use the circuit in Fig. 11 to measure the X parity of the data qubits located at the vertices of red plaquettes (semicircles at the boundary), similarly for Z checks.

Each stabilizer check is a five-qubit process that can be implemented in the pipelining architecture in Fig. 4(b). There the data qubits are transformed into the surrounding (white) *data loops* and the ancilla qubit is transformed into the central (gray) *ancilla loop*. After a given ancilla qubit moves around the loop, it is able to interact with every surrounding data qubit and carry out the parity-check circuit in Fig. 11.

The pipelining structure can be easily extended beyond the five-qubit check array to the whole surface-code patch as shown in Fig. 12. It is shown in Ref. [34] that X and Z checks in the surface code can be carried out in parallel if we perform each check in a zigzag pattern across the four data qubits. The way we perform each check in a circular manner in a loop in Fig. 4(b) means that we can perform only X and Z checks in a staggered manner instead of in parallel. Of course, this is a feature of the round-shaped shuttling track rather than of pipelining, and it can be prevented by having a zigzag shuttling track instead. It is also worth noting that even without pipelining, we might want to stagger X and Z checks anyway since it can improve code performance by preventing error propagations and/or help with combating special kinds of errors [36–39].

In a given stabilizer check, we need to be careful about "hook" errors [3] in which single-qubit errors on the ancilla qubits propagate and become weight-two errors on data qubits. To mitigate the damage due to hook errors, when performing checks around a loop, we need to start the X checks at the position of the purple or orange dot in Fig. 12, while the Z checks need to start at the position of the blue or brown dot, such that the resultant weight-two errors do not align with the logical operators. This is reflected through the different orientation of the initialization or measurement devices in X and Z ancilla in Fig. 12.

We construct a pipeline for one surface-code cycle taking into account the restrictions above as shown in Fig. 13, which is written as a combination of the data pipelines, X ancilla pipelines and Z ancilla pipelines. We denote the

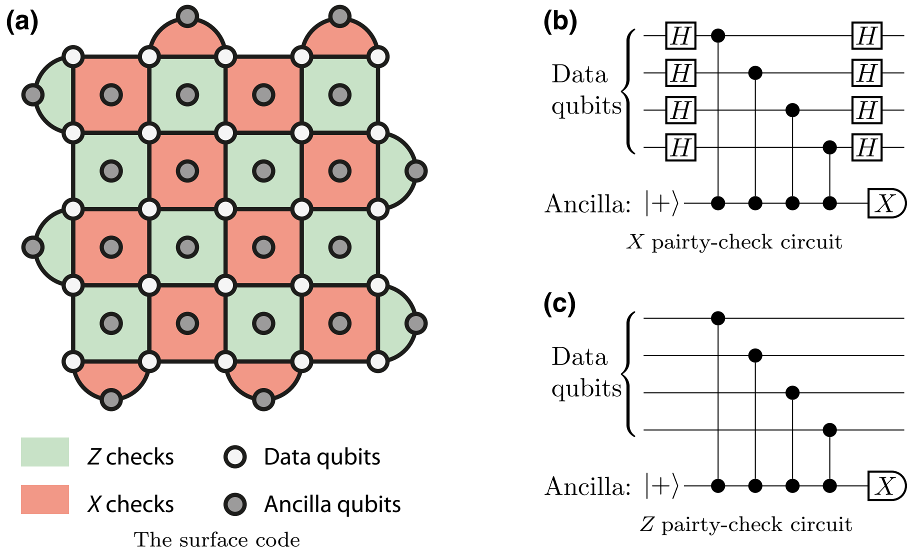


FIG. 11. The surface code and its parity-check circuits.

time required for qubit measurements, initialization, CZ gates, H gates, and shuttling through *one full round* of the loop as τ_{meas}, τ_{init}, τ_{CZ}, τ_{H}, and τ_{sh}, respectively. The rate-limiting step time is given by

$$\tau_{\max} = \max(\tau_{\text{meas}}, \tau_{\text{init}}, \tau_{\text{CZ}}, \tau_{\text{H}}).$$

Note that τ_{sh} contains many small shuttling steps and we assume that shuttling is not the rate-limiting step as mentioned before (this is merely to simplify our analysis and is not essential).

The code-cycle time T_{cycle} is obtained by looking at the data pipeline and the ancilla pipeline in one code cycle. In each code cycle, the data qubits will flow through three

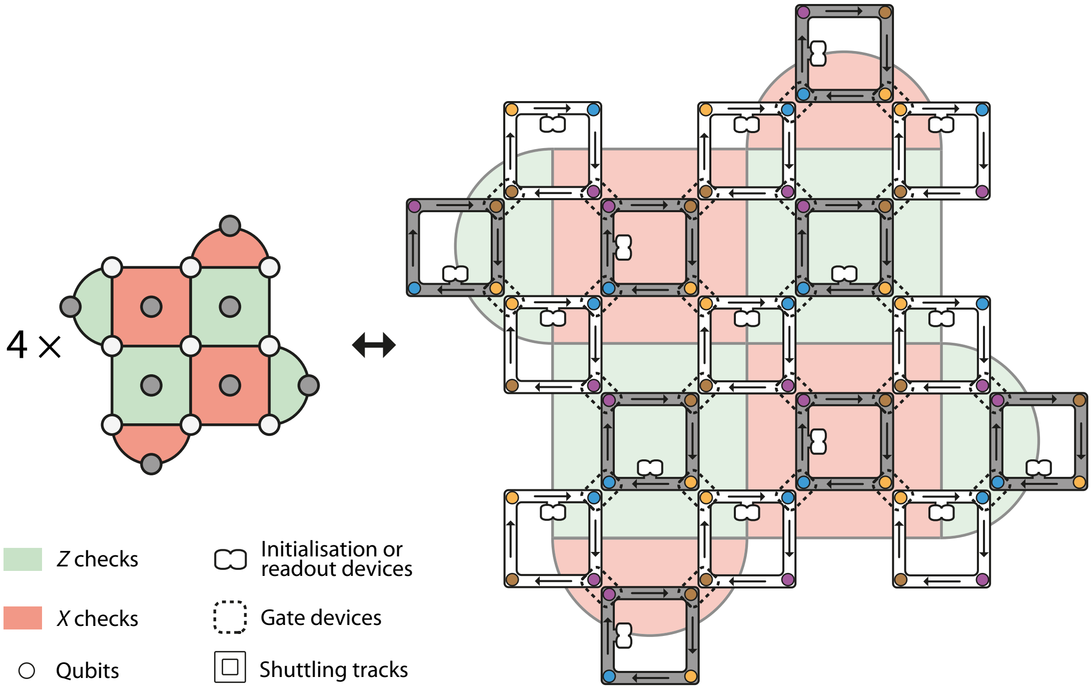


FIG. 12. Pipelining architecture for surface codes. Intraloop gate devices are not explicitly shown for visual clarity. The initialize or readout devices on the data qubit tracks are not used during a normal stabilizer cycle. More specialized hardware design with heterogeneous data and ancilla loops is possible, but is not shown here for simplicity. Note that here all data and ancilla pipelines are flowing in the clockwise direction. It is also possible to run in a configuration that all data pipelines flow clockwise while all ancilla pipeline flow anticlockwise.

rounds of the loop with eight CZs and two H applied along the way. Thus the circuit time needed for the data qubits for one code cycle is

$$T^{\mathrm{data}}_{\mathrm{circ}} = 3\tau_{\mathrm{sh}} + 8\tau_{\mathrm{CZ}} + 2\tau_{\mathrm{H}}. \tag{7}$$

On the other hand, the ancilla qubits (both X and Z) flow through one round of the loop with initialization, four CZs and measurement applied along the way. Thus the circuit time needed for the ancilla qubits for one code cycle is

$$T^{\mathrm{anc}}_{\mathrm{circ}} = \tau_{\mathrm{sh}} + 4\tau_{\mathrm{CZ}} + \tau_{\mathrm{init}} + \tau_{\mathrm{meas}}. \tag{8}$$

The code-cycle time is simply determined by the rate-limiting pipeline

$$T_{\mathrm{cycle}} = \max(T^{\mathrm{data}}_{\mathrm{circ}}, T^{\mathrm{anc}}_{\mathrm{circ}}), \tag{9}$$

with buffering added to the faster pipeline to achieve synchronization. We can then use Eqs. (6) and (9) to obtain the time needed for pipelining k surface-code patches through D cycles.

For the data pipeline, the data qubits go around the loop three rounds in each code cycle, and the minimum cycling period is given by the last round in Fig. 13 in which only one CZ gate and one H gate are applied: $T^{\mathrm{min}}_{\mathrm{loop}} = \tau_{\mathrm{sh}} + \tau_{\mathrm{CZ}} + \tau_{\mathrm{H}}$. Hence, the maximum number of qubits we can fit into the pipeline is given by Eq. (5):

$$K_{\mathrm{loop}} = \frac{T^{\mathrm{min}}_{\mathrm{loop}}}{\tau_{\mathrm{max}}} + 1 = \frac{\tau_{\mathrm{sh}} + \tau_{\mathrm{CZ}} + \tau_{\mathrm{H}}}{\tau_{\mathrm{max}}} + 1. \tag{10}$$

For the ancilla pipeline, the ancilla qubit goes through one round of the loop without wrapping around, thus there is no danger of the front of the qubit stream colliding with the rear.

When the rate-limiting step is measurement or initialization, we can reduce the effective rate-limiting step time by adding more initialization or measurement devices as discussed in Sec. II B. Note that in many platforms, one can perform nondestructive projective measurements (e.g., Pauli spin blockade measurement for semiconductor spin qubits in Sec. V A), which acts as the initialization step for the next code cycle and thus we have $\tau_{\mathrm{init}} = 0$. When the CZ or H gate is the rate-limiting step, we might switch to a loop array with edge interactions to reduce the effective rate-limiting step time as discussed in Sec. II D.

The pipeline we propose in Fig. 13 is simply one of the possible workflows. A simpler (but not necessarily efficient) stabilizer check scheme can be mimicking the conventional zigzag checking pattern [34] by flowing the qubits through two rounds of the loops and activating the gates at the appropriate moments. There are also other possibilities depending on the exact processing steps and the structure of the loops, which would be an interesting future direction to investigate.

C. Color codes

The structure of the color code [40] is shown in Fig. 14. In color codes, instead of performing X and Z checks separately on different plaquettes like in surface codes, we need to perform both X and Z checks for all plaquettes. Such a structure is why we can implement transversal H and S logical gates in color codes. The local checks in color codes are weight-6 (weight-4 at the boundary), and thus can be carried out using a parity-check circuit similar to Fig. 11 but with six data qubits. As the parity-check circuit involves more qubits and gates, the error threshold of color codes is usually lower than that of the surface codes.

Color codes can be transformed into a pipelining architecture following Sec. II. Each stabilizer check becomes the structure in Fig. 15 in which the data qubits are transformed into the surrounding (white) data loops and the ancilla qubit is transformed into the central (gray) ancilla loop. Since we have weight-6 checks (weight-4 at the boundary) for color codes, we see that the ancilla loops are hexagons. The data loops are triangles since each data qubit is connected to three ancilla qubits. Note that performing checks using a single ancilla in triangular color codes is prone to hook errors just as in surface codes. This can be avoided by adding extra ancilla qubits called *flag qubits* to detect these hook errors [41], and the new qubit layouts with the additional flag qubits can still be pipelined using the method discussed in Sec. II.

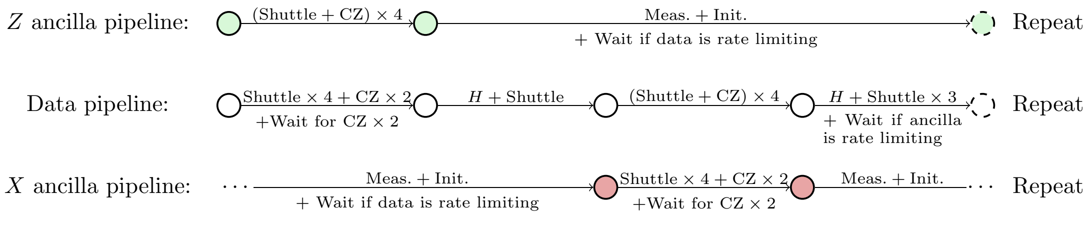


FIG. 13. Pipeline workflow for a code cycle in surface codes. “Shuttle” here means shuttling along *one edge* of the square shuttling loop. “Wait” here means buffering and it needs to be added into the pipeline for synchronization depending on which pipeline is the rate-limiting pipeline. We do not include the initializations of qubits before all code cycles and the measurement of all qubits after all code cycles.

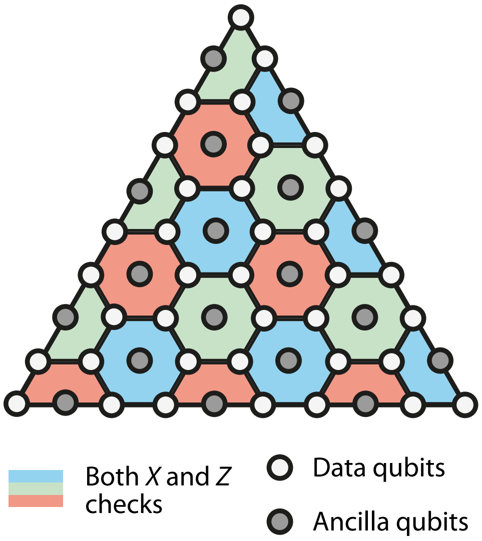


FIG. 14. A logical qubit in the color code.

In surface codes, each ancilla loop is either responsible for the X checks or the Z checks, thus the operations in the X checks and the Z checks can be carried out in parallel. On the other hand, in color codes, every ancilla loop is responsible for both the X and Z checks, thus in the simplest case, we can carry out only the X and Z checks in a strictly sequential manner. A possible code-cycle pipeline for the color codes is discussed in Sec. B.

D. Transversal logical CNOT

Most of the state-of-the-art topological codes like surface codes and color codes are Calderbank-Shor-Steane (CSS) codes for which the stabilizer checks are either X parity checks or Z parity checks, i.e., there are no checks consisting of a mixture of X and Z operators. For CSS codes, the logical CNOT can be implemented transversally, i.e., it can be implemented by performing physical CNOT between the corresponding physical qubits in the two logical qubits [42]. Such a transversal logical CNOT is usually not considered in practical implementations since it is challenging to link the corresponding physical qubits within two topological code patches in a 2D architecture. Therefore, instead, people turn to defect-based methods [23,24] or lattice surgeries [43,44]. However, these methods are based on *code deformation*, which requires $\mathcal{O}(d)$ code cycles to be performed before the CNOT can be successfully completed, where d is the *distance of the code*. In contrast, transversal CNOTs and more generally any transversal gates can be directly implemented in the looped pipeline paradigm; one means is simply to modify a single code cycle as discussed in Sec. II C. If this modified cycle introduces only a typical level of noise to the device, then no additional code cycles are required—this is the assumption we make presently in our resource estimations. However, even if higher-than-normal noise is introduced by the modified cycle, we would require only a small fixed (d-independent) number of additional cycles [45], which would not significantly alter our conclusions.

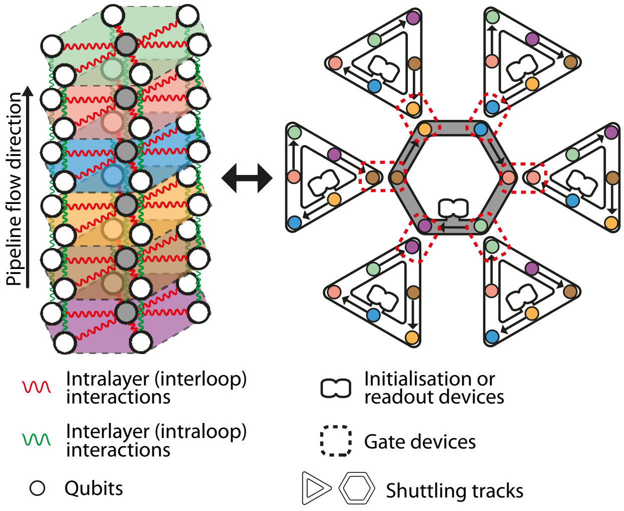

Pipeline flow direction
Intralayer (interloop) interactions
Interlayer (intraloop) interactions
Qubits
Initialisation or readout devices
Gate devices
Shuttling tracks

FIG. 15. Pipelining architecture for color-code parity checks with six qubits in the pipeline. Intraloop gate devices are not explicitly shown for visual clarity. Here we effectively have six layers of parity-check units stacking on top of each other.

The numbers of code cycles required for the different logical operations in the universal set are summarized in Table I. We see that CNOTs, if implemented via lattice surgeries, can be a big part of the time cost for both surface codes and color codes. For color codes, CNOT via lattice surgeries would be the *only* operation requiring $\mathcal{O}(d)$ code cycles and thus optimizing it via pipelining can potentially speed up the computation by a large factor. For the surface code, CNOT is one of the slowest steps, but there are other steps requiring $\mathcal{O}(d)$ code cycles like the Hadamard gate H and the phase gate S. One must also note that lattice surgeries also require a larger qubit overhead (more ancilla patches) than transversal operations.

E. Multiple code stacks

Due to the qubit collision constraint outlined in Sec. II B, there is a limit to the number of logical qubits (2D code layers) that we can fit into the same pipeline (stack). When the number of logical qubits exceeds the capacity of a single stack we employ multiple stacks forming an array, as we now discuss.

For a conventional 2D architecture, we might partition the device into multiple zones, or *patches*, each of which can store a separate logical qubit. In our looped-pipeline architecture, we may also employ patches whose lateral dimensions are sufficient to represent one logical qubit; however those dimensions are measured now in terms of a certain number of complete loops. Therefore, in each

TABLE I. Number of code cycles needed for different logical operations. The correspondence is not exact due to reasons including the following: (1) we assume that d code cycles are needed for fault-tolerant syndrome extraction, which might change with, e.g., the measurement error rate; (2) we assume transversal operations have low enough noise such that the usual frequency of code cycles for memories is enough for correcting their errors. However, this table gives us a good order-of-magnitude estimate for the operational speed of a given scheme.

	H	S	CNOT	$\vert T\rangle$ Init.	$\vert 0\rangle/\vert +\rangle$ Init.	X/Z Meas.
Surface code	$3d$ [46,47][a]	d [46,48][b]	$2d$ [46] (lattice surgery) or 0 (transversal)	6 [44]	0 [47][c]	0
Color code	0	0				

[a] 3d code cycles are needed for rotating a surface code in place (with the help of ancilla patches).

[b] Code distance is halved during the gate.

[c] We need $\mathcal{O}(d)$ code cycles for projecting the incoming state into the code space. However, this can be done simultaneously with all the other operations after initialization. Hence, as long as there are $\mathcal{O}(d)$ code cycles between the qubit initialization and measurement, the initialization time cost can be taken to be 0, otherwise we need to take it to be $\mathcal{O}(d)$.

patch we can in fact store *a stack* of k logical qubits (layers), where k is also the number of physical qubits within each loop. Denoting by A the number of distinct patches, each now seen as supporting a stack, we have kA logical qubits in total. Assuming that k is fixed for a given hardware technology and application, then we scale the device by increasing A, i.e., adding to the number of hardware patches—this is the spatial overhead of a given architecture.

Within the same stack, fast transversal CNOTs between logical qubits are available, while in between different stacks, logical CNOTs need to be carried out through, e.g., lattice surgery. Within the same stack, swapping logical qubits can be carried out by transversally swapping the positions of the corresponding qubits in each loop, and thus can be implemented using only shuttling operations, which are likely to be much faster than other steps in the pipeline. Such fast and reliable swap operations within the stack, along with the constant height of the stack, mean that we effectively have all-to-all connectivity within the stack. On the other hand, the connectivity between logical qubits in different stacks is dependent on the layout of the data and ancilla patches. If we have a chequerboard pattern of data and ancilla patches, then we just have 2D nearest-neighbor connectivity between data stacks via lattice surgery.

The simple solution mentioned above can be improved at the cost of using some of our resource as ancillas. A suitable layout is shown in Fig. 16 where we have 3 ancilla patches per data patch; we then ensure direct connectivity between any two data stacks via lattice surgery. This can be achieved by using the ancilla space (white regions of Fig. 16) around the two data stacks to create a Bell pair through which the long-range CNOT can be performed [49,50]. Such a layout can, of course, be used without pipelining—i.e., using single-layer data patches instead of stacks—but then there is a limitation to the parallelizability of the lattice-surgery CNOT gates since the connecting ancilla patches cannot cross each other [50], see Fig. 17(a). By adopting pipelining, lattice surgeries can be carried out in parallel in different layers of the stacks, and thus having k layers in each stack will increase the parallelizability of the CNOT gates by k fold (on top of enabling transversal CNOTs within each stack). A loose analogy is the use of flyovers to permit highways to cross one another without intersecting.

While the advantages of introducing additional ancilla space are attractive, we stress that this is not essential for the pipelining paradigm: pipelining k layers in each stack immediately leads to a k-fold reduction in the spatial overhead compared to the other shuttling-based architectures. Indeed, the improved logical qubit connectivity in the pipelining architecture also means that a chequerboard arrangement of the data and ancilla stacks could be sufficient, which can lead to a further 2 times reduction in the spatial overhead compared to the arrangement in Fig. 16.

The impact of the pipelining paradigm on execution speed depends on the configuration details, but two limits

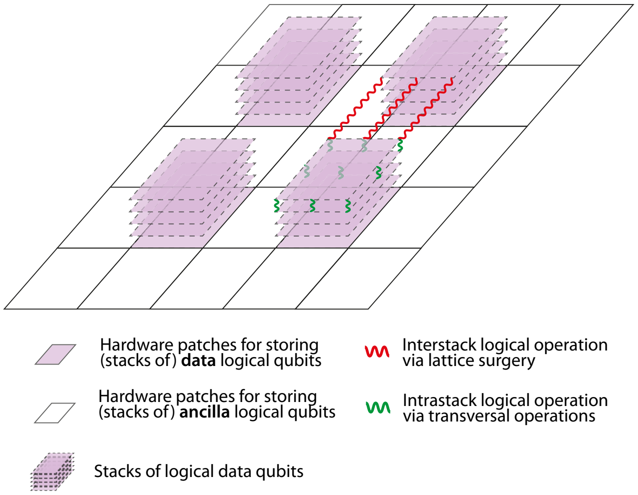

Hardware patches for storing (stacks of) **data** logical qubits
Hardware patches for storing (stacks of) **ancilla** logical qubits
Stacks of logical data qubits
Interstack logical operation via lattice surgery
Intrastack logical operation via transversal operations

FIG. 16. A possible layout for storing logical information in multiple stacks. Here each square is an array of loops that can store a stack of logical qubits. The ancilla logical qubits (stacks) do not need to be always present and thus are not explicitly shown here.

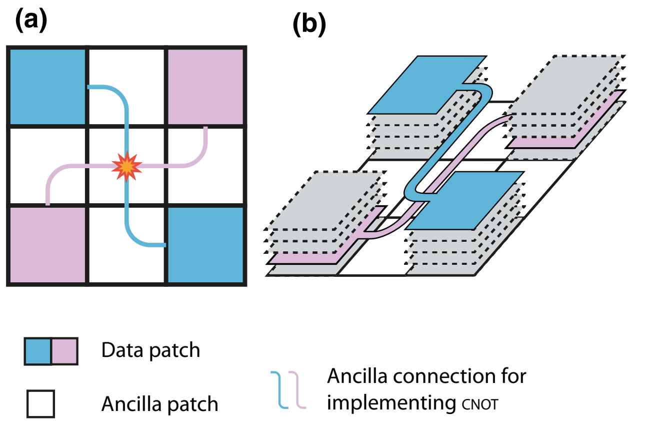


FIG. 17. Illustration of how CNOTs can fail to be parallelized when using lattice surgery (a), when using a simple 2D architecture. Here we are trying to perform CNOTs using lattice surgery between the two blue qubits and the two purple qubits. They cannot be carried out in parallel because they both need to make use of the center ancilla patch. In (b), we see this is possible with the virtual stack.

are trivial: in the extreme case in which the circuit consists of only intrastack CNOTs, we can achieve $\mathcal{O}(d)$ reduction in the time overhead by using transversal CNOTs instead of lattice surgery. On the other hand, for circuits consisting of only interstack CNOTs, pipelining enables the parallelization of CNOTs in k different layers, which can potentially reduce the CNOT depth by k times, achieving a k times reduction in the time overhead.

Beyond these simple observations, it is apparent that savings are possible by tailoring the specific circuit we want to implement towards the pipelining architecture. In the following section, we look at one of the most important subroutines for fault-tolerant computation: magic state distillation. For a given well-studied approach, we explore the exact space-time overhead saving achievable through pipelining.

IV. APPLICATION TO MAGIC STATE DISTILLATION

For full fault-tolerant quantum computation we need to perform logical gates beyond Clifford gates. One such non-Clifford gate, suitable for completing the universal set of operations, is the T gate, which is a $\pi/4$ rotation around the Z axis in the Bloch sphere: $T = e^{-i\pi/8Z}$. It can be implemented using Clifford gates and a supply of T states: $|T\rangle = T|+\rangle$ through gate teleportation shown in Fig. 18.

The circuit in Fig. 18 is a logical circuit consisting of only logical Clifford gates. Hence, provided we can implement all the Clifford gates fault tolerantly, the problem of implementing fault-tolerant T gates becomes the problem of fault-tolerant preparation of logical $|T\rangle$. This is usually done through *magic state distillation*, which is a process that consumes multiple noisy logical $|T\rangle$ and outputs fewer logical $|T\rangle$ of higher fidelity using logical Clifford operations and postselection. The standard distillation protocol consists of the concatenation of two QEC codes. One is the *base code* that we use to ensure the fault-tolerant implementation of Clifford gates, e.g., surface codes and color codes as we described above. The other one is the *distillation code*, which has transversal T gates.

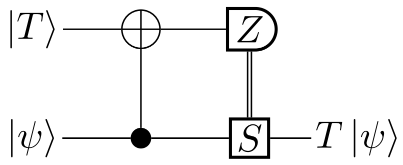


FIG. 18. Gate teleportation circuit for implementing T gate. The S gate ($S = e^{-i\pi/4Z}$) is only applied when we obtain the 1 outcome from the Z measurement. If we change the correction to applying an $S^\dagger$ gate when we obtain the 0 outcome from the Z measurement, we would be implementing $T^\dagger|\psi\rangle$ instead.

One of the most widely studied distillation codes is the 15-qubit Reed-Muller code [52], which is the smallest 3D color code. Each execution of the distillation protocol using this code will consume 15 noisy $|T\rangle$ and output a single $|T\rangle$ with higher fidelity if all checks are passed, thus it is called a 15-to-1 distillation scheme. There are two different implementations of the distillation scheme, one requires more qubits and a shallower circuit as shown in Fig. 19 [34], while the other requires fewer qubits and a deeper circuit as shown in Fig. 20 [51].

A. The space-time overhead for different pipelining schemes

To implement the 15-to-1 distillation scheme for color codes and surface codes, there are many different possible schemes, which include instances on the different extremes of the pipelining spectrum. As mentioned before, pipelining 2D codes can be essentially viewed as putting the 2D codes into stacks. Different pipelining schemes simply means a different number of logical qubits (layers) in each stack. Here we focus on three different pipelining schemes tailored to the 15-to-1 distillation scheme.

1. One layer (no pipelining)

This means only $k = 1$ logical qubit in each stack (and thus 1 qubit in each loop). We carry out the distillation process using lattice surgeries, and thus the CNOTs require $\mathcal{O}(d)$ code cycles (so constituting one of the time bottlenecks). In this case, it is natural to use the circuit in Fig. 19, which requires fewer rounds of CNOTs than Fig. 20. To implement the distillation circuit in Fig. 19, we need to store 31 logical qubits in 31 separate regions; for each of which we use the term “stack” although this is the trivial

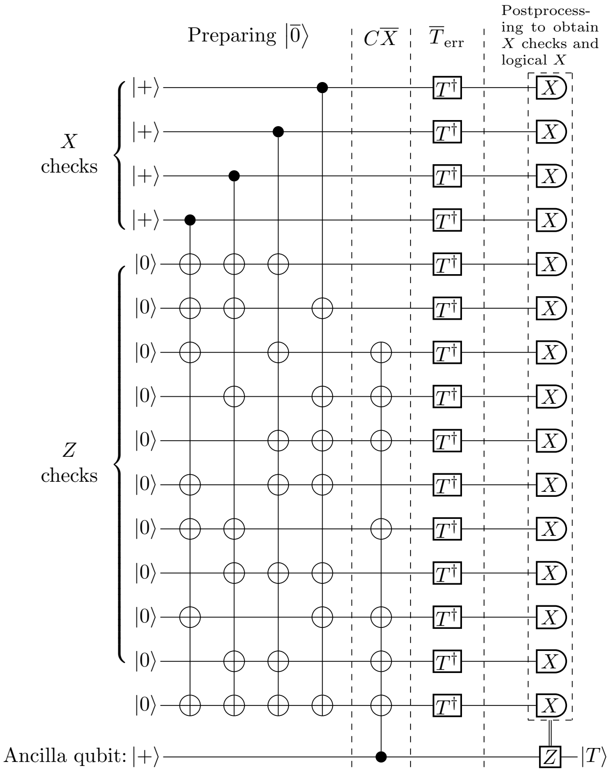

Preparing $|\bar{0}\rangle$
$C\bar{X}$
$\bar{T}_{\mathrm{err}}$
Postprocessing to obtain X checks and logical X
X checks
Z checks
$|+\rangle$
$|0\rangle$
$T^\dagger$
X
Z
Ancilla qubit: $|+\rangle$
$|T\rangle$

FIG. 19. Circuit to perform 15-to-1 magic state distillation from Ref. [34]. It starts with preparing the logical state of the distillation code $|\bar{0}\rangle$, then we prepare a Bell state between this logical qubit and the ancilla qubit $\propto |\bar{0}0\rangle + |\bar{1}1\rangle = |\bar{+}+\rangle + |\bar{-}-\rangle$. After that we implement the transversal T gates by consuming noisy $|T\rangle$ using the circuit in Fig. 18. Note this requires an additional qubit at each location marked $T^\dagger$ in the figure, 15 in total, implying 31 qubits to execute the circuit. All qubits above are encoded in the base code (none are simply "physical" qubits). At the end, we perform local X measurements to obtain the X stabilizer checks and logical X measurement of the distillation code. Any circuit runs with failed X stabilizer checks are discarded.

$k = 1$-layer limit. The resources needed for the ancilla logical qubits for implementing lattice surgeries will be of a similar order. Hence, in total we need $A \sim 50$ "stacks."

2. Ten layers

We put all the logical data qubits into one single stack ($A = 1$). In this way, all the CNOTs can be implemented transversally, and thus the circuit depth due to CNOT will not contribute much to the time cost. On the other hand, there is also a limit on the number of qubits we can fit in each stack following Eq. (4). Hence, it is natural to use the circuit in Fig. 20, which requires fewer qubits than Fig. 19. We need 5 qubits for the distillation code and 5 qubits for storing the noisy T states, thus accounting for a total of $k = 10$ logical qubits (layers) in the stack.

3. Five layers

We put the qubits for the noisy T states in one stack and the qubits for the distillation code in another. Using the circuit in Fig. 20, we have $k = 5$ logical qubits in each stack. We also need another ancilla stack for the lattice surgeries between the two data stacks. Hence, in total we need three stacks of logical qubits ($A = 3$).

By stepping through the distillation circuits in Figs. 19 and 20 and following the time costs of the different logical gates in Table I, we can derive the number of code cycles D required for the magic state distillation circuit for the different pipelining schemes. We specify the details in Sec. C, and summarize the results here in Table II. There we can see a decrease of D as we increase k due to the increased availability of transversal CNOTs. The code cycle time of the k-layer pipelining scheme is denoted as $T_{\mathrm{cycle}}^{(k)}$, which can be multiplied with the corresponding D to obtain the full time overhead of the given pipelining scheme. Combining with the space overheads (A) discussed above, we can obtain the total space-time overhead ($ADT_{\mathrm{cycle}}^{(k)}$) for different pipelining schemes for implementing magic state distillation as summarized in Table II. There we see that if all code-cycle times are similar $T_{\mathrm{cycle}}^{(1)} \simeq T_{\mathrm{cycle}}^{(5)} \simeq T_{\mathrm{cycle}}^{(10)}$, which is achievable for silicon platforms as we discuss in Sec. V, then we can achieve substantial reductions in space-time overhead: by 1 order of magnitude for the 5-layer stack and by 2 orders for the 10-layer approach.

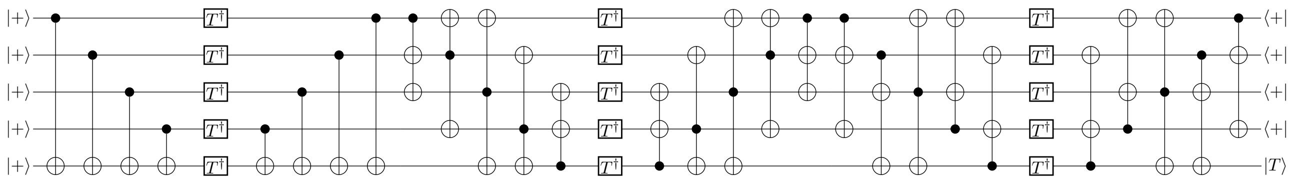


FIG. 20. Circuit to perform 15-to-1 magic state distillation with fewer qubits but a deeper circuit from Ref. [51]. In this case, the 15 noisy $|T\rangle$ are not consumed in parallel, but in three rounds. In other words, we need only to consume 5 $|T\rangle$ at one time. Thus we can use five qubits for the distillation code and five qubits for generating the noisy $|T\rangle$. For the measurement at the end, we are only postselecting the circuit runs that measure $|+\rangle$ for the first four qubits. The circuit here can be further optimized by removing some of the CNOT gates at the end that acts trivially on the postselected states of the first four qubits $|+\rangle^{\otimes 4}$, and by recompiling the CNOT gates in between different rounds of T gates. Note that all qubits above are encoded in the base code.

TABLE II. Summary of the space-time overhead of the various pipelining schemes for the magic state distillation circuit. Here $T^{(k)}_{\text{cycle}}$ is the code-cycle time for the k-layer scheme for the given code.

k: no. layers (no. qubits per loop)	A: no. stacks (no. loop arrays)	D: no. code cycles	Space-time overhead
1	50	$12d$	$600dT^{(1)}_{\text{cycle}}$
5	3	$6d$	$18dT^{(5)}_{\text{cycle}}$
10	1	d	$dT^{(10)}_{\text{cycle}}$

(a) Color codes

k: no. layers (no. qubits per loop)	A: no. stacks (no. loop arrays)	D: no. code cycles	Space-time overhead
1	50	$13d$	$650dT^{(1)}_{\text{cycle}}$
5	3	$9d$	$27dT^{(5)}_{\text{cycle}}$
10	1	$3d$	$3dT^{(10)}_{\text{cycle}}$

(b) Surface codes

B. Implication for surface-code fault-tolerant computation

In Sec. D, we provide an analysis of the space-time saving achievable beyond just the magic state distillation stage. As discussed there, since magic state distillation is often the bottleneck of fault-tolerant computation [46,47,53], the factor of time savings achieved by the magic state distillation circuit in Table II can be largely translated into the same time-saving factor for the overall fault-tolerant computation (barring some subtleties related to multiround magic state distillation). The space-saving factor is at least k when we fit k qubits into each loop (k layers in each stack), and can be more if we optimize the circuit implemented to take advantage of the increased connectivity and faster CNOTs in the pipelining architecture, as we have done for the magic state distillation circuit. The final space-time saving achieved is of course highly dependent on the circuit we try to run and the implementation details. Nevertheless, our estimates in Sec. D suggest that we should expect to achieve 1-to-2 orders of magnitude space-time saving for the *entire* fault-tolerant computation process using pipelining, just as we achieve for magic state distillation specifically.

V. PIPELINING SURFACE CODES USING SEMICONDUCTOR SPIN QUBITS

A. Implementation of a code cycle

As discussed in Sec. III B, in a surface-code cycle we want to perform one round of X checks and Z checks using the parity-check circuits in Fig. 11. There are a number of specific implementation choices that are possible in a platform where qubits are embodied by semiconductor spins. In this section we summarize one natural set of choices, so that threshold calculations can be presented in the next section. Needless to say, multiple other options exist and we refer the reader to reviews such as Ref. [54]

1. Ancilla initialization and measurements in X basis.

One of the spins in a singlet and triplet pair is used as the effective ancilla qubit to flow around the loop and interact with the data qubits, while the other spin is stored at the initialization and readout node to act as the reference qubit for readout [36,55]. The singlet and triplet pair can be read out using Pauli spin blockade, which can achieve a measurement time of $\tau_{\text{meas}} \approx 1$ µs [56–58]. This is a projective nondestructive measurement that allows us to use the measured ancilla qubits directly in the next code cycle without initialization, thus we have $\tau_{\text{init}} = 0$. We note that the field is evolving rapidly and faster measurement time may well be possible without compromising fidelity. However, it is instructive to take the time 1μs and see the consequences.

2. Single-qubit gate.

We can implement single-qubit gates using electric dipole spin resonance (EDSR), which can achieve a Hadamard ($\pi/2$ Y rotations) gate time of $\tau_{\text{H}} \approx 25$ ns [59] and similarly for $\pi/2$ Z rotations. We presently discuss a variant employing ESR.

3. Two-qubit gates.

A number of high-fidelity two-qubit gates have now been experimentally demonstrated through controlling the exchange interaction [60–62]. The specific physical operation may be $\sqrt{\text{SWAP}}$ or a controlled-rotation, but generally we can assume that it is possible to implement (directly or as a composite) a CZ gate [63] with a gate time of $\tau_{\text{CZ}} \sim 100$ ns.

4. Shuttling.

In addition to the standard gate set above for the parity check circuit, we also need to include the shuttling operation in order to implement the pipelining architecture. In semiconductor spin qubits, each step of shuttling can be carried out by tipping the electrical potential of an occupied quantum dot to transfer the spin into an adjacent quantum dot adiabatically. Various modes such as "bucket brigade" [64] and "conveyor" [65] have been explored, and high fidelity shuttling steps on the timescale of nanoseconds is expected to be possible [12,66–69] with the corresponding shuttling speed being tens of $\mathrm{m\,s^{-1}}$. Hence, the shuttling step will not be the rate-limiting step in the pipeline. In Ref. [11], it is estimated that a qubit separation of $\sim 10\ \mu\mathrm{m}$ is needed for the local integration of various classical control electronics for the implementation of surface code. Such a qubit separation would correspond to a loop with an edge length of approximately $7\ \mu\mathrm{m}$ in the pipelined surface-code architecture in Fig. 12. Assuming a shuttling speed of approximately $25\ \mathrm{m\,s^{-1}}$, the time required for shuttling through one round of the loop is then $\tau_{\mathrm{sh}} \sim 1\ \mu\mathrm{s}$.

In this case, the rate-limiting step is the measurement with a processing time of

$$\tau_{\max} = \tau_{\mathrm{meas}} \sim 1\ \mu\mathrm{s}.$$

Using Eqs. (7) and (8), the time needed to implement the parity-check circuits for one code cycle for the data and ancilla pipelines are

$$T_{\mathrm{circ}}^{\mathrm{data}} = 3\tau_{\mathrm{sh}} + 8\tau_{\mathrm{CZ}} + 2\tau_{\mathrm{H}} = 3.85\ \mu\mathrm{s}, \qquad (11)$$

$$T_{\mathrm{circ}}^{\mathrm{anc}} = \tau_{\mathrm{sh}} + 4\tau_{\mathrm{CZ}} + \tau_{\mathrm{meas}} = 2.4\ \mu\mathrm{s}. \qquad (12)$$

In this scenario the data pipeline is the rate-limiting pipeline, which determines the code-cycle time:

$$T_{\mathrm{cycle}} = \max(T_{\mathrm{circ}}^{\mathrm{data}}, T_{\mathrm{circ}}^{\mathrm{anc}}) = 3.85\ \mu\mathrm{s}.$$

We also need to know the maximum number of qubits we can fit into the pipeline without qubit collision. Following Eq. (10), we have $K_{\mathrm{loop}} \approx 2$, i.e., we can only fit two qubits in each loop, which is not enough for implementing any of the pipelining magic state distillation schemes discussed in Sec. IV. The most straightforward way to solve this is by adding more measurement devices as discussed in Sec. II B. If we equip each loop with four measurement devices such that the rate-limiting step time is now $\tau_{\max} = \tau_{\mathrm{meas}}/4 = 0.25\ \mu\mathrm{s}$, we are able to fit in five qubits in each loop following Eq. (10), which is enough for carrying out the five-layer magic state distillation scheme in Sec. IV. Note that the code-cycle time T_{cycle} remains unchanged this way.

However, there is a much more efficient scheme to fit in more qubits as discussed in Sec. E. There we deviate from the steady-flow scheme and reduce the pipelining qubit time gap below $\tau_{\max} = \tau_{\mathrm{meas}}$. No measurement steps occur in the data pipeline, thus this has no effect on the data pipeline. On the other hand, we need to add buffering to the ancilla pipeline since it contains measurement steps. If we add enough measurement devices such that the data pipeline remains the rate-limiting pipeline, the code-cycle time T_{cycle} *remains unchanged*. In this way, we can fit 5 qubits using only two measurement devices in each loop. Hence, we need only two measurement devices per loop for carrying out the five-layer magic state distillation scheme, instead of five measurement devices mentioned above. With three measurement devices in each loop, we can fit up to ten qubits, which enables us to carry out the superior ten-layer distillation scheme.

Using the reduced time gap, if we *do not add any measurement devices*, then the more qubits we fit in, the more buffering we need to add to the ancilla pipeline, which may then become the rate-limiting pipeline and lead to an increase in T_{cycle}. As discussed in Sec. E, for our parameter regime, it is more efficient to add buffering into the data pipeline instead, which will increase the minimum cycling period $T_{\mathrm{loop}}^{\min}$ and thus increase the number of qubits we can fit in using Eq. (4). As shown in Sec. E, to implement the five-layer ($k = 5$) and ten-layer ($k = 10$) distillation schemes without adding measurement devices, we need a code-cycle time of $T_{\mathrm{cycle}} = 6\ \mu\mathrm{s}$ and $T_{\mathrm{cycle}} = 10.5\ \mu\mathrm{s}$, respectively.

As mentioned before, the space-time overhead is simply the product of the number of loop arrays required (A), the number of code cycles required (D,) and the code-cycle time (T_{cycle}). The space-time overhead savings achieved by applying pipelining to the magic state distillation circuit in a semiconductor spin-qubit platform is summarized in Table III. There we see that without adding any measurement devices, the best we can achieve is an 100 times reduction in the space-time overhead using the ten-layer distillation scheme. If we add two more measurement devices per loop, we can achieve a 200 times reduction in the space-time overhead using the ten-layer distillation scheme. The five-layer scheme does not offer any advantages over the ten-layer scheme in the cases that we consider. The detailed space-time savings expected for the whole computation are discussed in Sec. D for the cases in which additional measurement devices are added, as necessary, such that the code-cycle time stays the same (the implications of using a single measuring device are also noted).

As mentioned above, we consider the case that the single-qubit gates are carried out via EDSR; this can necessitate the use of gate devices involving micromagnets placed close to the quantum dots. If such devices are placed on the shuttling loop, then our qubits may undergo unwanted rotations every time they pass these structures. To the extent that such rotations are deterministic, it may be possible to account for them within the algorithm we are implementing. We might also tackle such an issue at the architectural level by having these single-qubit gate devices branching off the shuttling loop (similar to the readout devices in earlier figures), so that the qubits do not

TABLE III. Summary of the space-time saving brought by pipelining for the surface code magic state distillation circuit implemented in the semiconductor spin-qubit platform. The time overhead is simply DT_{cycle} and the factor of time saving is calculated using the no-pipeline ($k = 1$) scheme as the baseline. Similarly, the space-time overhead is ADT_{cycle} and the factor of space-time saving is calculated using the no-pipeline ($k = 1$) scheme as the baseline.

No. of qubits per loop: k	No. of loop arrays: A	No. of code cycles: D	No. of meas. devices per loop: m	Pipelined time gap: τ_{gap} (μs)	Code cycle time: T_{cycle} (μs)	Time saving factor	Space-time saving factor
1	50	$13d$	1	1	3.85	1	1
5	3	$9d$	1	0.4	4.8	1.2	19
			2	0.28	3.85	1.4	24
10	1	$3d$	1	0.32	8.55	2	100
			3	0.125	3.85	4.3	200

go near these gate devices unless we want to apply gates on them. Since we expect the speed of the shuttling step to be relatively fast compared to the other operations in the pipeline, taking this detour need not have much effect on the arguments we make above.

As an alternative to the EDSR route, a platform might employ electron-spin resonance (ESR) to implement the single-qubit gates instead. As shown in Sec. F, using ESR means a slower code cycle without pipelining $T^{(1)}_{\text{cycle}} = 5.9$ μs. However, an advantage to the slower code cycle is that we can actually fit more qubits into the pipeline without adding buffering and measurement devices. With the exact same hardware (without adding any measurement devices), we can carry out the ten-layer distillation scheme with a code cycle of 9.15 μs, achieving a 140 times reduction in the space-time overhead compared to the unpipelined scheme.

B. Threshold calculations

In order to carry out parity checks for quantum error-correction codes like the surface code, we need to use circuits like those in Fig. 11 with the understanding that all components are at some level faulty. There exists a threshold for the error rate of these faulty physical components, below which the logical error rate of the code can be suppressed to any desired level by suitably scaling up the code. Hence, this error threshold implies a target component fidelity that experimentalists aim to surpass in order to scale up their system through error correction. The error threshold is different for different codes and is highly dependent on the exact implementation of the stabilizer check process. For the surface code with components suffering depolarizing noise, various stabilizer check implementations have been studied and typically yield an error threshold of 0.5% to 1% [70].

In this section, we perform an error-threshold simulation for the implementation of surface-code pipelines using the semiconductor spin qubits as outlined in Sec. V A. The noise model we use for the standard gate components are as follows:

(a) Measurement, initialization, and CZ gates experience fully depolarizing noise with probability p.

(b) Single-qubit gates experience completely depolarizing noise with probability $p/10$.

This is consistent with some of the most widely used noise models for the standard gate set, so that our result can be compared to threshold results in other studies. On top of these standard gate noise sources, we also consider the noise due to the shuttling process itself, and the noise due to placing multiple qubits in the same loop.

1. Shuttling dephasing

A leading source of errors for shuttling semiconductor spin qubits is phase rotation due to inhomogeneity of effective g factors across different quantum dots in the loop [10,66,69]. We can think of it as some deterministic phase rotation on the qubit after one round around the shuttling loop, which can, in principle, be corrected by applying a calibrated inverse rotation at the end of each round. Phase rotations commute with CZ, thus as long as we correct it after each round so that the phase rotation on the data qubits does not go through the Hadamard gate and become bit rotation, ideally the shuttling noise can be perfectly removed. However, as time goes on, some shuttling loops may go out of calibration and there will be some remnant phase rotation after each round. If we *twirl* this remnant noise by conjugating the circuit with random Pauli gates [71], the noise will effectively become pure *dephasing noise* on the qubits with the error probability p_{sh} in each code cycle [10]. One must note, however, that the additional single-qubit Pauli gates used for twirling will also introduce single-qubit depolarizing noise with the probability $p/10$.

2. Shuttling leakage.

In silicon spin qubits, the mixture of two *valley orbitals* (equivalent minima) in the bulk silicon conduction band gives rise to the ground orbital that our spin qubit lives in and an excited orbital. The amount of mixing between the two valley orbitals and the resultant energy separation (*valley splitting*) between the ground orbital and the excited orbital is influenced by the heterointerface at the quantum dot. Due to variations of these heterointerfaces from one quantum dot to another, we see variations in the ground orbital mixture from one dot to another. Such a variation means that as we shuttle the spin qubit from one dot to another, the spin qubit may interact with the excited orbitals in the two dots, resulting in a state that has qubit information "leaked" into the excited orbital. Such a leaked component is analogous to a spin qubit with a miscalibrated detuning, thus its interaction with the other qubits will also be erroneous [72]. The leakage rate p_{leak} is highly dependent on the system's structure, and this is the free parameter we sweep in our modeling. A second key parameter is the intervalley relaxation time, i.e., the typical duration for which a leaked state persists before relaxing back; we would wish this to be fast. This relaxation time can go below 100 ns [69] and even reach 10 ns [73,74], which is much smaller than the code-cycle time we consider (μs scale). Hence, we expect the leaked qubits can be restored back into the computational subspace within each code cycle. In Ref. [69] the authors argue that valley excitation and relaxation during shuttling will effectively lead to dephasing noise. Here we will make the pessimistic assumption of an even *more damaging* model for the shuttling leakage errors: all qubits will leak with the probability p_{leak} at the start of each code cycle, and if leakage happens, the leaked qubit *completely depolarizes all qubits that it interacts with* and will become completely depolarized itself at the end of the code cycle.

3. Unwanted intraloop interaction.

Another possible noise source is the undesired long-range dipole-dipole interaction among qubits in the same loop. The strength of the dipole-dipole interaction is given by $J = \mu_0 g_e^2 \mu_B^2 / 4\pi r^3$ with μ_0, g_e, μ_B, and r being the vacuum permeability, electron g factor, Bohr magneton, and the distance between the spins, respectively. As discussed in the last section, a reasonable assumption for the length of one edge of the loop is 10 μm. Even if we have approximately 100 qubits in the loop, it would mean a qubit spacing of $r \sim 0.1$ μm, translating into $J \sim 100$ Hz. Such a noise strength is negligible compared to the other energy scales (on the order of MHz to GHz) in the scenario.

The error locations in the X parity check circuit is summarised in Fig. 21, similarly for the Z checks.

The error threshold we obtain for purely standard gate noise (without any of the shuttling noise) is 0.74%, which agrees with previous well-known results [23]. Now suppose we manage to achieve a gate error rate of $p = 0.5\%$, which is below threshold and has been demonstrated in state-of-the-art experiments [61,62]. Then the level of shuttling dephasing noise we can tolerate is found from the threshold plot in Fig. 22(a), namely $p_{\text{sh}} = 0.36\%$. As mentioned, the shuttling dephasing noise is due to equipment drifting out of calibration and thus should be at a much lower level than the gate noise. Hence, the shuttling dephasing noise should not pose a problem for our implementation. This is consistent with the result in Ref. [10].

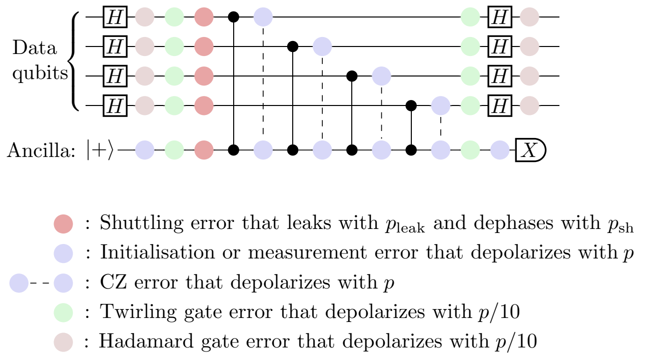
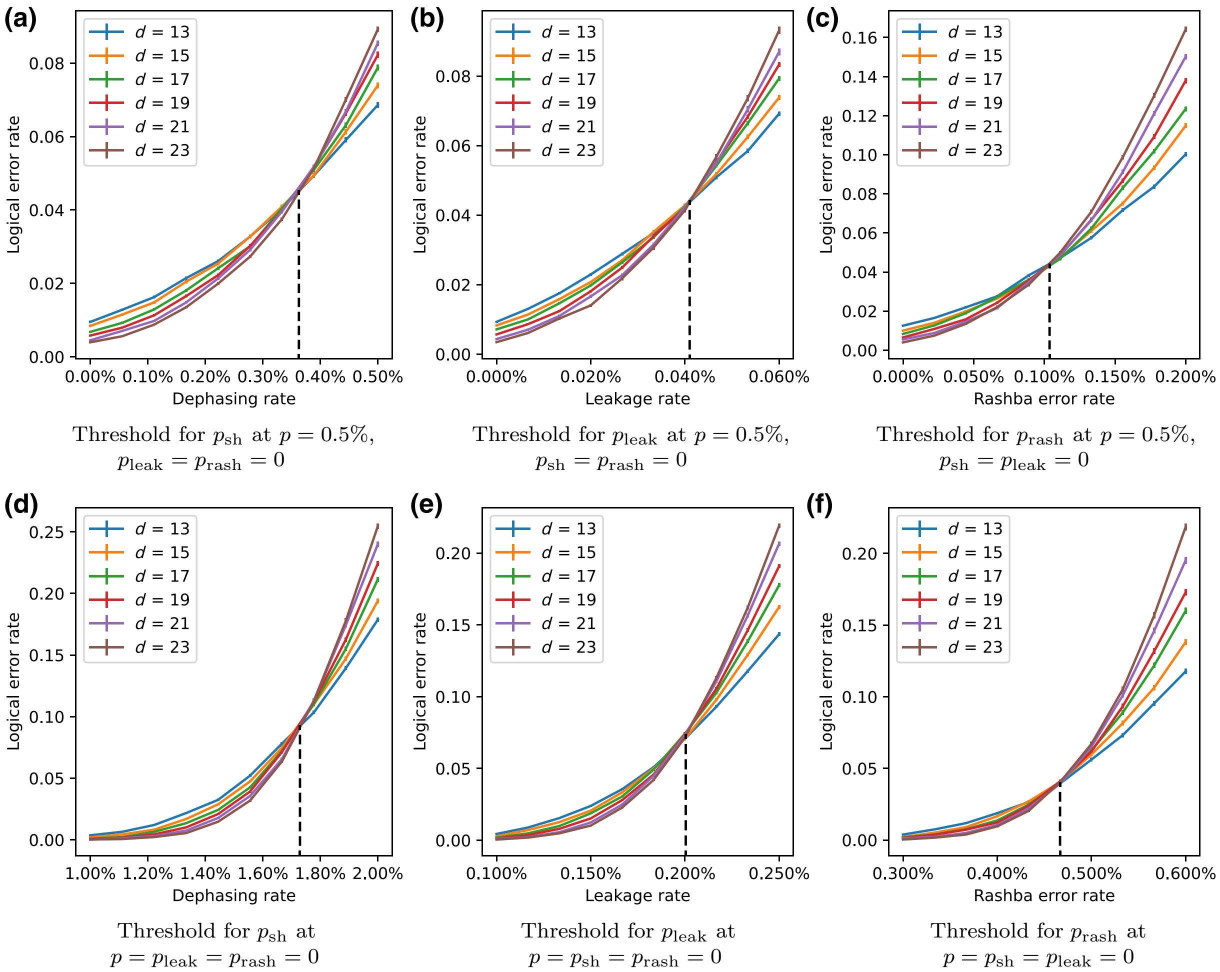

Data qubits
H
H
H
H
Ancilla: |+⟩
X
● : Shuttling error that leaks with p_{leak} and dephases with p_{sh}
● : Initialisation or measurement error that depolarizes with p
●- - ● : CZ error that depolarizes with p
● : Twirling gate error that depolarizes with $p/10$
● : Hadamard gate error that depolarizes with $p/10$

FIG. 21. Diagram showing the different error locations in the X-check circuits in our error model.

Holding the gate error rate at $p = 0.5\%$, now let us shift our focus to the leakage noise. In the threshold plot in Fig. 22(b), we obtain a leakage threshold of $p_{\text{leak}} = 0.04\%$ in the presence of gate noise. Reference [69] has argued that such a shuttling error rate is achievable when shuttling across tens of μm at a speed of tens of m s^{-1}, which is exactly the physical setting that we are considering in Sec. V A. Note that the achievable error rate given in Ref. [69] includes shuttling dephasing error, thus our achievable leakage error rate will be even lower. The leakage error model in Ref. [69] is also less damaging than the error model we assume, hence our figures provide a pessimistic lower bound of the threshold for the model in Ref. [69].

If we can further suppress our gate noise below $p = 0.5\%$, we can further improve our tolerance against the dephasing noise and leakage noise due to shuttling. In the extreme of zero gate noise ($p = 0$), we can tolerate a level of dephasing noise $p_{\text{sh}} = 1.75\%$ and a level of leakage noise $p_{\text{leak}} = 0.20\%$. Overall, we see that the dephasing noise in shuttling is less damaging than standard gate noise, while the leakage noise, even assuming a very damaging noise model, is just 1 order of magnitude more damaging than the standard gate noise. Hence, they should not impose any fundamental limitation on implementing shuttling-based architectures in semiconductor spin qubits. It would be interesting to perform a similar analysis for trapped-ion shuttling, for example using the error model presented in Refs. [19,21].

A further noise mechanism worth considering is the Rashba spin-orbit interaction due to the shuttling of the spin qubits. This leads to coherent rotations of the qubits



Threshold for $p_{\rm sh}$ at $p = 0.5\%$, $p_{\rm leak} = p_{\rm rash} = 0$

Threshold for $p_{\rm leak}$ at $p = 0.5\%$, $p_{\rm sh} = p_{\rm rash} = 0$

Threshold for $p_{\rm rash}$ at $p = 0.5\%$, $p_{\rm sh} = p_{\rm leak} = 0$

Threshold for $p_{\rm sh}$ at $p = p_{\rm leak} = p_{\rm rash} = 0$

Threshold for $p_{\rm leak}$ at $p = p_{\rm sh} = p_{\rm rash} = 0$

Threshold for $p_{\rm rash}$ at $p = p_{\rm sh} = p_{\rm leak} = 0$

FIG. 22. Threshold plots for the dephasing errors ($p_{\rm sh}$), leakage errors ($p_{\rm leak}$), and the Rashba error ($p_{\rm rash}$) due to shuttling when implementing surface codes in semiconductor spin qubit using shuttling-based architectures. Note that in order to investigate the tolerance of the surface code against a given specific type of shuttling noise, the other types shuttling noise are turned off in our simulation. Each noise type is evaluated both for a gate error rate p of zero, and for $p = 0.5\%$.

as they go around the loop, which can be characterized and corrected analogously to the shuttling dephasing [75]. We can again use twirling to decohere any uncompensated residual noise; in this case however it will give rise to stochastic Pauli X and Y noise. For simplicity, we assume X and Y errors occur with the same probability $p_{\rm rash}/2$ each time we shuttle the qubit around the whole loop. In the context of performing X stabilizer check, we label the error locations of the Rashba errors in Fig. 23, with the error strength in each location adjusted according to the corresponding shuttling length in that section of the circuit. Note that there we show only the shuttling errors due to Rashba interaction. The corresponding thresholds for the shuttling error due to Rashba interaction are shown in Fig. 22 where the threshold is $p_{\rm rash} = 0.1\%$ with gate noise being $p = 0.5\%$, and $p_{\rm rash} = 0.46\%$ in the absence of gate noise $p = 0\%$. These thresholds are lower than those for shuttling dephasing noise because we are using CZ gates in our parity-check circuit as shown in Fig. 23 along which X and Y errors can propagate but Z noise cannot. A higher threshold can be obtained if we are using CNOT gates in the parity-check circuits instead. The thresholds given above effectively determine the required accuracy for the coherent correction we need to perform. They are of the similar order as the shuttling dephasing noise we study above and should not present any fundamental roadblocks for our implementations.

Before concluding this section, it is worth pointing out that instead of using single electron spins as our qubits, it is also possible to use singlet-triplet qubits [76] or exchange-only qubits [77] in our pipelining architectures. An advantage is that faster single-qubit gates could then be available. Furthermore, if we are transporting all the constituent spins together, some of the shuttling noise we mention above may have trivial effects since these qubits live in decoherence-free subspaces [78]. However, there

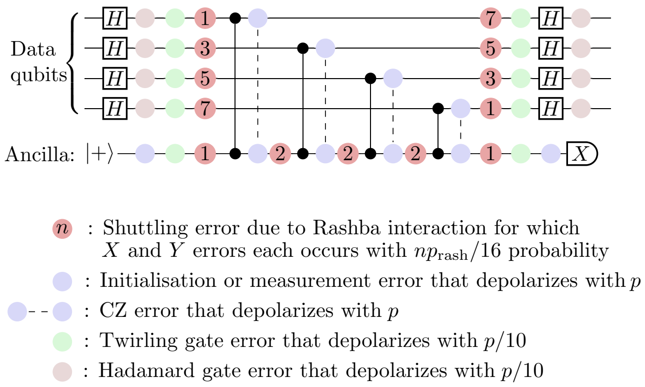

Data qubits
Ancilla: $|+\rangle$
H
1
3
5
7
2
X
n : Shuttling error due to Rashba interaction for which X and Y errors each occurs with $np_{\mathrm{rash}}/16$ probability
● : Initialisation or measurement error that depolarizes with p
●- - ● : CZ error that depolarizes with p
● : Twirling gate error that depolarizes with $p/10$
● : Hadamard gate error that depolarizes with $p/10$

FIG. 23. Diagram showing the different error locations in the X-check circuits with the shuttling errors coming from Rashba interaction.

are additional challenges in shuttling and additional noise mechanisms involved like leakage from the qubit subspace. Hence, the performance and trade-offs for such qubit encodings in the pipelining architecture can be an interesting future direction to explore.

VI. APPLICATION TO QUANTUM ERROR MITIGATION

Besides quantum error correction, pipelining can also find an application in recently proposed purification-based quantum error mitigation using multiple copies of the noisy state, which is called *error suppression by derangement* [6] or *virtual distillation* [7]. In these methods, the ideal state we want to prepare is some *pure state* ρ_0, but due to the noise in the circuit, we have the noisy state ρ instead. If the dominant eigenvector of ρ is ρ_0, then we can construct a M^{th}-degree purified state

$$\rho_{\text{pur}}^{(M)} = \frac{\rho^M}{\mathrm{Tr}(\rho^M)},$$

which has removed up to the $(M-1)^{\text{th}}$-order errors in the noisy state, i.e., the errors in the state is exponentially suppressed with the increase of the degree of purification M. If the dominant eigenvector of ρ is not ρ_0, the error suppression still works well as long as it is not too far from ρ_0, but now there is an upper bound on the amount of noise it can remove in the name of *noise floor* [7] or *coherent mismatch* [6]. In some practical cases, such coherent mismatch is shown to be small [79].

Now in practice, instead of trying to obtain the ideal state, we are often interested in the expectation value of some observable O on the ideal state ρ_0: $\mathrm{Tr}(O\rho_0)$. Then instead of trying to construct the purified state $\rho_{\text{pur}}^{(M)}$, we can construct the "purified" expectation value $\mathrm{Tr}(O\rho_{\text{pur}}^{(M)})$:

$$\mathrm{Tr}(O\rho_{\text{pur}}^{(M)}) = \frac{\mathrm{Tr}(O\rho^M)}{\mathrm{Tr}(\rho^M)}. \tag{13}$$

Hence, all we need do is to estimate $\mathrm{Tr}(O\rho^M)$ for some observable O, and obtain $\mathrm{Tr}(\rho^M)$ using the same method but with $O = I$. Then $\mathrm{Tr}(O\rho_{\text{pur}}^{(M)})$ is obtain by dividing $\mathrm{Tr}(O\rho^M)$ by $\mathrm{Tr}(\rho^M)$.

As shown in Refs. [6,7], $\mathrm{Tr}(O\rho^M)$ can be rewritten as

$$\mathrm{Tr}(O\rho^M) = \mathrm{Tr}(O^{(1)}\overline{C}_M\rho^{\otimes M}), \tag{14}$$

where $\rho^{\otimes M}$ denotes M copies of the noisy state ρ, $\overline{C}_M$ is the cyclic permutation operators among these M copies, and $O^{(1)}$ means the operator O is *only applied to the first copy*. Hence, the target expectation value $\mathrm{Tr}(O\rho^M)$ can be obtained by preparing M copies of the same noisy state $\rho^{\otimes M}$ and measuring the product of the observable $O^{(1)}$ and the copy-cyclic-permutation operator $\overline{C}_M$.

Without loss of generality, we can assume O to be Pauli since any observable of interest can be decomposed into a linear sum of Pauli operators. Hence, O can be written as a tensor product of *single-qubit Pauli operators* $\{G_i\}$ acting on the different qubits in the first copy:

$$O^{(1)} = \bigotimes_{i=1}^{N} G_i^{(1)}.$$

Here N is *the number of qubits in each copy* of ρ, and i is the labeling of these qubits. We can measure $O^{(1)}$ simply by measuring G_i on the i^{th} qubit of the first copy.

On the other hand, the M-copy cyclic permutation operator $\overline{C}_M$ is equivalent to applying the M-qubit cyclic-permutation operator C_M transversally to the corresponding physical qubits in each of the copies:

$$\overline{C}_M = C_M^{\otimes N}.$$

Hence, the observable $O^{(1)}\overline{C}_M$ can be decomposed into the following tensor product:

$$O^{(1)}\overline{C}_M = \bigotimes_{i=1}^{N} G_i^{(1)} C_M,$$

which *can be measured transversally*.

Consider using a conventional 2D nearest-neighbor qubit architecture, in which different copies are stored in different 2D qubit arrays (for example, adjacent large regions of entire complete system's array). If we want to measure a multicopy transversal operator like $O^{(1)}\overline{C}_M$, w need to interlace the qubit arrays corresponding to different copies using a number of swaps gates that are

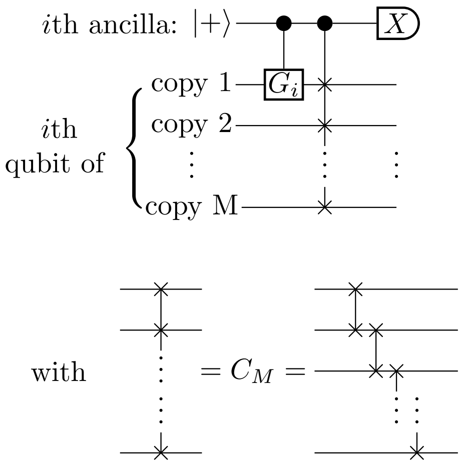


FIG. 24. Measurement circuit in each loop for purification-based QEM.

proportional to the size of the qubit array. This would be extremely costly—indeed it is for an analogous reason that in 2D topological code, it is conventionally more efficient to employ lattice surgery rather than transversal CNOTs. Instead we would ideally use the looped-pipelining architecture proposed in the present paper, with all the M-qubit arrays in the same loop array (i.e., M qubits in each loop structure). Then the transversal measurement of the M-copy operator $O^{(1)}\overline{C}_M$ can simply be carried out by measuring the M-qubit operator $G_i^{(1)}C_M$ in the i^{th} loop.

To be more specific, the full process of implementing purification-based QEM using the pipelining scheme is now as follows: using the qubit-array pipeline discussed in Sec. II C, we can have M qubits in each loop and process them in the same way to prepare M noisy copies of our target quantum states. However, we require one further ancilla qubit in each loop, effectively adding another array of ancilla qubits on top of the M noisy copies. The i^{th} loop contains the i^{th} qubit of each copy of ρ as well as the i^{th} ancilla, for a total of $M+1$ qubits. We implement the circuit in Fig. 24 in each loop. The i^{th} ancilla measures the observable $G_i^{(1)}C_M$, and the *product of the measurement results of all ancilla* measures the observable $O^{(1)}\overline{C}_M$ on the M copies of state $\rho^{\otimes M}$, giving us the expectation value in Eq. (14), which we can then use to obtain the error-mitigated expectation value in Eq. (13).

As discussed before, in many hardware platforms, shuttling is an essential component for scaling up. In that case, if we scale up using a loop array as discussed in Sec. II C, we can pipeline extra qubits at zero spatial overhead. Moreover, the pipelining time cost is also negligible if the computation is deep as seen from Eq. (3). Hence, purification-based QEM can be carried out *at almost zero space-time overhead* in the pipelining architecture. In contrast, conventional architectures need multiple copies of the same machines and long-range interactions.

Of course, the above arguments assume that the number of qubit arrays that we pipeline for the error mitigation is not limited by the qubit collision problem mentioned in Sec. II B. This is not a serious limitation since in practice, due to the noise floor and the sampling overhead, it would be unlikely that one would wish to go beyond fourth-degree purification [6,7], which can be implemented using only three qubit arrays consisting of two noisy copies and one ancilla with the help of state verification [80,81]. For the main computation, we need only to compute on two noisy copies, which means two qubits per loop, which should not pose any significant challenges as we have seen in the example of pipelining surface codes. At the error-mitigation stage we need to add in the ancilla qubit to each loop, but any additional buffering time here can be expected to be insignificant since the main computation is presumably much deeper. Note that if we are interested in only second-degree purification, it can be carried out by performing transversal measurements on two noisy copies without using any ancilla [7].

VII. CONCLUSION

Several proposed platforms for scalable quantum computing aim to employ physical shuttling of qubits in order to optimally space out the components of the technology, thus making room for classical control systems, permitting heat dissipation, avoiding crosstalk, and so on. Shuttling structures are typically implemented or envisaged as *linear* “highways” that connect components in a sparse grid. In this paper, we have examined an alternative realization where one stores a sparse qubit array in an array of shuttling *loops*. The key merit of this formulation is that in such a loop-array structure, we can put additional qubits into each loop, to form a local stream of qubits where each follows the same trajectory as the first qubit. Thus one can store and process multiple qubit arrays using the same architecture *without adding any additional hardware*.

With k qubits in each loop, we have increased the qubit density (reduced the spatial overhead) by a factor of k compared to the other shuttling-based schemes while still retaining all of their advantages for scaling up. This loop array can be viewed as a pipeline for processing a stack of k-qubit arrays layer by layer, i.e., a *qubit-array pipeline*. Furthermore, by allowing interactions between qubits within the same loop, we can connect between different layers of the stack of qubit arrays. In this way, we can perform computations on an effectively 3D qubit lattice using a 2D loop array whose hardware requirements are similar to those that are needed *in any case* for 2D shuttling-based platform. The height of the resultant 3D qubit lattice is given by the number of qubits in the loop k and thus cannot be increased indefinitely. We have shown that significant improvements in both NISQ and fault-tolerant applications can be achieved via our architecture despite this limited height.

Processing multiple topological logical qubits in one loop array using pipelining will reduce the space overhead of logical qubits. Furthermore, intraloop qubit interactions enable transversal CNOT among logical qubits, which can significantly speed up most fault-tolerant applications since CNOT is usually one of the rate-limiting steps. We have outlined a possible implementation of the surface code pipeline for semiconductor spin qubits, and we estimate that one could achieve a reduction by a factor of 100 in the space-time overhead for magic state distillation compared to a generic shuttling-based architecture. Moreover, a reduction by a factor of 200 in the space-time overhead can be achieved if we add two more measurement devices to each loop. Achieving such factors would of course require that certain bottlenecks are avoided, and we discuss these caveats. When considering the full process of fault-tolerant quantum computation, we have estimated that one to two orders of space-time saving can be achieved by our architecture depending on the implementation details.

For the case of silicon spin-qubit devices, we have shown that one can easily fit ten qubits in each loop for implementing the surface code, which is enough for carrying out magic state distillation using only transversal CNOTs. Furthermore, we perform surface-code threshold simulations using several models for the noise, which may be introduced by shuttling. We concluded that the permissible noise levels are well within those that are expected for spin shuttling.

In the final section of our analysis, we considered the utility of the looped-pipeline architecture for an application that is more relevant to NISQ-era devices: certain powerful purification-based quantum error-mitigation methods that have recently been proposed. These mitigation methods require two or more entire copies of the computer's output, and then these copies need to be interacted qubit by qubit; this is potentially costly and unwieldy for a canonical 2D grid system. We note that a natural solution is to arrange that the i^{th} qubit of each copy of the state are stored within the i^{th} loop of the architecture. In this way, we can perform the circuit for purification-based QEM at almost zero space-time overhead compared to the unmitigated circuit.

There are many other possible applications using the pipelining architecture beyond what we have studied. For example, the pipelining architecture is a natural choice for concatenating other codes on top of a topological code (indeed the case of magic state distillation, which we studied is essentially an instance of this). It would be interesting to see if such code concatenation would bring advantages in fault-tolerant memory or storage. It might also be possible to connect the boundaries of the code patches in the same pipeline to construct a long strip of folded code that might be robust against biased noise [82].

We focused on an application of the qubit-array pipeline where qubits form layers of a virtual 3D stack. Here the majority of gate operations are interloop and remain within each 2D qubit array. There is a high degree of regularity between different qubit arrays in the pipeline, e.g., the stabilizer checks for the 2D topological code pipeline and the noisy copy preparation for the QEM pipeline. Given this scenario we were able to analyze the qubit movement scheduling and time cost of the whole pipeline by studying only one of the arrays. However, relaxing these regularities may be perfectly possible in both electron spin-qubit devices and ion traps; generalizations would include performing different operations on distinct qubit arrays in the same pipeline, having more frequent scheduling of transversal operations between different qubit arrays, or varying either the number of qubits in the loops or their cycling frequency. Given some of these capabilities, a particularly interesting application that can be explored in the future is the implementation of 3D codes. Note that one might need to use just-in-time decoding [83,84] to overcome the limited height of the 3D lattice in the pipelining architecture. Going beyond 3D codes, there has been recent work on implementing more general LDPC codes using our pipelining architecture [85]. When the end-to-end process of fault-tolerant computation for these more general LDPC codes is fully worked out, it will be interesting to perform more detailed optimizations and performance analysis for their pipelining implementations.

Note added in proof.—For Eq. (5) (and its derivation), removing the "+1" leads to a closer answer to the practical implementation of the proposed schemes.

ACKNOWLEDGMENTS

The authors are grateful to Benjamin Brown, Hamza Jnane, Bálint Koczor, Markus Müller, and Armands Strikis for helpful comments. We thank Michael Fogarty for discussions about possible implementation methods for full 3D codes, which we intend to explore comprehensively in a subsequent study. The authors acknowledge support from Innovate UK project 133997: Multicore NISQ Processors on Silicon Chips, from the EPSRC QCS Hub EP/T001062/1, from EU H2020-FETFLAG-03-2018 under Grant Agreement No. 820495 (AQTION) and from the IARPA funded LogiQ project. Z.C. is supported by the Junior Research Fellowship from St John's College, Oxford.

APPENDIX A: SYNCHRONIZING AN ARRAY OF SHUTTLED QUBITS

Let us again consider a large qubit array stored in a loop array similar to Fig. 4(a), which can also be viewed as a qubit-array pipeline processing just *one* qubit array. The looped qubit pipelines in the array are engineered to have the *same cycling period* at every round so that they can work in synchronization, but they are *at different phases* to allow the right qubit interactions to happen. The cycling period of the qubit-array pipeline is determined by the constituent looped qubit pipeline with the longest period, and other loops will synchronize with this rate-limiting looped qubit pipeline through buffering. Such a local buffering (idling) stage is no different from those required for the gate scheduling in the other 2D architectures.

We can define a *global clock cycle* that takes the time of one cycling period, within which all the qubit streams flow around the loops exactly once, enabling us to implement single-qubit operations on any qubits and two-qubit operations between any neighboring qubits for all the qubit arrays in the pipeline. Through multiple global clock cycles, we can implement any unitaries we want on the qubit arrays. Since the qubit streams in the loop array are flowing in synchronization like an array of meshed cogwheels, we can study the time taken for the qubit-array pipeline by just looking at the time needed for any of its constituent looped qubit pipelines. The initialization at the beginning and the measurement at the end for the whole qubit-array pipeline also have all of its constituent looped qubit pipelines working in perfect synchronization, thus the same argument applies. Hence, when we want to perform any *unitary* circuits on the qubit arrays using such a shuttling architecture (the qubit-array pipeline), the time required is directly given by the time needed for any of its constituent looped qubit pipelines adding the buffering time needed for synchronizing the cycling period.

Going beyond unitary circuits, we may want to perform midcircuit nondestructive measurement on some qubits in the arrays. Suppose we want to perform a midcircuit measurement on a given qubit pipeline within the array between the n^{th} and the $n+m^{\text{th}}$ global clock cycle. The qubit stream in the corresponding qubit pipeline needs to break away from the global clock cycle after the n^{th} round, flow to the measurement device for the operation and then flow back to the outer loop to rejoin the $n+m^{\text{th}}$ global clock cycle. If the measurement operation described here (including the additional shuttling) takes longer than $m-1$ global clock cycles, then the measured qubit pipeline would become the rate-limiting pipeline, and we need to add buffering on top of the $m-1$ global clock cycles for the other qubits in the array to synchronize with the measured qubit. Otherwise, the midcircuit measurement of the given qubit would not affect the global schedule of the other qubits in the circuits, and thus it can be carried out without additional time cost. We can always add more measurement devices as discussed in Sec. II B to speed up the measurement process and avoid the additional time cost. All the discussion of midcircuit nondestructive measurement above also applies to destructive measurement plus reinitialization.

APPENDIX B: COLOR-CODE PIPELINE

Similar to the surface-code pipeline in Sec. III B, we look at the data pipeline and the ancilla pipeline separately. A possible pipeline is shown in Fig. 25, in which we carry out the X and Z checks in a strictly sequential manner. The time required for processing one qubit array is given by

$$T_{\text{circ}}^{\text{data}} = 2\tau_{\text{sh}} + 12\tau_{\text{CZ}} + 2\tau_{\text{H}}, \tag{B1}$$

$$T_{\text{circ}}^{\text{anc}} = 2\tau_{\text{sh}} + 12\tau_{\text{CZ}} + 2\tau_{\text{init}} + 2\tau_{\text{meas}}. \tag{B2}$$

They are both longer than the corresponding times in the surface code as expected. The overall code-cycle time is again determined by the rate-limiting pipeline out of the data pipeline and the ancilla pipeline and is given by Eq. (9) with the new $T_{\text{circ}}^{\text{data}}$ and $T_{\text{circ}}^{\text{anc}}$ for the color code.

The above process can be sped up by doubling the number of ancilla qubits in the ancilla pipeline, with the first half of them being the Z-check ancilla qubits and the second half of them being the X-check ancilla qubits. That is to say, when we have k qubits in each data pipeline corresponding to k different color-code patches, we have $2k$ qubits in each ancilla pipeline. In this way, different operations of the X and Z checks might be applied at the same time, e.g., the CZs of the Z checks and initialization of the X checks, or the measurements of the Z checks and the CZs of the X checks.

APPENDIX C: TIME OVERHEAD OF PIPELINED MAGIC STATE DISTILLATION

1. Color codes

Let us first consider the case in which the base code is a color code. For color codes, the only operation requiring

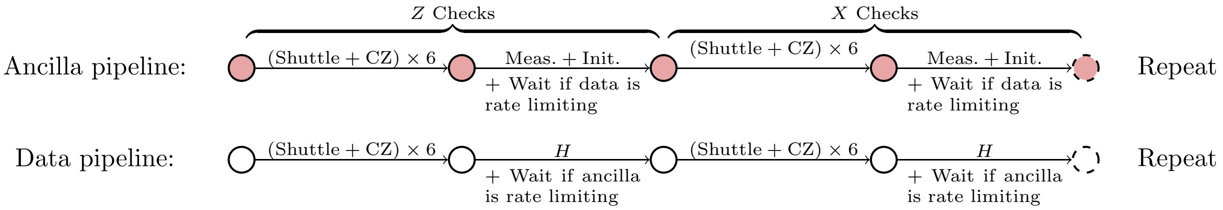


FIG. 25. Pipeline workflow for a code cycle in color code. As illustrated in the diagram, “Wait” needs to be added into the pipeline for synchronization depending on which pipeline is the rate-limiting pipeline. We do not include the initializations of qubits before all code cycles and the measurement of all qubits after all code cycles.

$\mathcal{O}(d)$ code cycles are CNOTs via lattice surgeries. Hence, the time required for a given circuit is largely determined by the number of rounds of CNOTs via lattice surgeries we need to implement, with each round costing $2d$ code cycles (Table I). In this way, the number of code cycles D we need for the distillation circuit for different pipelining schemes are given below.

(a) **One layer (no pipelining).** We need to perform five rounds of multitarget CNOTs in Fig. 19 and one additional round of CNOT for teleporting the T gates. Hence, we have

$$D = (5 + 1)2d = 12d. \tag{C1}$$

(b) **Five layers.** We need to implement three rounds of CNOTs between the T-state stack and the distillation stack via lattice surgeries for T-gate teleportation. All the other CNOTs are implemented transversally. Hence, we have

$$D = 3 \times 2d = 6d, \tag{C2}$$

i.e., half the time needed without pipelining in Eq. (C1).

(c) **Ten layers.** All CNOTs can be implemented transversally and thus no CNOTs via lattice surgeries are required. However, as mentioned in Table I, the initialization process requires the whole circuit to last at least d code cycles. Hence, we have

$$D = d, \tag{C3}$$

which is 12 times smaller than the time needed without pipelining in Eq. (C1).

2. Surface codes

For the surface code, the only difference is that the S gate for the correction in the T-gate teleportation also requires $\mathcal{O}(d)$ code cycles. Hence, for each round of T-gate teleportation, we need to add d code cycles to the overall time cost. The number of code cycles D we need for the distillation circuit for different pipelining schemes are given below.

(a) **One layer (no pipelining).** One round of T-gate teleportation, which means adding d to Eq. (C1) and thus we have

$$D = 13d. \tag{C4}$$

(b) **Five layers.** Three rounds of T-gate teleportation, which means adding $3d$ to Eq. (C2). However, note that in this case we can again view initialization as instantaneous and thus we have

$$D = 9d,$$

which is 1.4 times smaller than the time needed without pipelining in Eq. (C4).

(c) **Ten layers.** Three rounds of T gate teleportation, which means adding $3d$ to Eq. (C3) and thus we have

$$D = 3d,$$

which is 4.3 times smaller than the time needed without pipelining in Eq. (C4).

We do not discuss the pipelining time cost in Eq. (6). This is because magic state distillation is just a subroutine and not the full circuit while the pipelining time cost is for implementing the full circuit. As mentioned in Sec. III A, when we look at the full circuit, for most of the interesting applications we have relatively deep computations, such that we can neglect the pipelining time cost. Even if we focus only on the magic state distillation circuit itself, the number of qubits we need in each pipeline is at most $k = 10$. This is only a fraction of the code distance in practice ($d \sim 30$), which is in turn smaller than the number of code cycles D. Hence, following Eq. (6), the pipelining time cost is only a fraction of the time needed for one round of the magic state distillation subroutine and thus should be negligible compared to the time needed for the full circuit.

APPENDIX D: SPACE-TIME OVERHEAD OF PIPELINED SURFACE-CODE FAULT-TOLERANT COMPUTATION

The full process of fault-tolerant quantum computation can be viewed as a pipeline in which magic states are distilled and then consumed. At each time step, we input N_0 magic states into the pipeline, which are distilled into $N_1 \leq N_0$ magic states after the first round of magic state distillation (MSD) and then further distilled into $N_2 \leq N_1$ magic states after the second round. Practical implementations of surface codes rarely go beyond two rounds of MSD, thus these N_2 magic states go straight into the main computation to be consumed for implementing T gates. Magic states input at *different* time steps can be processed concurrently in the first round of MSD, the second round of MSD and the main computation, and thus these three steps can form a pipeline for the full fault-tolerant computation.

We analyze the space-time saving achievable by the first round of MSD and the main computation in Secs. IV and III E, respectively. Unlike the first round of MSD in which the input T states are prepared in place in constant time (Table I), for the second round of MSD, *all* 15 of its input T states need to coexist with the qubits for the distillation code before we can carry out the distillation circuit. Hence, when using the pipelining architecture, unless we can fit 20 logical qubits into one stack, we need to have multiple data stacks to carry out the second round of MSD. For the five-layer pipelining scheme, we can still achieve the same time saving as the first round of MSD, but now the number of code stacks needed is $A = 5$ (including one ancilla stack). For the ten-layer pipelining scheme, additional time is needed for the second round of MSD to move the input T states into the same stack as the distillation code *or* we need to carry out CNOTs between different stacks for T-gate teleportation. Correspondingly, the number of code cycles needed becomes $D = 7d$ (slightly less than the five-layer scheme since the distillation code can be initialized in the same stack with five of the T states) and the number of code stacks needed is $A = 3$ (including one ancilla stack).

The space-time saving brought by different pipelining schemes at the different steps of the full fault-tolerant computation is summarized in Table IV. There for simplicity we assume the code-cycle times stay the same as we fit five and ten qubits into each loop: $T_{\text{cycle}}^{(1)} = T_{\text{cycle}}^{(5)} = T_{\text{cycle}}^{(10)}$, which is achievable in silicon architecture by adding measurement devices as shown in Sec. V A. *The speed of the full computation is determined by the rate-limiting steps in the whole computation pipeline*. In practice, one can only fit in a limited number of MSD factories due to the significant space overhead involved, and thus the rate-limiting step is usually one of the two MSD steps [46,47,53]. Hence, the time-saving factor achievable by pipelining for the full computation is given by the time-saving factor achievable by the rate-limiting MSD step, which is a 1.4 times reduction in the time overhead for the five-layer pipelining scheme. For the ten-layer pipelining scheme, the factor of achievable time overhead reduction is 4.3 and 1.9 when the rate-limiting step is the first and second rounds of MSD, respectively. On the other hand, the space overhead saving for the full computation lies between the saving achieved by the different steps, and is dependent on the fraction of the total spaces used for different steps. One can *at least* achieve a x time reduction in the spatial overhead by using the x-layer pipelining scheme, and this space saving will increase as we increase the fraction of spaces used for MSD. Hence, the five-layer pipelining scheme can achieve a 5 to 16 times reduction in the space overhead, while the ten-layer pipelining scheme can achieve a 10 to 50 times reduction in the space overhead.

At the early stage of fault-tolerant computation, we have only enough resources for one round of magic state distillation, or a very limited number of first-round MSD factories in a two-round MSD process. In such cases, the rate-limiting step will be the first round of MSD, which means a 7 to 20 times reduction in the space-time overhead for the five-layer pipelining scheme, and a 40 to 200 times reduction in the space-time overhead for the ten-layer pipelining scheme. In the later stage of fault-tolerant computation, since it is possible to fit in more MSD factories for the first round, sometimes the second round of MSD might become the rate-limiting step, which will lead to the *same* reduction in the space-time overhead for the five-layer pipelining scheme, and a 20 to 100 times reduction in the space-time overhead for the ten-layer pipelining scheme.

It is worth noting that the analysis above is just a crude estimate of the overall saving for the full fault-tolerant computation. It is impossible to get an exact figure for the space-time saving without knowing the specific hardware constraints and the exact logical circuits that we try to implement. Another subtlety that we do not mention above is the stochastic nature of MSD, i.e., inputting N_0 magic states into the pipeline does not guarantee to output N_1 and N_2 magic states after the first and second round of MSD, respectively. The output numbers indicate only

TABLE IV. Summary of the space and time saving of the various pipelining schemes for different steps in surface-code fault-tolerant computation. Here d is the code distance. We assume the code-cycle time does not change as we fit five and ten qubits into the pipeline, which can be achieved by having one or two more measurement devices in each loop for silicon platforms as noted in Sec. V A. Without the additional measurement devices, the savings indicated above will reduce by factors of 1.2 and 2.2 for the five-layer scheme and the ten-layer scheme, respectively.

Steps in the pipeline	Space saving factor	Time saving factor
First round MSD	16.6	1.4
Second round MSD	10	1.4
Main computation	≥ 5	intrastack CNOTs: $\mathcal{O}(d)$ interstack CNOTs: ≤ 5

(a) 5 qubits per loop.

Steps in the pipeline	Space saving factor	Time saving factor
First round MSD	50	4.3
Second round MSD	16.6	1.9
Main computation	≥ 10	Intrastack CNOTs: $\mathcal{O}(d)$ interstack CNOTs: ≤ 10

(b) Ten qubits per loop.

the expected behavior in each step. Nonetheless, the above analysis gives a good intuition about the expected saving achievable via pipelining, at least in the limit of large N_0.

If we are allowed to shuttle qubits across different loops using, e.g., the Y junctions shown in Fig. 9 that connect different loops, then it is possible to move logical qubits from one stack to another using shuttling to enable intrastack transversal CNOTs rather than performing interstack lattice surgery CNOTs. The time needed for such shuttling of logical qubits from one stack to another will still scale with the code distance d. However, since shuttling is usually much faster than the other operations, such movement of logical qubits may still be much faster than the other logical operations for practical code sizes. Therefore, this may be an interesting option to explore to effectively enable transversal CNOTs among all qubits even in a multistack picture.

APPENDIX E: SURFACE-CODE PIPELINE USING SEMICONDUCTOR SPIN QUBITS

Now let us look at surface-code pipelines using semiconductor spin qubits following a similar argument to Sec. V A, there the ranking of the processing time for different steps in the pipeline is $\tau_{\text{meas}} > \tau_{\text{CZ}} > \tau_{\text{H}}$ while the individual shuttling step and the initialization step take negligible time. Note that $\tau_{\text{meas}} = \tau_{\text{meas}}^{m=1}/m$, where m is the number of measurement devices in each loop. An assumption that measurement remains the rate-limiting step, implies that

$$\tau_{\text{meas}} = \tau_{\text{meas}}^{m=1}/m \geq \tau_{\text{CZ}} \quad \Rightarrow \quad m \leq \tau_{\text{meas}}^{m=1}/\tau_{\text{CZ}}. \tag{E1}$$

In Sec. V A, we discuss the steady-flow scheme in which the time gap is $\tau_{\text{gap}} = \tau_{\text{max}} = \tau_{\text{meas}}$ and the minimum cycle period is inside the data pipeline with the value of

$$T_{\text{loop}}^{\text{min}} = \tau_{\text{sh}} + \tau_{\text{CZ}} + \tau_{\text{H}}. \tag{E2}$$

The maximum number of qubits we can fit into the pipeline is given by Eq. (4):

$$K_{\text{loop}} = \frac{T_{\text{loop}}^{\text{min}}}{\tau_{\text{gap}}} + 1. \tag{E3}$$

Hence, in order to fit in more qubits, we can either reduce the time gap such that $\tau_{\text{gap}} < \tau_{\text{max}} = \tau_{\text{meas}}$ and/or increase the minimum cycle period such that $T_{\text{loop}}^{\text{min}} > \tau_{\text{sh}} + \tau_{\text{CZ}} + \tau_{\text{H}}$.

In this section, we focus only on time gaps that are larger than the rate-limiting component time on the data pipeline $\tau_{\text{gap}} \geq \tau_{\text{CZ}}$, i.e.,

$$\frac{T_{\text{loop}}^{\text{min}}}{K_{\text{loop}} - 1} \geq \tau_{\text{CZ}}, \tag{E4}$$

so that no time-gap buffering would be needed in the data pipeline.

There is one cycle around the loop in the ancilla pipeline while there are three cycles in the data pipeline. Out of these cycles, for the given operation times, the minimum cycle period $T_{\text{loop}}^{\text{min}}$ is most likely inside the data pipeline. Hence, increasing $T_{\text{loop}}^{\text{min}}$ will mean adding buffering to the data pipeline, but have no effects on the ancilla pipeline. On the other hand, reducing the time gap τ_{gap} below $\tau_{\text{max}} = \tau_{\text{meas}}$ means we need to add time-gap buffering to the measuring step in the ancilla pipeline following Eq. (G6):

$$\begin{aligned}\Delta T_{\text{gap,anc}}^{\text{loop}} &= (K_{\text{loop}} - 1)(\tau_{\text{meas}} - \tau_{\text{gap}})\\ &= (K_{\text{loop}} - 1)\tau_{\text{meas}} - T_{\text{loop}}^{\text{min}},\end{aligned}$$

where we use Eq. (E3). Combining with the ancilla circuit time given by Eq. (12), we can obtain the effective ancilla circuit time to be

$$\begin{aligned}T_{\text{eff}}^{\text{anc}} &= T_{\text{circ}}^{\text{anc}} + \Delta T_{\text{gap,anc}}\\ &= \tau_{\text{sh}} + 4\tau_{\text{CZ}} + \tau_{\text{meas}} + (K_{\text{loop}} - 1)\tau_{\text{meas}} - T_{\text{loop}}^{\text{min}}\\ &= \tau_{\text{sh}} + 5\tau_{\text{CZ}} + K_{\text{loop}}\tau_{\text{meas}} - T_{\text{loop}}^{\text{min}}\end{aligned} \tag{E5}$$

when we increase $T_{\text{loop}}^{\text{min}}$ and/or decrease τ_{gap}. Note that τ_{gap} does not explicitly appear in the equation here since we can re-express $T_{\text{loop}}^{\text{min}}$ in terms of τ_{gap} for a given K_{loop} using Eq. (E3).

For the time needed for the data pipeline, we need to consider several parameter regimes.

1. No buffering in the data pipeline

No buffering in the data pipeline means that $T_{\text{loop}}^{\text{min}}$ will not change, which is given by Eq. (E2). The time gap used can be obtained using Eqs. (E3) and (E2) for different K_{loop}:

$$\tau_{\text{gap}} = \frac{\tau_{\text{sh}} + \tau_{\text{CZ}} + \tau_{\text{H}}}{K_{\text{loop}} - 1} \tag{E6}$$

this gives $\tau_{\text{gap}} = 0.28\ \mu\text{s}$ for $K_{\text{loop}} = 5$ and $\tau_{\text{gap}} = 0.125\ \mu\text{s}$ for $K_{\text{loop}} = 10$.

No buffering in the data pipeline also means $\tau_{\text{gap}} \geq \tau_{\text{CZ}}$, which simply means that

$$K_{\text{loop}} \leq \frac{\tau_{\text{sh}} + \tau_{\text{CZ}} + \tau_{\text{H}}}{\tau_{\text{CZ}}} + 1 = 12, \tag{E7}$$

i.e., we can at most achieve $K_{\text{loop}} = 12$ without adding any buffering to the data pipeline.

In this way, the effective circuit time for the data pipeline is the *same* as before in Eq. (11)

$$T_{\text{eff}}^{\text{data}} = T_{\text{circ}}^{\text{data}} = 3\tau_{\text{sh}} + 8\tau_{\text{CZ}} + 2\tau_{\text{H}}, \tag{E8}$$

which is independent of the time gap.

Substituting Eq. (E2) into Eq. (E5) we get the effective ancilla circuit time in this case to be

$$T_{\text{eff}}^{\text{anc}} = 4\tau_{\text{CZ}} - \tau_{\text{H}} + K_{\text{loop}}\frac{\tau_{\text{meas}}^{m=1}}{m}. \tag{E9}$$

a. Data pipeline is rate limiting

For the data pipeline to be rate limiting, the number of measurement devices we need per loop is

$$\begin{aligned} T_{\text{eff}}^{\text{anc}} &\leq T_{\text{eff}}^{\text{data}} \\ m &\geq \left\lceil \frac{\tau_{\text{meas}}^{m=1} K_{\text{loop}}}{3\tau_{\text{sh}} + 4\tau_{\text{CZ}} + 3\tau_{\text{H}}} \right\rceil = \left\lceil \frac{K_{\text{loop}}}{3.5} \right\rceil. \end{aligned} \tag{E10}$$

In this way, the code-cycle time is just the data circuit time, which in turn is unchanged since no buffering is added to the data pipeline. Hence, the code-cycle time T_{cycle} is just the *same* as the steady-flow scheme in Sec. V A with $T_{\text{cycle}} = 3.85$ μs.

Hence, following Eq. (E10), to maintain the same code-cycle time, we can fit in $K_{\text{loop}} = 5$ qubits to carry out the five-layer distillation scheme if we have at least $m = \lceil 5/3.5 \rceil = 2$ measurement devices per loop. In order to fit in $K_{\text{loop}} = 10$ qubits to carry out the ten-layer distillation scheme, we need $m = \lceil 10/3.5 \rceil = 3$ measurement devices per loop.

b. Ancilla pipeline is rate limiting

For the ancilla pipeline to be rate limiting, we need the reverse of Eq. (E10) to be true, i.e., $m \leq \lfloor K_{\text{loop}}/3.5 \rfloor$. If we want to fit in $K_{\text{loop}} = 5$ qubits, this implies $m = 1$. Since the ancilla pipeline is rate limiting, the code-cycle time is simply the effective ancilla circuit time given by Eq. (E5). For $m = 1$ and $K_{\text{loop}} = 5$ we have

$$T_{\text{cycle}} = T_{\text{eff}}^{\text{anc}} = 4\tau_{\text{CZ}} - \tau_{\text{H}} + K_{\text{loop}}\frac{\tau_{\text{meas}}^{m=1}}{m} \approx 5.4 \text{ μs}.$$

If we want to fit in $K_{\text{loop}} = 10$ qubits, this implies $m \leq 2$, which gives a code-cycle time of

$$T_{\text{cycle}} = 4\tau_{\text{CZ}} - \tau_{\text{H}} + K_{\text{loop}}\frac{\tau_{\text{meas}}^{m=1}}{m} \approx \begin{cases} 10.4 \text{ μs} & m = 1 \\ 5.4 \text{ μs} & m = 2 \end{cases}.$$

2. Adding buffering to the data pipeline

Here we consider the case in which we add buffering to the data pipeline to increase $T_{\text{loop}}^{\text{min}}$ in order to fit in more qubits. For simplicity, we consider only the case in which $T_{\text{loop}}^{\text{min}}$ is increased until it is larger than the slowest cycle in the data pipeline

$$T_{\text{loop}}^{\text{min}} \geq \tau_{\text{sh}} + 4\tau_{\text{CZ}}, \tag{E11}$$

so that all three cycles in the data pipeline can take the same cycling period. (We also require $\tau_{\text{gap}} \geq \tau_{\text{CZ}}$, which is true as long as $K_{\text{loop}} \leq 12$). In such a case, the effective circuit time for the data pipeline is simply

$$T_{\text{eff}}^{\text{data}} = 3T_{\text{loop}}^{\text{min}}. \tag{E12}$$

For a given K_{loop}, this effective circuit time for the data pipeline and that for the ancilla pipeline in Eq. (E5) is equal when

$$T_{\text{loop}}^{\text{min}*} = \frac{\tau_{\text{sh}} + 4\tau_{\text{CZ}} + \tau_{\text{meas}}^{m=1} K_{\text{loop}}/m}{4}. \tag{E13}$$

When $T_{\text{loop}}^{\text{min}}$ increase beyond $T_{\text{loop}}^{\text{min}*}$, we have $T_{\text{eff}}^{\text{data}} > T_{\text{eff}}^{\text{anc}}$ while if we have $T_{\text{loop}}^{\text{min}} < T_{\text{loop}}^{\text{min}*}$, then $T_{\text{eff}}^{\text{data}} < T_{\text{eff}}^{\text{anc}}$.

Hence, the surface-code cycle time is given as

$$\begin{aligned} T_{\text{cycle}} &= \max(T_{\text{eff}}^{\text{data}}, T_{\text{eff}}^{\text{anc}}) \\ &= \begin{cases} 3T_{\text{loop}}^{\text{min}} & T_{\text{loop}}^{\text{min}} \geq T_{\text{loop}}^{\text{min}*} \\ \tau_{\text{sh}} + 4\tau_{\text{CZ}} + K_{\text{loop}}\tau_{\text{meas}} - T_{\text{loop}}^{\text{min}} & T_{\text{loop}}^{\text{min}} < T_{\text{loop}}^{\text{min}*} \end{cases}, \end{aligned}$$

which has the minimum at $T_{\text{loop}}^{\text{min}} = T_{\text{loop}}^{\text{min}*}$, which gives

$$T_{\text{cycle}}^{*} = 3T_{\text{loop}}^{\text{min}*} = \frac{3\tau_{\text{sh}} + 12\tau_{\text{CZ}} + 3K_{\text{loop}}\tau_{\text{meas}}}{4}.$$

Substituting the time needed for the different operations into Eq. (E13), the minimum code-cycle time is given by

$$T_{\text{loop}}^{\text{min}} = T_{\text{loop}}^{\text{min}*} = \left(0.35 + \frac{K_{\text{loop}}}{4m}\right) \text{ μs}.$$

The restriction on $T_{\text{loop}}^{\text{min}}$ in Eqs. (E11) and (E4) translates into $T_{\text{loop}}^{\text{min}} \geq 1.4$ and $T_{\text{loop}}^{\text{min}} \geq 0.1K_{\text{loop}} - 0.1$, which is always be true if we have $K_{\text{loop}}/m \geq 5$ and $m \leq 2$.

For $K_{\text{loop}} = 10,\ m = 2$ and $K_{\text{loop}} = 5,\ m = 1$, we have $K_{\text{loop}}/m = 5$ and

$$\begin{aligned} T_{\text{loop}}^{\text{min}*} &= 1.6 \text{ μs}, \\ T_{\text{cycle}}^{*} &= 3T_{\text{loop}}^{\text{min}*} = 4.8 \text{ μs}. \end{aligned}$$

The corresponding time gap is $\tau_{\text{gap}} = T_{\text{loop}}^{\text{min}^*}/9 = 0.18$ μs for $K_{\text{loop}} = 10$ and $\tau_{\text{gap}} = T_{\text{loop}}^{\text{min}^*}/4 = 0.4$ μs for $K_{\text{loop}} = 5$.

For $K_{\text{loop}} = 10,\ m = 1$, we have $K_{\text{loop}}/m = 10$ and

$$\begin{aligned} T_{\text{loop}}^{\text{min}*} &= 2.85 \text{ μs}, \\ T_{\text{cycle}}^{*} &= 3T_{\text{loop}}^{\text{min}*} = 8.55 \text{ μs}. \end{aligned}$$

The corresponding time gap is $\tau_{\text{gap}} = T_{\text{loop}}^{\text{min}^*}/9 = 0.32$ μs.

3. Others

Further pipelining schemes can be explored where $\tau_{\text{gap}} < \tau_{\text{CZ}}$. However, in that case, we need to take into account the time-gap buffering needed in the data pipeline and the corresponding change in the minimum cycling period $T_{\text{loop}}^{\text{min}}$.

APPENDIX F: SEMICONDUCTOR SPIN-QUBIT SURFACE-CODE PIPELINE USING ESR

In Sec. E, we assume the single-qubit gates are performed using EDSR with the time required being $\tau_{\text{H}} =$ 25ns. However, EDSR has required the incorporation of micromagnets into the architecture, which is not always possible in practice. In this section, we perform the same analysis but with the single-qubit gates carried out using electron-spin resonance (ESR) instead. High-fidelity Rabi oscillation using ESR with a period of approximately 1 μs has been demonstrated for both local field and global field control [86,87]. Hence, the $\pi/2$ Y rotation: $Y^{\frac{1}{2}}$, which corresponds to a quarter of the Rabi cycle and can be used in place of the Hadamard gates, should be able to be carried out in $\tau_{\text{H}} \sim 250$ ns. The Z rotation used for constructing CZ can be carried out using stark shift [88] in a similar time scale. Hence, we assume the CZ gate can be carried out in $\tau_{\text{CZ}} \sim 300$ ns. To realize a many-qubit processor, targeting ESR effects to specific qubit(s) could be a fundamental challenge. In that respect it may be a favorable feature of homogeneous codes that multiple Hadamard gates should be performed simultaneously. Certainly it is interesting to explore the potential performance under the assumption that the architectural challenges can be met.

1. No buffering in the data pipeline

In this case, the time gap used can be obtained for different K_{loop} can be obtained using Eq. (E6), which gives $\tau_{\text{gap}} = 0.39$ μs for $K_{\text{loop}} = 5$ and $\tau_{\text{gap}} = 0.17$ μs for $K_{\text{loop}} = 10$.

The restriction $\tau_{\text{gap}} \geq \tau_{\text{CZ}}$ translate into [see Eq. (E7)]

$$K_{\text{loop}} \leq \frac{\tau_{\text{sh}} + \tau_{\text{CZ}} + \tau_{\text{H}}}{\tau_{\text{CZ}}} + 1 = 6, \tag{F1}$$

i.e., we can at most achieve $K_{\text{loop}} = 6$ without adding any buffering to the data pipeline.

The effective circuit time for the data pipeline is

$$T_{\text{eff}}^{\text{data}} = T_{\text{circ}}^{\text{data}} = 3\tau_{\text{sh}} + 8\tau_{\text{CZ}} + 2\tau_{\text{H}} = 5.9\ \mu\text{s}. \tag{F2}$$

a. Data pipeline is rate limiting

In this way, the code-cycle time is just the data circuit time given by Eq. (F2): $T_{\text{cycle}} = 5.9$ μs. This includes the steady-flow scheme ($T_{\text{cycle}}^{(1)} = 5.9$ μs).

The requirement on the number of measurement devices is given by Eq. (E10):

$$m \geq \left\lceil \frac{\tau_{\text{meas}}^{m=1} K_{\text{loop}}}{3\tau_{\text{sh}} + 4\tau_{\text{CZ}} + 3\tau_{\text{H}}} \right\rceil = \left\lceil \frac{K_{\text{loop}}}{5} \right\rceil.$$

Hence, with $m = \lceil 5/5 \rceil \approx 1$ measurement devices per loop, we can fit in $K_{\text{loop}} = 5$ qubits to carry out the five-layer distillation scheme and maintain the same code-cycle time ($T_{\text{cycle}}^{(5)} = 5.9$ μs).

b. Ancilla pipeline is rate limiting

We need $m < K_{\text{loop}}/5$. If we want to fit in $K_{\text{loop}} = 10$ qubits, this implies $m = 1$. The corresponding code-cycle time is just the ancilla circuit time given by Eq. (E9):

$$T_{\text{cycle}} = T_{\text{eff}}^{\text{anc}} = 4\tau_{\text{CZ}} - \tau_{\text{H}} + K_{\text{loop}} \frac{\tau_{\text{meas}}^{m=1}}{m} \approx 10.95\ \mu\text{s}.$$

2. Adding buffering to the data pipeline

Substituting the time needed for the different operations into Eq. (E13) and focusing on the case of $K_{\text{loop}} = 10$, the minimum code-cycle time is given by

$$\begin{aligned} T_{\text{loop}}^{\text{min}} = T_{\text{loop}}^{\text{min*}} &= \frac{\tau_{\text{sh}} + 4\tau_{\text{CZ}} + K_{\text{loop}}\tau_{\text{meas}}}{4} \\ &= \left(0.55 + \frac{5}{2m}\right)\ \mu\text{s}. \end{aligned}$$

This satisfies both Eqs. (E11) and (E4) as long as $m = 1$.

For $m = 1$, the minimum code cycle time is simply

$$T_{\text{cycle}}^{*} = 3T_{\text{loop}}^{\text{min*}} = 9.15\ \mu\text{s},$$

which allow us to fit ten qubits into the pipeline to carry out the ten-layer distillation scheme ($T_{\text{cycle}}^{(10)} = 9.15$ μs). The corresponding time gap is $\tau_{\text{gap}} = T_{\text{loop}}^{\text{min*}}/9 = 0.34$ μs.

3. Space-time overhead saving for magic state distillation

From the arguments above, with $m = 1$ measurement devices per loop, the time required for no one-layer, five-layer, and ten-layer schemes for the magic state distillation circuit are

$$\begin{aligned} T_{\text{cycle}}^{(1)} &= 5.9\ \mu\text{s} \quad (\text{Sec. F1a}), \\ T_{\text{cycle}}^{(5)} &= 5.9\ \mu\text{s} \quad (\text{Sec. F1a}), \\ T_{\text{cycle}}^{(10)} &= 9.15\ \mu\text{s} \quad (\text{Sec. F2}). \end{aligned}$$

Substituting into Table II, we get 24 times space-time saving by using the five-layer scheme and 140 times space-time saving by using the ten-layer scheme.

Note that with these new operation times using ESR, we are not able to simply increase the number of measurement devices m in each loop to improve the efficiency of the pipeline. This is because the measurement time is much closer to the second slowest step: CZ gates. Therefore, the measurement step will not be the rate-limiting step anymore when there are more than two measurement devices in each loop and improving its efficiency will not the pipelining efficiency. However, it is possible to add more gate devices at the same time to further parallelize the other steps as well, which can then improve the efficiency of the whole pipeline.

APPENDIX G: TIME COSTS FOR QUBIT PIPELINES

Here we consider different ways k qubits can flow through a pipeline with M steps. The time required for the rate-limiting step is given as $\tau_{\max}$. For every step in the pipeline, we can put buffering regions before and after the step, and we call them *the entry buffer* and *the exit buffer* of the step, respectively. These buffers can temporarily hold the qubits if necessary and otherwise cost zero time. When we say we have a qubit stream with a qubit time gap $\tau_{\mathrm{gap},m}$ in between the m^{th} and $(m+1)^{\mathrm{th}}$ steps, it means that after any given qubit in the qubit stream exits the m^{th} exit buffer and enters the $m+1^{\mathrm{th}}$ exit buffer, the next qubit will do the same after time $\tau_{\mathrm{gap},m}$. The time gap for the input qubit stream is denoted as $\tau_{\mathrm{gap},0}$ and the time gap of the qubit stream at the output is simply $\tau_{\mathrm{gap},M}$. We always have to make sure the time gap between the qubits entering the m^{th} step is larger than the processing time of the m^{th} step τ_m. Hence, sometimes there is a need for changing the time gap from $\tau_{\mathrm{gap},m-1}$ to τ_m before entering the m^{th} step if $\tau_{\mathrm{gap},m-1} < \tau_m$.

There are two ways to change the time gap:

(a) Increasing the time gap from τ_{gap} to τ'_{gap}, we need to buffer the nth qubit by the amount of $(n-1)(\tau'_{\mathrm{gap}} - \tau_{\mathrm{gap}})$. Note that all qubits are buffered other than the first qubit.

(b) Decreasing the time gap from τ_{gap} to τ'_{gap}, we need to buffer the nth qubit by the amount of $(k-n)(\tau_{\mathrm{gap}} - \tau'_{\mathrm{gap}})$. Note that all qubits are buffered other than the last qubit.

1. Constant time gap

Let us suppose we want to maintain a time gap of τ_{gap} for the qubit stream throughout the pipeline: $\tau_{\mathrm{gap},m} = \tau_{\mathrm{gap}} \quad \forall m$. Such a qubit stream can pass through all the steps that have $\tau_m \le \tau_{\mathrm{gap}}$ without any buffering. However, whenever we need to pass through a step with $\tau_m > \tau_{\mathrm{gap}}$, we need to temporary increase the time gap from τ_{gap} to τ_m, passing through the step, and reduce the time gap from τ_m to τ_{gap} again. This will buffer the *first qubit* by the amount of $(k-1)(\tau_m - \tau_{\mathrm{gap}})$. Hence, after passing through all steps in the pipeline, the first qubit will be buffered by an amount of

$$\Delta T_{\mathrm{gap}} = (k-1)\left(\sum_{m=1}^{M} \max(\tau_{\mathrm{m}} - \tau_{\mathrm{gap}}, 0)\right). \tag{G1}$$

Hence, the effective time needed to process the first qubit is now

$$T_{\mathrm{eff}} = T_{\mathrm{circ}} + \Delta T_{\mathrm{gap}}. \tag{G2}$$

Time needed to process any additional qubit is simply given by the time gap τ_{gap} and thus the total time needed to process k qubits using this pipeline is

$$T_{\mathrm{pipe}}(k) = T_{\mathrm{eff}} + (k-1)\tau_{\mathrm{gap}}. \tag{G3}$$

If we have $\tau_{\mathrm{gap}} = \tau_{\max}$, we then have $\max(\tau_{\mathrm{m}} - \tau_{\mathrm{gap}}, 0) = 0$ for all m and hence the pipelining time is simply given by Eq. (3) as expected, in which the first qubit will need T_{circ} to complete the pipeline while and any additional qubit will need $\tau_{\max}$, regardless of the depth of the pipeline M.

Using Eq. (G1), we can rewrite Eq. (G3) as

$$T_{\mathrm{pipe}}(k) = T_{\mathrm{circ}} + (k-1)\left(\tau_{\mathrm{gap}} + \sum_{m=1}^{M} \max(\tau_m - \tau_{\mathrm{gap}}, 0)\right), \tag{G4}$$

which can also be viewed as Eq. (G3) with $\Delta T_{\mathrm{gap}} = 0$ and $\tau'_{\mathrm{gap}} = \tau_{\mathrm{gap}} + \sum_{m=1}^{M} \max(\tau_m - \tau_{\mathrm{gap}}, 0)$. Hence, if we have $\tau_{\mathrm{gap}} < \tau_{\max}$, then the time needed for processing an additional qubit through the pipeline is now $\tau_{\mathrm{gap}} + \sum_{m=1}^{M} \max(\tau_m - \tau_{\mathrm{gap}}, 0)$, which is dependent on the depth of the pipeline M.

2. Steady flow around the loop

Due to the repeated use of the loop in the loop pipeline, an interesting pipelining scheme besides trying to achieve a steady flow throughout the whole pipeline will be achieving a steady flow only on the loop instead (excluding measurement and initialization), which we simply call the *steady-loop-flow* scheme. This is essentially the same scheme as the steady-flow scheme in Sec. II A, but only taking into account the steps on the loop. Hence, the smallest time gap we can have for the *steady-loop-flow* scheme is simply

$$\tau_{\max}^{\mathrm{loop}} = \max_{\text{on loop}} \tau_i, \tag{G5}$$

which is the time needed for the slowest step on the loop.

The *steady-loop-flow* scheme will be different from the the *steady-flow* scheme *only when the rate-limiting step is initialization or measurement*, so that we have $\tau_{\max} = \tau_{\text{init/meas}} > \tau_{\max}^{\text{loop}}$. By definition, the steady-loop-flow scheme does not require any buffers on the loop. Hence, we have the same cycling period T_{loop} as the steady-flow scheme. Due to the smaller time gap $\tau_{\max}^{\text{loop}} < \tau_{\max}$, we are able to use the steady-loop-flow scheme to fit more qubits in the pipeline compared to the steady-flow scheme according to Eq. (4).

However, using the steady-loop-flow scheme comes with additional time costs. While the steady-loop-flow scheme does not need any buffers on the loop, it requires buffers before and after it passes through the rate-limiting step (initialization or measurement) to maintain the qubit time gap $\tau_{\max}^{\text{loop}}$. The amount of such buffering put on the first qubit is denoted as $\Delta T_{\text{gap}}^{\text{loop}}$. As shown in Sec. G, to process k qubits using the steady-loop-flow scheme, the amount of buffering we need to apply is

$$\Delta T_{\text{gap}}^{\text{loop}} = (k-1)\left(\sum_m \max(\tau_m - \tau_{\max}^{\text{loop}}, 0)\right). \tag{G6}$$

Here we sum over all steps in the pipeline and only the steps are slower than the rate-limiting elements on the loop (i.e., measurements and/or initializations) are effectively included.

Hence, the effective time needed to process the first qubit is given by Eq. (G2) with $\Delta T_{\text{gap}} = \Delta T_{\text{gap}}^{\text{loop}}$:

$$T_{\text{eff}}^{\text{loop}} = T_{\text{circ}} + \Delta T_{\text{gap}}^{\text{loop}}. \tag{G7}$$

The total time needed to process k qubits using this pipeline is given by Eq. (G3) with $T_{\text{eff}} = T_{\text{eff}}^{\text{loop}}$ and $\tau_{\text{gap}} = \tau_{\max}^{\text{loop}}$:

$$T_{\text{pipe}}(k) = T_{\text{eff}}^{\text{loop}} + (k-1)\tau_{\max}^{\text{loop}}. \tag{G8}$$

3. Varying time gap

If we allow the time gap to vary, we have a series of peaks (local maxima) and troughs (local minima). Each transition from a peak to a trough will be called a *fall*. Suppose there are L falls throughout with the ℓ^{th} fall being the transition from $t_{\text{pk},\ell}$ to $t_{\text{tr},\ell}$, then the buffering we need to apply to the first qubit to carry out these L falls is simply

$$\Delta T_{\text{gap}} = (k-1)\left(\sum_{\ell=1}^{L} \tau_{\text{pk},\ell} - \tau_{\text{tr},\ell}\right).$$

Hence, using Eq. (G3), the total time needed for pipelining all k qubits (i.e., the time taken for the last qubit to exit the pipeline) is

$$T_{\text{pipe}}(k) = T_{\text{circ}} + (k-1)\left(\tau_{\text{gap},M} + \sum_{\ell=1}^{L} \tau_{\text{pk},\ell} - \tau_{\text{tr},\ell}\right).$$

If $\tau_{\text{gap},M}$ is a trough, then $\tau_{\text{tr},L} = \tau_{\text{gap},M}$ and we have

$$T_{\text{pipe}}(k) = T_{\text{circ}} + (k-1)\left(\tau_{\text{pk},L} + \sum_{\ell=1}^{L-1} \tau_{\text{pk},\ell} - \tau_{\text{tr},\ell}\right). \tag{G9}$$

Otherwise, if $\tau_{\text{gap},M}$ is a peak, then $\tau_{\text{tr},L+1} = \tau_{\text{gap},M}$ and we have

$$T_{\text{pipe}}(k) = T_{\text{circ}} + (k-1)\left(\tau_{\text{pk},L+1} + \sum_{\ell=1}^{L} \tau_{\text{pk},\ell} - \tau_{\text{tr},\ell}\right). \tag{G10}$$

In either case, if there are Q peaks throughout the whole process with their time gaps denoting as $\{\tau_{\text{pk},q} \mid 1 \le q \le Q\}$. In between these Q peaks, there are $Q-1$ troughs (i.e., excluding any troughs at the beginning and end of the pipeline) with their corresponding time gaps denoting as $\{\tau_{\text{tr},q} \mid 1 \le q \le Q-1\}$. An example is shown in

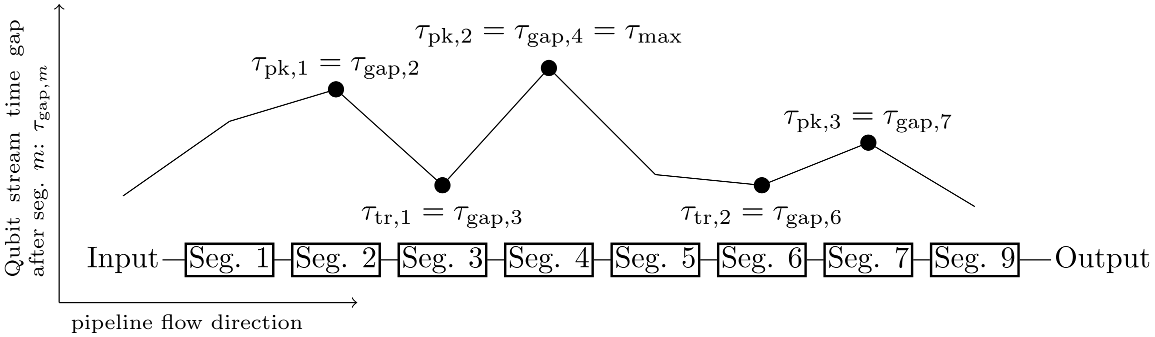

$\tau_{\text{pk},2} = \tau_{\text{gap},4} = \tau_{\max}$
$\tau_{\text{pk},1} = \tau_{\text{gap},2}$
$\tau_{\text{pk},3} = \tau_{\text{gap},7}$
$\tau_{\text{tr},1} = \tau_{\text{gap},3}$
$\tau_{\text{tr},2} = \tau_{\text{gap},6}$
Qubit stream time gap after seg. m: $\tau_{\text{gap},m}$
Input
Seg. 1
Seg. 2
Seg. 3
Seg. 4
Seg. 5
Seg. 6
Seg. 7
Seg. 9
Output
pipeline flow direction

FIG. 26. Diagram showing the change of the time gap of the qubit stream as it passes through different steps in the pipeline.

Fig. 26. Then by defining $\tau_{\mathrm{tr},0} = 0$, the additional time cost in Eqs. (G9) and (G10) can be combined into

$$T_{\mathrm{pipe}}(k) = T_{\mathrm{circ}} + (k-1)\left(\sum_{q=1}^{Q} \tau_{\mathrm{pk},q} - \tau_{\mathrm{tr},q-1}\right). \quad \text{(G11)}$$

We restrict our scheme such that at all peaks, we have $\tau_{\mathrm{pk},q} = \tau_{\mathrm{gap},m} = \tau_m$ in which we have the qth peak occurs at the mth step. Hence, $\sum_{q=1}^{Q} \tau_{\mathrm{pk},q}$ is simply the sum of time required for some subset of steps where the time peak happens, and thus it will be smaller than the overall time require for the all steps $\sum_{q=1}^{Q} \tau_{\mathrm{pk},q} < T_{\mathrm{circ}}$, which implies that

$$T_{\mathrm{pipe}}(k) = T_{\mathrm{circ}} + (k-1)\left(\sum_{q=1}^{Q} \tau_{\mathrm{pk},q} - \tau_{\mathrm{tr},q-1}\right)$$
$$< kT_{\mathrm{circ}},$$

i.e., the pipelining scheme will always have time saving over the sequential scheme.

For the sum $\sum_{q=1}^{Q} \tau_{\mathrm{pk},q} - \tau_{\mathrm{tr},q-1}$, we know that one of the peaks would be $\tau_{\max}$. Hence, we can extract out $\tau_{\max}$ and pair up the peaks and troughs before the $\tau_{\max}$ peak using "falls" and pair up the peaks and troughs after the $\tau_{\max}$ peak using "rises." In this way, we can show that $\sum_{q=1}^{Q} \tau_{\mathrm{pk},q} - \tau_{\mathrm{tr},q-1} \geq \tau_{\max}$. In other words, any time-gap variation will never outperform the scheme with a *constant* time gap of $\tau_{\max}$.

When there is only one peak, i.e., $Q = 1$, which would occur at the rate-limiting element with $\tau_{\mathrm{pk},1} = \tau_{\max}$, we would have

$$T_{\mathrm{pipe}}(k) = T_{\mathrm{circ}} + (k-1)\tau_{\max},$$

which is simply Eqs. (3) and (2). In other words, there is no additional time cost if there is just one peak in the time-gap variation.

[1] S. Bravyi, D. Gosset, R. König, and M. Tomamichel, Quantum advantage with noisy shallow circuits, Nat. Phys. **16**, 1040 (2020).

[2] S. Bravyi, D. Gosset, and R. Movassagh, Classical algorithms for quantum mean values, Nat. Phys. **17**, 337 (2021).

[3] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, Topological quantum memory, J. Math. Phys. **43**, 4452 (2002).

[4] H. Bombin and M. A. Martin-Delgado, Topological Computation without Braiding, Phys. Rev. Lett. **98**, 160502 (2007).

[5] M. Vasmer and D. E. Browne, Three-dimensional surface codes: Transversal gates and fault-tolerant architectures, Phys. Rev. A **100**, 012312 (2019).

[6] B. Koczor, Exponential Error Suppression for Near-Term Quantum Devices, Phys. Rev. X **11**, 031057 (2021).

[7] W. J. Huggins, S. McArdle, T. E. O'Brien, J. Lee, N. C. Rubin, S. Boixo, K. B. Whaley, R. Babbush, and J. R. McClean, Virtual Distillation for Quantum Error Mitigation, Phys. Rev. X **11**, 041036 (2021).

[8] X. Qiu, P. Zoller, and X. Li, Programmable Quantum Annealing Architectures with Ising Quantum Wires, PRX Quantum **1**, 020311 (2020).

[9] L. M. K. Vandersypen, H. Bluhm, J. S. Clarke, A. S. Dzurak, R. Ishihara, A. Morello, D. J. Reilly, L. R. Schreiber, and M. Veldhorst, Interfacing spin qubits in quantum dots and donors—hot, dense, and coherent, npj Quantum Inf. **3**, 34 (2017).

[10] B. Buonacorsi, Z. Cai, E. B. Ramirez, K. S. Willick, S. M. Walker, J. Li, B. D. Shaw, X. Xu, S. C. Benjamin, and J. Baugh, Network architecture for a topological quantum computer in silicon, Quantum Sci. Technol. **4**, 025003 (2019).

[11] J. M. Boter, J. P. Dehollain, J. P. van Dijk, Y. Xu, T. Hensgens, R. Versluis, H. W. Naus, J. S. Clarke, M. Veldhorst, F. Sebastiano, and L. M. Vandersypen, Spiderweb Array: A Sparse Spin-Qubit Array, Phys. Rev. Appl. **18**, 024053 (2022).

[12] H. Jnane, B. Undseth, Z. Cai, S. C. Benjamin, and B. Koczor, Multicore Quantum Computing, Phys. Rev. Appl. **18**, 044064 (2022).

[13] D. Kielpinski, C. Monroe, and D. J. Wineland, Architecture for a large-scale ion-trap quantum computer, Nature **417**, 709 (2002).

[14] B. Lekitsch, S. Weidt, A. G. Fowler, K. Mølmer, S. J. Devitt, C. Wunderlich, and W. K. Hensinger, Blueprint for a microwave trapped ion quantum computer, Sci. Adv. **3**, e1601540 (2017).

[15] V. Kaushal, B. Lekitsch, A. Stahl, J. Hilder, D. Pijn, C. Schmiegelow, A. Bermudez, M. Müller, F. Schmidt-Kaler, and U. Poschinger, Shuttling-based trapped-ion quantum information processing, AVS Quantum Sci. **2**, 014101 (2020).

[16] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes, and R. P. Stutz, Realization of Real-Time Fault-Tolerant Quantum Error Correction, Phys. Rev. X **11**, 041058 (2021).

[17] J. Hilder, D. Pijn, O. Onishchenko, A. Stahl, M. Orth, B. Lekitsch, A. Rodriguez-Blanco, M. Müller, F. Schmidt-Kaler, and U. G. Poschinger, Fault-Tolerant Parity Readout on a Shuttling-Based Trapped-Ion Quantum Computer, Phys. Rev. X **12**, 011032 (2022).

[18] Y. Tomita, M. Gutiérrez, C. Kabytayev, K. R. Brown, M. R. Hutsel, A. P. Morris, K. E. Stevens, and G. Mohler, Comparison of ancilla preparation and measurement procedures for the Steane [[7,1,3]] code on a model ion-trap quantum computer, Phys. Rev. A **88**, 042336 (2013).

[19] A. Bermudez, X. Xu, R. Nigmatullin, J. O'Gorman, V. Negnevitsky, P. Schindler, T. Monz, U. G. Poschinger, C. Hempel, J. Home, F. Schmidt-Kaler, M. Biercuk, R. Blatt, S. Benjamin, and M. Müller, Assessing the Progress of Trapped-Ion Processors Towards Fault-Tolerant Quantum Computation, Phys. Rev. X **7**, 041061 (2017).

[20] M. Gutiérrez, M. Müller, and A. Bermúdez, Transversality and lattice surgery: Exploring realistic routes toward coupled logical qubits with trapped-ion quantum processors, Phys. Rev. A **99**, 022330 (2019).

[21] A. Bermudez, X. Xu, M. Gutiérrez, S. C. Benjamin, and M. Müller, Fault-tolerant protection of near-term trapped-ion topological qubits under realistic noise sources, Phys. Rev. A **100**, 062307 (2019).

[22] R. Raussendorf, J. Harrington, and K. Goyal, A fault-tolerant one-way quantum computer, Ann. Phys. (N. Y) **321**, 2242 (2006).

[23] R. Raussendorf, J. Harrington, and K. Goyal, Topological fault-tolerance in cluster state quantum computation, New J. Phys. **9**, 199 (2007).

[24] R. Raussendorf and J. Harrington, Fault-Tolerant Quantum Computation with High Threshold in Two Dimensions, Phys. Rev. Lett. **98**, 190504 (2007).

[25] Y. Li, P. C. Humphreys, G. J. Mendoza, and S. C. Benjamin, Resource Costs for Fault-Tolerant Linear Optical Quantum Computing, Phys. Rev. X **5**, 041007 (2015).

[26] J. M. Auger, H. Anwar, M. Gimeno-Segovia, T. M. Stace, and D. E. Browne, Fault-tolerant quantum computation with nondeterministic entangling gates, Phys. Rev. A **97**, 030301 (2018).

[27] D. Herr, A. Paler, S. J. Devitt, and F. Nori, A local and scalable lattice renormalization method for ballistic quantum computation, npj Quantum Inf. **4**, 1 (2018).

[28] K. Fukui, W. Asavanant, and A. Furusawa, Temporal-mode continuous-variable three-dimensional cluster state for topologically protected measurement-based quantum computation, Phys. Rev. A **102**, 032614 (2020).

[29] J. E. Bourassa, R. N. Alexander, M. Vasmer, A. Patil, I. Tzitrin, T. Matsuura, D. Su, B. Q. Baragiola, S. Guha, G. Dauphinais, K. K. Sabapathy, N. C. Menicucci, and I. Dhand, Blueprint for a scalable photonic fault-tolerant quantum computer, Quantum **5**, 392 (2021).

[30] M. V. Larsen, C. Chamberland, K. Noh, J. S. Neergaard-Nielsen, and U. L. Andersen, Fault-Tolerant Continuous-Variable Measurement-based Quantum Computation Architecture, PRX Quantum **2**, 030325 (2021).

[31] S. Patomäki, M. Fogarty, Z. Cai, S. C. Benjamin, and J. J. L. Morton, in *Bulletin of the American Physical Society*, Vol. Volume 66, Number 1 (American Physical Society).

[32] S. B. Bravyi and A. Y. Kitaev, Quantum codes on a lattice with boundary, (1998), ArXiv:quant-ph/9811052.

[33] H. Bombin and M. A. Martin-Delgado, Topological quantum error correction with optimal encoding rate, Phys. Rev. A **73**, 062303 (2006).

[34] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Surface codes: Towards practical large-scale quantum computation, Phys. Rev. A **86**, 032324 (2012).

[35] A. J. Landahl, J. T. Anderson, and P. R. Rice, Fault-tolerant quantum computing with color codes, (2011), ArXiv:1108.5738.

[36] Z. Cai, M. A. Fogarty, S. Schaal, S. Patomäki, S. C. Benjamin, and J. J. L. Morton, A silicon surface code architecture resilient against leakage errors, Quantum **3**, 212 (2019).

[37] X. Xu, Q. Zhao, X. Yuan, and S. C. Benjamin, High-Threshold Code for Modular Hardware With Asymmetric Noise, Phys. Rev. Appl. **12**, 064006 (2019).

[38] O. Higgott and N. P. Breuckmann, Subsystem Codes with High Thresholds by Gauge Fixing and Reduced Qubit Overhead, Phys. Rev. X **11**, 031039 (2021).

[39] A. Strikis, S. C. Benjamin, and B. J. Brown, Quantum computing is scalable on a planar array of qubits with fabrication defects, (2021), ArXiv:2111.06432.

[40] H. Bombin and M. A. Martin-Delgado, Topological Quantum Distillation, Phys. Rev. Lett. **97**, 180501 (2006).

[41] C. Chamberland, A. Kubica, T. J. Yoder, and G. Zhu, Triangular color codes on trivalent graphs with flag qubits, New J. Phys. **22**, 023019 (2020).

[42] D. E. Gottesman, *Stabilizer Codes and Quantum Error Correction*, Ph.D. thesis, California Institute of Technology (1997).

[43] C. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, Surface code quantum computing by lattice surgery, New J. Phys. **14**, 123011 (2012).

[44] A. J. Landahl and C. Ryan-Anderson, Quantum computing by color-code lattice surgery, (2014), ArXiv:1407.5103.

[45] M. E. Beverland, A. Kubica, and K. M. Svore, Cost of Universality: A Comparative Study of the Overhead of State Distillation and Code Switching with Color Codes, PRX Quantum **2**, 020341 (2021).

[46] A. G. Fowler and C. Gidney, Low overhead quantum computation using lattice surgery, (2019), ArXiv:1808.06709.

[47] D. Litinski, A game of surface codes: Large-scale quantum computing with lattice surgery, Quantum **3**, 128 (2019).

[48] B. J. Brown, K. Laubscher, M. S. Kesselring, and J. R. Wootton, Poking Holes and Cutting Corners to Achieve Clifford Gates with the Surface Code, Phys. Rev. X **7**, 021029 (2017).

[49] D. Litinski and F. von Oppen, Braiding by Majorana tracking and long-range CNOT gates with color codes, Phys. Rev. B **96**, 205413 (2017).

[50] M. Beverland, V. Kliuchnikov, and E. Schoute, Surface Code Compilation via Edge-Disjoint Paths, PRX Quantum **3**, 020342 (2022).

[51] J. Haah and M. B. Hastings, Codes and protocols for distilling T, controlled-S, and Toffoli gates, Quantum **2**, 71 (2018).

[52] S. Bravyi and A. Kitaev, Universal quantum computation with ideal Clifford gates and noisy ancillas, Phys. Rev. A **71**, 022316 (2005).

[53] R. Babbush, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, A. Paler, A. Fowler, and H. Neven, Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity, Phys. Rev. X **8**, 041015 (2018).

[54] G. Burkard, T. D. Ladd, J. M. Nichol, A. Pan, and J. R. Petta, Semiconductor spin qubits, (2021), ArXiv:2112.08863.

[55] C. Jones, M. A. Fogarty, A. Morello, M. F. Gyure, A. S. Dzurak, and T. D. Ladd, Logical Qubit in a Linear Array of Semiconductor Quantum Dots, Phys. Rev. X **8**, 021058 (2018).

[56] S. Schaal, I. Ahmed, J. A. Haigh, L. Hutin, B. Bertrand, S. Barraud, M. Vinet, C.-M. Lee, N. Stelmashenko, J. W. A. Robinson, J. Y. Qiu, S. Hacohen-Gourgy, I. Siddiqi, M. F. Gonzalez-Zalba, and J. J. L. Morton, Fast Gate-Based Readout of Silicon Quantum Dots Using Josephson Parametric Amplification, Phys. Rev. Lett. **124**, 067701 (2020).

[57] A. Jones, E. Pritchett, E. Chen, T. Keating, R. Andrews, J. Blumoff, L. De Lorenzo, K. Eng, S. Ha, A. Kiselev, S. Meenehan, S. Merkel, J. Wright, L. Edge, R. Ross, M. Rakher, M. Borselli, and A. Hunter, Spin-Blockade Spectroscopy of Si/Si-Ge Quantum Dots, Phys. Rev. Appl. **12**, 014026 (2019).

[58] E. J. Connors, J. Nelson, and J. M. Nichol, Rapid High-Fidelity Spin-State Readout in Si/Si-Ge Quantum Dots via rf Reflectometry, Phys. Rev. Appl. **13**, 024019 (2020).

[59] J. Yoneda, K. Takeda, T. Otsuka, T. Nakajima, M. R. Delbecq, G. Allison, T. Honda, T. Kodera, S. Oda, Y. Hoshi, N. Usami, K. M. Itoh, and S. Tarucha, A quantum-dot spin qubit with coherence limited by charge noise and fidelity higher than 99.9%, Nat. Nanotechnol. **13**, 102 (2018).

[60] A. R. Mills, C. R. Guinn, M. J. Gullans, A. J. Sigillito, M. M. Feldman, E. Nielsen, and J. R. Petta, Two-qubit silicon quantum processor with operation fidelity exceeding 99%, Sci. Adv. **8**, eabn5130 (2022).

[61] X. Xue, M. Russ, N. Samkharadze, B. Undseth, A. Sammak, G. Scappucci, and L. M. K. Vandersypen, Quantum logic with spin qubits crossing the surface code threshold, Nature **601**, 343 (2022).

[62] A. Noiri, K. Takeda, T. Nakajima, T. Kobayashi, A. Sammak, G. Scappucci, and S. Tarucha, Fast universal quantum gate above the fault-tolerance threshold in silicon, Nature **601**, 338 (2022).

[63] D. Loss and D. P. DiVincenzo, Quantum computation with quantum dots, Phys. Rev. A **57**, 120 (1998).

[64] A. R. Mills, D. M. Zajac, M. J. Gullans, F. J. Schupp, T. M. Hazard, and J. R. Petta, Shuttling a single charge across a one-dimensional array of silicon quantum dots, Nat. Commun. **10**, (2019).

[65] I. Seidler, T. Struck, R. Xue, N. Focke, S. Trellenkamp, H. Bluhm, and L. R. Schreiber, Conveyor-mode single-electron shuttling in Si/SiGe for a scalable quantum computing architecture, (2021), ArXiv:2108.00879.

[66] B. Buonacorsi, B. Shaw, and J. Baugh, Simulated coherent electron shuttling in silicon quantum dots, Phys. Rev. B **102**, (2020).

[67] J. A. Krzywda and Ł. Cywiński, Adiabatic electron charge transfer between two quantum dots in presence of $1/f$ noise, Phys. Rev. B **101**, 035303 (2020).

[68] J. A. Krzywda and Ł. Cywiński, Interplay of charge noise and coupling to phonons in adiabatic electron transfer between quantum dots, Phys. Rev. B **104**, 075439 (2021).

[69] V. Langrock, J. A. Krzywda, N. Focke, I. Seidler, L. R. Schreiber, and Ł. Cywiński, Blueprint of a scalable spin qubit shuttle device for coherent mid-range qubit transfer in disordered Si/SiGe/SiO$_2$, (2022), ArXiv:2202.11793.

[70] A. M. Stephens, Fault-tolerant thresholds for quantum error correction with the surface code, Phys. Rev. A **89**, (2014).

[71] J. J. Wallman and J. Emerson, Noise tailoring for scalable quantum computation via randomized compiling, Phys. Rev. A **94**, 052325 (2016).

[72] D. Buterakos and S. Das Sarma, Spin-Valley Qubit Dynamics in Exchange-Coupled Silicon Quantum Dots, PRX Quantum **2**, 040358 (2021).

[73] C. H. Yang, A. Rossi, R. Ruskov, N. S. Lai, F. A. Mohiyaddin, S. Lee, C. Tahan, G. Klimeck, A. Morello, and A. S. Dzurak, Spin-valley lifetimes in a silicon quantum dot with tunable valley splitting, Nat. Commun. **4**, 2069 (2013).

[74] C. Tahan and R. Joynt, Relaxation of excited spin, orbital, and valley qubit states in ideal silicon quantum dots, Phys. Rev. B **89**, 075302 (2014).

[75] T. Tanttu, B. Hensen, K. W. Chan, C. H. Yang, W. W. Huang, M. Fogarty, F. Hudson, K. Itoh, D. Culcer, A. Laucht, A. Morello, and A. Dzurak, Controlling Spin-Orbit Interactions in Silicon Quantum Dots Using Magnetic Field Direction, Phys. Rev. X **9**, 021028 (2019).

[76] J. Levy, Universal Quantum Computation with Spin-1/2 Pairs and Heisenberg Exchange, Phys. Rev. Lett. **89**, 147902 (2002).

[77] D. P. DiVincenzo, D. Bacon, J. Kempe, G. Burkard, and K. B. Whaley, Universal quantum computation with the exchange interaction, Nature **408**, 339 (2000).

[78] D. A. Lidar, in *Quantum Information and Computation for Chemistry* (John Wiley & Sons, Ltd, 2014), p. 295.

[79] B. Koczor, The dominant eigenvector of a noisy quantum state, New J. Phys. **23**, 123047 (2021).

[80] T. E. O'Brien, S. Polla, N. C. Rubin, W. J. Huggins, S. McArdle, S. Boixo, J. R. McClean, and R. Babbush, Error Mitigation via Verified Phase Estimation, PRX Quantum **2**, 020317 (2021).

[81] Z. Cai, Resource-efficient purification-based quantum error mitigation, (2021), ArXiv:2107.07279.

[82] D. K. Tuckett, A. S. Darmawan, C. T. Chubb, S. Bravyi, S. D. Bartlett, and S. T. Flammia, Tailoring Surface Codes for Highly Biased Noise, Phys. Rev. X **9**, 041031 (2019).

[83] H. Bombin, 2D quantum computation with 3D topological codes, (2018), ArXiv:1810.09571.

[84] B. J. Brown, A fault-tolerant non-Clifford gate for the surface code in two dimensions, Sci. Adv. **6**, eaay4929 (2020).

[85] A. Strikis and L. Berent, Quantum LDPC codes for modular architectures, (2022), ArXiv:2209.14329.

[86] M. Veldhorst, J. C. C. Hwang, C. H. Yang, A. W. Leenstra, B. de Ronde, J. P. Dehollain, J. T. Muhonen, F. E. Hudson, K. M. Itoh, A. Morello, and A. S. Dzurak, An addressable quantum dot qubit with fault-tolerant control-fidelity, Nat. Nanotechnol. **9**, 981 (2014).

[87] E. Vahapoglu, J. P. Slack-Smith, R. C. C. Leon, W. H. Lim, F. E. Hudson, T. Day, J. D. Cifuentes, T. Tanttu, C. H. Yang, A. Saraiva, N. V. Abrosimov, H.-J. Pohl, M. L. W. Thewalt, A. Laucht, A. S. Dzurak, and J. J. Pla, Coherent control of electron spin qubits in silicon using a global field, (2021), ArXiv:2107.14622.

[88] J. C. C. Hwang, C. H. Yang, M. Veldhorst, N. Hendrickx, M. A. Fogarty, W. Huang, F. E. Hudson, A. Morello, and A. S. Dzurak, Impact of g-factors and valleys on spin qubits in a silicon double quantum dot, Phys. Rev. B **96**, 045302 (2017).