SWIPER: Minimizing Fault-Tolerant Quantum Program Latency via Speculative Window Decoding

Joshua Viszlai*†
University of Chicago
Chicago, Illinois, USA
viszlai@uchicago.edu

Jason D. Chadwick*
University of Chicago
Chicago, Illinois, USA
jchadwick@uchicago.edu

Sarang Joshi
University of Chicago
Chicago, Illinois, USA
sarangj@uchicago.edu

Gokul Subramanian Ravi
University of Michigan
Ann Arbor, Michigan, USA
gsravi@umich.edu

Yanjing Li
University of Chicago
Chicago, Illinois, USA
yanjingl@uchicago.edu

Frederic T. Chong
University of Chicago
Chicago, Illinois, USA
chong@cs.uchicago.edu

Abstract

Real-time decoding is a key ingredient in future fault-tolerant quantum systems, yet many decoders are too slow to run in real time. Prior work has shown that parallel window decoding can scalably meet throughput requirements in the presence of increasing decoding times. However, windowed decoding require that some decoding tasks be delayed until others have completed, which can be problematic during time-sensitive operations such as T gate teleportation, leading to suboptimal program runtimes. To alleviate this, we introduce SWIPER, a *speculative* window decoder. Taking inspiration from branch prediction in classical computer architecture, SWIPER utilizes a light-weight speculation step to predict data dependencies between adjacent decoding windows, allowing multiple layers of decoding tasks to be resolved simultaneously. Through a state-of-the-art compilation pipeline and a detailed open-source simulator, we find that SWIPER reduces application runtimes by 40% on average compared to prior parallel window decoders.

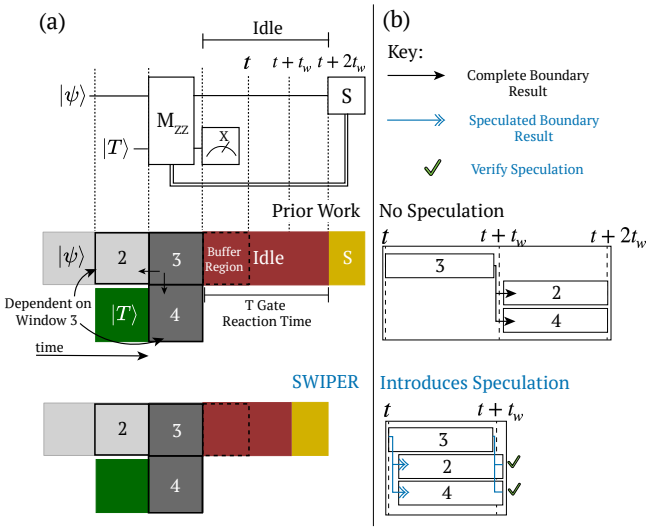

Figure 1: (a) Decoding a lattice surgery T gate teleportation (top left) using prior parallel window decoders (middle) and using SWIPER (bottom). (b) Resulting decoding pipelines. SWIPER improves decoding throughput with a lightweight *speculation* step, allowing dependent windows to begin decoding earlier.

CCS Concepts

• **Computer systems organization → Quantum computing**.

Keywords

Quantum Computing, Quantum Error Correction, Surface Code, Decoding, Window Decoding, Lattice Surgery

ACM Reference Format:
Joshua Viszlai, Jason D. Chadwick, Sarang Joshi, Gokul Subramanian Ravi, Yanjing Li, and Frederic T. Chong. 2025. SWIPER: Minimizing Fault-Tolerant Quantum Program Latency via Speculative Window Decoding. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3731022

1 Introduction

Quantum computers are poised to deliver computational speedups for problems intractable classically. It is known that many of these problems, such as quantum chemistry and factoring, will require fault-tolerant systems to translate noisy physical qubits into resilient logical qubits via quantum error correcting (QEC) codes. A critical component in the realization of QEC is the *decoder*, which

*Both authors contributed equally to this research.
†Correspondance: viszlai@uchicago.edu

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731022

must operate in real time with the quantum device. Through classical algorithms acting on streams of parity check data, the decoder effectively tracks the state of error on logical qubits. A leading proposal for decoding a long-running quantum computation is to decode *windows* of parity check data as they are generated. Adjacent windows must pass completed decoding data across window boundaries to create a global solution. In the first proposal from Dennis et al. [20], known as sliding window decoding, data is passed forward in time, creating sequentially dependent decoding problems.

Although an accurate decoding system is necessary to enable QEC, its speed also has strong implications for the success of the quantum computation. Terhal points out that if the rate of data production is higher than the rate of data decoding, then the computation will experience an *exponential* slowdown [58] in the sliding window setting due to an accumulating backlog of decoding tasks that all depend on their predecessors. In response, research has broadly improved decoder performance through a combination of algorithmic innovations [18, 34, 64] and tailored optimizations [3, 48, 62]. However, when addressing the backlog problem, it is important to clarify decoder *throughput* versus decoder *latency*. Decoder throughput is the rate at which information is decoded, whereas decoder latency is the time taken to actively decode a single problem. Innovations and optimizations in latency are important and, in turn, increase throughput. However, as discussed by Skoric et al. [55] and Tan et al. [56] the decoding backlog is primarily a throughput problem. The *parallel window decoding* strategies that they propose show how throughput requirements can be scalably met while remaining agnostic to the underlying decoding latency. In parallel window decoding, the direction of data movement between windows is modified, partitioning windows into two alternating layers. Windows in the same layer are independent and can therefore be decoded in parallel. This removes long dependency chains inherent to sliding window decoding, and as a result, given enough classical decoders to operate in parallel, the throughput of the decoding system can be arbitrarily high, effectively resolving the backlog problem.

Although prior parallel window decoding schemes successfully meet throughput requirements, their impact on the time from when a decoding window is generated to when it is decoded, known as the *reaction time*, is suboptimal. Ideally, the decoding of each window should begin as soon as the window is generated from the device. However, as shown in the decoding pipelines of Figure 1b, in prior implementations a window which depends on data from an adjacent window cannot begin until the adjacent window is complete, increasing the reaction time. If the time to decode a window is t_w, the reaction time becomes at least $2t_w$. This is particularly detrimental for "blocking" operations, such as non-Clifford gates (e.g. T, CCZ), which require the system to be fully decoded before the program can continue. An increase in reaction time in this context causes additional idle operations to be inserted until decoding is complete, slowing down the computation.

In this work, we introduce SWIPER to reduce decoding reaction time in window decoding schemes. SWIPER translates speculation strategies commonly used in classical computer architectures to the problem of window decoding, eliminating unnecessary idle time waiting for data dependencies as shown in Figure 1.

To perform impactful speculation, SWIPER leverages key insights: ① Since data dependencies between adjacent decoding windows exist only along the window boundaries, they only constitute a fraction of the total decoding problem. For example, at a code distance $d = 21$, the boundary is only $\sim 1/21 = 4.8\%$ of the total syndrome data of a window. ② Error chains that cross the boundaries between windows will be short and sparsely distributed in the overwhelming majority of cases. In such cases, solving for data dependencies does not need a full-fledged decoding process. With these insights in mind, SWIPER's predictor solves the simpler problem of finding short-weight matchings across the boundary, which can be done much more efficiently than a full decoding of the window. This tentative result can then be forwarded to the adjacent window, allowing it to begin decoding. Importantly, this does not replace the full decoder, which is still run lazily and verifies the speculation's correctness.

We find that SWIPER provides a 40% reduction in the runtime of the fault-tolerant program compared to previous work, a critical improvement given the high demand and time unit cost of quantum processors, which will need to be run on the order of hours to days for fault-tolerant programs [9, 29]. Our results utilize our SWIPER-SIM decoding simulator, which allows simulation of window decoders for fault-tolerant lattice surgery programs.

The main contributions of SWIPER are as follows:

- We introduce SWIPER, a new windowed decoder that leverages a lightweight, FPGA-compatible predictor to **improve decoder reaction time by up to 50%** compared to previous parallel window decoders.
- We develop **SWIPER-SIM, a round-level lattice surgery decoding simulator**, which enables novel program-level simulations of windowed decoding and allows us to analyze how decoding reaction time impacts overall benchmark runtime.
- Using SWIPER-SIM, we discover variance in reaction time for prior parallel window decoders based on the alignment of T gates. We therefore introduce an *aligned window schedule* to enforce proper alignment, **improving reaction time by up to 50% in alignment-limited settings**.
- We study SWIPER under varying decoder latency and show that **for fixed runtime constraints, SWIPER consistently relaxes decoder latency requirements by over $2-5\times$,** enabling the development of more powerful and accurate decoders.
- We evaluate SWIPER with realistic decoder parameters on a variety of representative benchmarks and **demonstrate consistent program runtime reductions of 40% regardless of program size**.
- We analyze the added classical overhead of SWIPER and provide a heuristic to determine how many classical decoders to allocate for SWIPER. We find that the benefits of SWIPER come at the cost of a consistent 31% increase in the number of concurrent decoders compared to prior parallel window decoding.

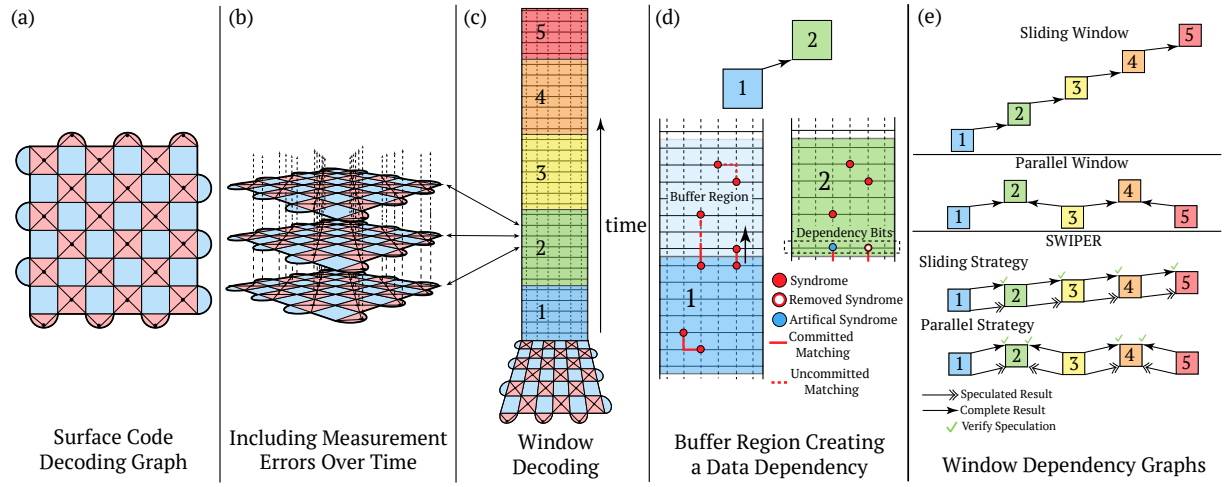

Figure 2: (a) A d = 7 surface code patch with Z(X) stabilizers indicated by red(blue) faces. Drawn is the decoding graph for Z stabilizers, with vertices of the graph indicating stabilizers and edges indicating error-prone data qubits. (b) Repeated rounds of measuring stabilizers. The decoding graph is augmented to include a temporal dimension with new edges indicating possible measurement errors. (c) A simplified view of decoding a surface code over time with one spatial dimension omitted. The decoding graph is split into windows labeled 1-5 which are each treated as a separate decoding problem. (d) Decoding of window 1 and its buffer region. Matchings in the window are committed and create Pauli frame updates while matchings crossing into the buffer region create either artificial syndromes or removed syndromes at the boundary. Dependency bits on the boundary are passed to window 2 and therefore constitute a data dependency between windows 1 and 2 (denoted by an arrow). (e) Dependency graphs in a sliding window strategy and parallel window strategy. Arrows indicate the presence of a buffer region belonging to the window at the source of the arrow. Also included are the same dependency graphs but when using SWIPER. Speculated dependencies (double arrows) allow SWIPER to start decoding windows sooner, and full decoding information is later used to verify the speculation (green check mark).

2 Background

For in-depth background, we refer to [22, 45, 50] for quantum computing fundamentals and [25, 30] for QEC and stabilizer codes. In this section, we give the necessary background on the decoding problem and prior windowed decoding approaches.

2.1 Surface Codes

The surface code, shown in Figure 2a, is a leading QEC code that fits into a 2D grid with nearest-neighbor connections. As a CSS code, it has two sets of stabilizers (Z, X) for decoding bit-flip errors and phase-flip errors, respectively, with each treated as a separate decoding problem. Operations on the surface code are typically formulated as lattice surgery [36], in which adjacent surface code patches are *merged* and *split* to perform logical operations. These primitives enable universal computation [24, 42] which has led to a growing set of resource estimates for large quantum programs [9, 29].

2.2 Quantum Error Correction Decoding

We presume that decoding takes place in the context of a long-running set of surface code patches, each generating syndrome data from their stabilizers. A decoding algorithm then operates on the syndrome data and aims to produce the most likely combination of physical errors that explains the observed syndromes. If all possible errors flip either a pair of stabilizers or a single stabilizer, as is the case with the surface code, we can use *matching* decoders [18, 26, 33]. In a matching decoder, we construct a decoding graph where nodes are stabilizers and edges are error mechanisms, such as data qubit errors and measurement errors. For the surface code, this graph is a 3D lattice, shown in Figure 2b. Decoding then consists of matching nodes with nonzero syndromes together to find a viable explanation for the observed syndromes. If the overall matching of syndromes is minimum weight, it is a good approximation for the most likely error.

2.3 Blocking Operations

Decoding results can be tracked in software for some time using Pauli frames [37, 51], omitting the need to apply immediate corrective operations. However, non-Clifford gates in the program constitute *blocking* operations which we cannot commute our Pauli frame past. Instead, a Clifford correction based on up-to-date decoding results must be applied physically to continue beyond the blocking operation [12]. For surface code computation, it is common to use T as the only non-Clifford gate, applied via a T gate teleportation circuit. In this case, the result of the measurement operation in the teleportation circuit must be fully decoded before the S gate correction can be applied [23, 58]. In T-based computation, the delay between this measurement operation and the conditional S correction is based on the decoder's *reaction time*, as shown in Figure 1. The presence of blocking operations therefore necessitates *real-time decoding*. In order to progress our quantum computation

past blocking operations, a classical decoder must operate in conjunction with the quantum device.

***Key Insight:* A delay in decoding a blocking operation causes a delay in the execution of a whole quantum program, so minimizing this reaction time is critical to ensure fast program runtimes**.

2.4 Decoding Windows

The latency of matching-based decoding algorithms grows polynomially with the decoding graph size, which places a practical limit on the size of a decoding problem. Instead of operating on the entire decoding history prior to each non-Clifford gate, we can instead operate on a pipeline of smaller *windows* of decoding data, shown in Figure 2c. In the original proposal, termed the overlapping recovery method [20], each window has a commit region and a buffer region, as shown in Figure 2d. In order to preserve the error-correcting performance of the underlying code, the buffer region should span $\sim d$ rounds, where d is the code distance. The commit and buffer regions together constitute a decoding task that is sent to the "inner" decoder. After a window is decoded, the matchings in the commit region are used to update the Pauli frame. Matchings that cross the boundary between the commit region and the buffer region can create "artificial" syndrome bits on the boundary that are passed to the next window. Similarly, any matchings from the commit region onto the boundary itself may remove a syndrome bit. The set of syndrome bits on the boundary between the commit and buffer regions therefore contains information that needs to be passed to the next window. We will refer to these bits as the *dependency bits*.

In general, buffer regions are needed to pass information related to whether a potential error string could cross the boundary between the commit regions of adjacent windows. A decoding window can also have multiple buffer regions if it is adjacent to multiple windows in space and time, which may occur during lattice surgery [41]. At a boundary between two adjacent windows, a choice must be made for which window contains the buffer region (and so must be decoded first). Its result is then passed to the other window via the dependency bits. To capture this, we can define each boundary in a decoding window's commit region as a "source" or "sink". Prior work has also referred to these as the "rough" and "smooth" boundaries, respectively. Source boundaries are followed by a buffer region and pass information to the adjacent window, while sink boundaries receive the passed information.

***Key Insight:* The choice of boundary types in each decoding window enforces an ordering of decoding problems and therefore has an important impact on overall decoding performance.**

2.5 Sliding Window Approach

The overlapping recovery method [20] is an example of *sliding window decoding*. Examining a single surface code over time, each window always starts with a sink boundary and ends with a source boundary, meaning data is always passed sequentially. As shown in Figure 2e, these dependencies mean that each window cannot begin decoding until the previous window is fully decoded. The decoding throughput is therefore equivalent to the decoding latency, so in order to avoid a backlog, the latency must be lower than the time to generate a new window.

We note that, to our knowledge, prior work has not examined the construction of spatially sliding windows in the context of lattice surgery operations. For completeness, however, we will assume that a sliding-window decoder uses a similar, feedforward approach for windows sliding along a spatial dimension.

2.6 Parallel Window Approach

In parallel window decoding [41, 55, 56], windows alternate between having all source boundaries and all sink boundaries. This minimizes the depth of the resulting dependency graph, shown in Figure 2e. Windows in the first layer have no data dependencies and can begin decoding immediately. Windows in the second layer are also independent from each other and can begin decoding after all their dependencies are complete. For a long-running quantum program, this occurs in a pipelined manner, allowing many separate windows to be actively decoded in parallel. For spatially parallel windowing of lattice surgery operations, an additional window type is needed that has a mix of source and sink boundaries [41].

3 Motivation

3.1 Decoding Latency

The latency of an inner decoder is not necessarily fixed and will generally vary with both the code distance and the specific decoding task (set of syndromes) at hand. Due to the complexity of decoding algorithms, in many cases the decoding latency will exceed the time it takes to generate a decoding window. Regimes exist where this is avoided, particularly smaller code distances with hardware-based implementations [6, 15, 43], however, in all of these the latency scales with the code distance d, with the highest distance achieved being $d = 23$, still below the distances expected for large-scale applications like factoring [29]. In Figure 3 we highlight this point by plotting decoding latencies using PyMatching 2, a state-of-the-art software decoder [33], and Stim [27], a detailed surface code simulator, with an assumed syndrome measurement round time of 1 µs based on recent experiments [2] and a physical error rate of $p = 10^{-3}$. We can see latency scales not only with code distance but also with the volume of the window, which varies with the number of buffer regions as discussed in Section 2.4. Furthermore, this issue of latency is not unique to matching-based decoders. Higher-accuracy decoders [7, 54, 61] and decoders for qLDPC codes [46] all struggle with decoding latency and therefore necessitate the use of an outer parallel window scheme to avoid a backlog. As such, we expect parallel window decoding to be a key ingredient in fault-tolerant quantum systems going forward.

3.2 Reaction Time

We assume that a quantum program running on a surface-code-based system will be decomposed into Clifford+T gates, where the T gates constitute the only blocking operation. This is the leading approach for compiling to surface codes, with some synthesis strategies even omitting Clifford gates entirely [42]. Since T gates are blocking operations, the reaction time of the decoder will determine how quickly T gates can be applied and, as a result, the overall program latency.

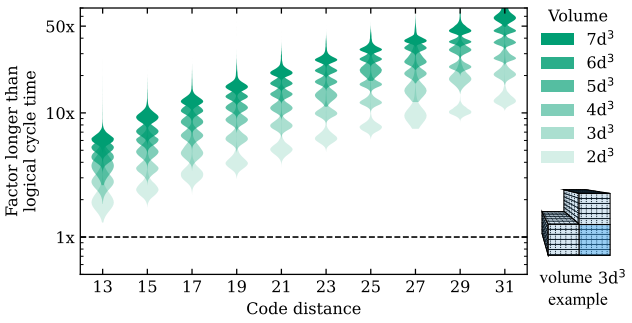

Figure 3: Sampled PyMatching 2 decoding latency distributions for varying window sizes. Latencies are relative to a logical cycle consisting of d measurement rounds where each measurement round is assumed to take 1 µs. *Lower right:* example of a $3d^3$ decoding volume in the style of Figure 2c.

In parallel window decoding, the reaction time of a decoding window is influenced by ① the latency of the inner decoder run on each window and ② the delays caused by waiting for window dependencies. In this work, we address ②, improving parallel window decoding in an inner-decoder-agnostic approach. With prior parallel windowing methods, ② constrains the reaction time for windows with dependencies to be at least $2t_w$, where t_w is the decoding latency. This is because a window with a dependency must wait until that dependency is *completely* finished decoding before it can begin decoding. However, we argue that this is not a fundamental constraint. In this work, we find that dependencies can be effectively predicted before the inner decoder is complete. SWIPER uses these speculations to reduce the impact of ②.

***Key Insight:* If the dependencies between windows can be resolved faster than the decoding latency, a window can begin decoding before its predecessors are complete.**

3.3 Key Definitions

Here we summarize three important, related concepts as used in this work.

- **Decoding Latency**: Time taken to **actively** decode a single window
- **Reaction Time**: Time taken from when a window is generated to when it is decoded. This includes the decoding latency and the time waiting for dependencies.
- **Decoder Throughput**: Rate at which windows finish being decoded. Should meet or exceed the rate at which windows are generated to avoid a backlog [58]

4 SWIPER: Speculative Window Decoding

In this section, we introduce SWIPER, a window decoder that includes a light-weight prediction step to speculate data dependencies between adjacent decoding windows. We organize this section as follows: in Section 4.1 we describe how SWIPER changes the reaction time of T gates, in Section 4.2 we outline a scalable predictor for surface code decoding, and in Section 4.3 we describe the classical decoder resources required by SWIPER. Then in Section 5 we describe the simulation methodology that we use and present benchmark evaluations comparing SWIPER to previous speculation-free window decoders.

4.1 T Gate Teleportation

Performing the T gate in the surface code requires the use of a T gate teleportation circuit, shown in Figure 1. The S gate correction occurs 50% of the time depending on the preceding measurement outcomes. Determining whether S should be applied is blocking and requires the decoder to be up-to-date through the ZZ merge operation (dark gray). As described in Figure 2e parallel window decoding requires that certain windows be dependent on future windows. This is seen in Figure 1 where windows 2 and 4 must wait to begin decoding until window 3 is completed at time $t + t_w$. In prior work, this is unavoidable; however, in SWIPER we speculate the dependencies, allowing windows 2 and 4 to begin decoding while window 3 is still decoding. After window 3 is fully decoded, we verify that the speculations were correct by comparing the speculated boundary-crossing matchings to the result of the full decoder. If verification succeeds, the resulting reaction time is reduced, allowing the program to continue past the blocking operation earlier than with prior work. Importantly, if verification fails upon completion of window 3, the reaction time is still no worse than in prior work, because we can restart windows 2 and 4 at time $t + t_w$.

***Key Insight:* Window dependency speculation will never worsen reaction times compared to baseline methods and will generally improve them significantly.**

4.2 Predictor Design

In designing a predictor, we leverage the commonly used fact that the majority of decoding problems will be simple with sparse, low-weight error chains [3, 48, 62]. Since most error chains are low-weight, we can predict these dependencies by looking for small, simple patterns along the boundary.

We develop our predictor design using an iterative approach. Here, we describe how we iterate on a simple, 1-step predictor to create a 3-step predictor that handles the majority of common syndrome patterns that create data dependencies. We define the overall *speculation accuracy* as the rate at which *all* dependency bits along a boundary are predicted correctly; any single mistake constitutes an incorrect speculation.

4.2.1 1-Step Predictor. In the 1-step predictor, we only look at single errors (edges in the decoding graph) that cross the boundary between a commit region and a buffer region. For each of these errors, the predictor checks their syndrome bits. If both bits are 1, it predicts that the error occurred and creates a dependency. We note that all errors can check their syndrome bits simultaneously, ensuring a low-latency, constant runtime prediction. Furthermore, for a distance d surface code, the number of errors that we need to check is only $O(d^2)$, which can be efficiently implemented in hardware logic.

In Figure 4 we plot the accuracy of the 1-step predictor as well as a breakdown of misprediction cases. A false positive is when a matching crosses the boundary was predicted but did not actually exist and a false negative is when a matching that did cross the

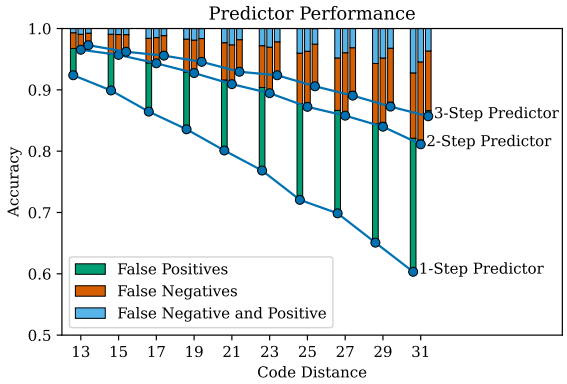

Figure 4: Accuracy of the three different predictors for increasing values of code distance. Bars indicate a breakdown of different failure cases.

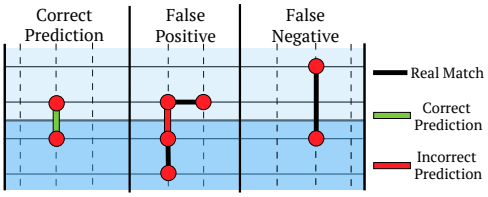

Figure 5: Example common cases for the 1-step predictor.

boundary was not predicted. We generate circuit-level surface code windows at a physical qubit error rate of 0.1% using Stim [27] and use PyMatching [34] as the reference decoder to determine the misprediction. Despite its simplicity, the 1-step predictor can still reach > 70% accuracy for code distances up to 25.

The 1-step predictor can be further improved by studying common failure cases. In Figure 5 we describe the most common cases that the 1-step predictor encounters. Although all failure cases are equally damaging and constitute a misprediction, from Figure 4 we find that false positives are the most frequent mistakes made by the 1-step predictor.

4.2.2 2-Step Predictor. To address common false positives in the 1-step predictor, we propose an improved 2-step predictor. The intuition for the 2-step predictor comes from the fact that, after step 1, we may have clusters of errors that need to be pruned to create a minimal matching. We therefore design step 2 similarly to the peeling decoder introduced in [19] where we prioritize matching syndrome bits with the fewest viable matches, akin to leaf nodes.

We increase the set of errors considered to include all errors at most a distance of 2 from the boundary. In the first step, like the 1-step predictor, each error checks if both its syndrome bits are 1. If so, instead of declaring a match, they increment both syndrome bits by 1. In the second step, each error that believed itself to be a match is assigned to a bin based on the sum of its adjacent syndrome bits. Then in increasing bin order, each error checks if its syndrome bits are nonzero. If so, it declares itself a match and sets both syndrome bits to zero. Importantly, the number of bins is limited by the degree

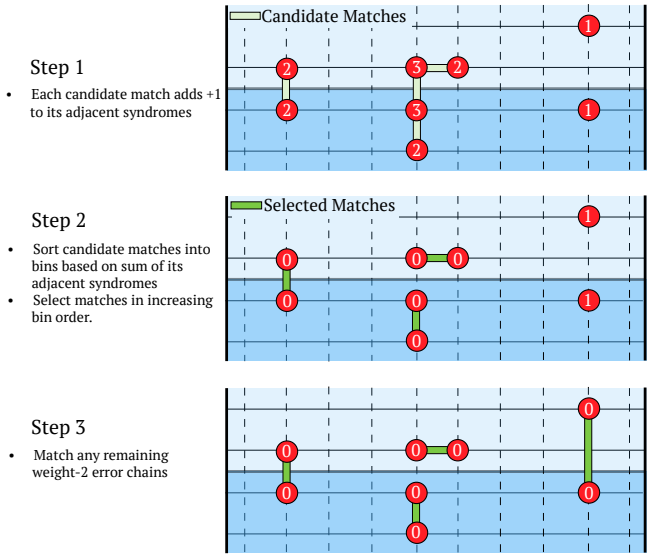

Figure 6: An overview of the final, 3-step predictor logic.

of vertices in the decoding graph, which for the surface code is constant due to its constant weight-4 parity checks. The runtime of this step is therefore independent of the code distance. In Figure 4 we find that the 2-step predictor resolves nearly all false positive cases mispredicted by the 1-step predictor.

4.2.3 3-Step Predictor. Finally, to address common false negative cases, we introduce a third step to the 2-step predictor. Offline, we can precompute the pairs of syndrome bits that can be matched by a weight-2 error chain crossing the boundary. Then, after the 2-step predictor determines its matches, we check each precomputed pair to see whether its syndrome bits are both 1 in the decoding graph, and if so, we declare a match. Similarly to the 1-step predictor, these checks can all occur in parallel along the boundary, ensuring a constant runtime with the number of such checks as $O(d^2)$. We summarize the logic of this final 3-step predictor in Figure 6.

In Figure 4 we can see that the 3-step predictor reduces the amount of false negatives produced by the 1 and 2-step predictors, achieving a prediction accuracy of > 90% for code distances up to 23. It is likely that more complex predictors could reach even higher accuracies, but we leave this as future work.

***Key Insight:* The 3-step predictor runs in time $O(1)$ since the number of unique bins in step 2 depends on the parity check weight, which is constant for the surface code.**

4.2.4 Hardware Implementation. To validate the behavior of the 3-step predictor, we implemented the algorithm on FPGA hardware. The FPGA environment was chosen as FPGAs have seen value as platforms for full QEC decoders [43, 63]. We performed a behavioral simulation of the 3-step predictor in AMD's Vivado Design Suite [4] to verify the $O(1)$ runtime for $d = 13$ through $d = 27$. The time taken for the predictor implementation is always 60ns, demonstrating that it is constant with respect to distance. Since the 3-step predictor only considers $O(d^2)$ syndrome bits compared

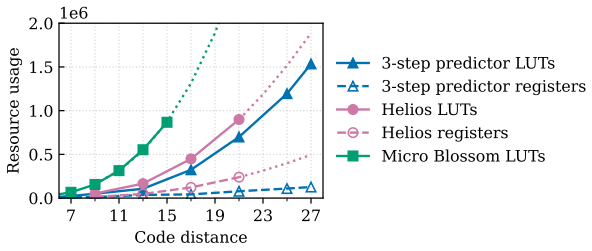

Figure 7: FPGA resource usage of 3-step predictor compared to FPGA-optimized full decoders ([43, 63]). Dotted lines represent fits to reported asymptotic costs.

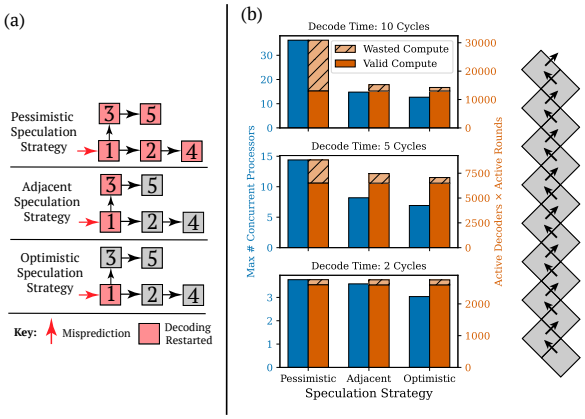

Figure 8: (a) Three different speculation strategies for handling a misprediction which has poisoned window 1. (b) Estimating classical decoder costs of the three different proposed speculation strategies. All values are averaged over 10,000 shots. The specific case simulated is 100 windows in a "zigzag" shape (right) using a sliding schedule with speculation. In the breakdown, "valid compute" indicates compute that finished and was correct, whereas "wasted compute" indicates compute that was restarted early or found to be incorrect.

to the full decoding volume of $O(d^3)$, we also expect area costs to scale favorably compared to the full decoder.

As a rough comparison, in Figure 7 we plot resource usage of our 3-step predictor implementation versus full decoder FPGA implementations [43, 63]. As expected, we find that the 3-step predictor has lower resource costs and more favorable scaling than a full decoder due to its restriction to processing smaller decoding volumes at window boundaries. We expect costs to be further reduced with future implementation optimizations.

4.3 Classical Resources

Parallel window decoding will need access to many classical decoders operating in parallel to keep up with throughput demands. Compared to prior speculation-free schemes, SWIPER collapses the dependency structure of windows, which incurs an increased demand on the number of concurrent decoders. Given that parallel window decoding relaxes latency requirements, for this work, we assume decoders exist out-of-fridge and are much less costly than the quantum device itself. However, here we study how to minimize the required classical costs of SWIPER. In particular, incorrect speculation can cause wasted classical computation as it requires restarting some decoding tasks. SWIPER mitigates this cost through an *optimistic speculation strategy*.

4.3.1 Handling Mispredictions. A waiting decoding window begins decoding after its incoming dependencies have passed boundary speculations to it. However, once each dependency is fully decoded, we must verify if the speculation was correct by comparing the dependency bits produced by the predictor and by the full decoder. In the event that the predictor was incorrect, we have a *misprediction* The relevant decoding window operated on incorrect syndrome data, leading to an invalid solution; the decoding task must be restarted using correct dependency bits. We call this window *poisoned*. However, incorrect matching of syndromes along one boundary can lead to incorrect matching of other syndromes on other boundaries, so the effects of a misprediction may propagate further in the window dependency graph. This raises the important question: which other decoding windows are unreliable in the event of a misprediction?

Given a dependency graph with a poisoned node, shown in Figure 8a, the *pessimistic speculation strategy* (left) restarts all descendants that have already begun decoding. If the poisoned root significantly increases the chances of these descendants being poisoned, this strategy is effective, as it halts likely-incorrect decoding tasks earlier and avoids needless computation. However, if the descendants' chances of success are not significantly affected, this is a poor strategy as it throws away valid, in-progress decoding tasks that are likely to be correct. As discussed in Section 4.2, we expect most decoding problems to be sparse with low-weight error chains. In this regime, we do not expect a change in syndrome bits on one boundary of a window to change a result on a different boundary. Therefore, an *optimistic speculation strategy* (Figure 8a, right) only restarts the poisoned node itself.

To evaluate this intuition, we use Stim and PyMatching to simulate the surface code at a physical qubit error rate of 0.1% and estimate the conditional probability of a misprediction of a source boundary given a received misprediction on a separate sink boundary. We find no change in accuracy if the two boundaries are not adjacent (e.g. one temporal boundary to the next) and a $\approx 4\%$ decrease in accuracy if the two boundaries are adjacent (e.g. one spatial boundary and one temporal boundary). Given this asymmetry, we propose an intermediate strategy that restarts the poisoned node and any windows that used a prediction on an *adjacent* boundary to the misprediction, shown in Figure 8a (center).

In Figure 8b, we show the classical costs for decoding a sequence of 100 windows with speculation failures using the three speculation strategies detailed above. We use a prediction accuracy of 90% with a reduced accuracy of 86% for boundaries adjacent to a misprediction. If the decode time is comparable to the time to generate a window (1 cycle), we see little variance, since the depth of the active dependency tree is small. However, at large decode times more similar to what we observed in Figure 3, we see that the pessimistic

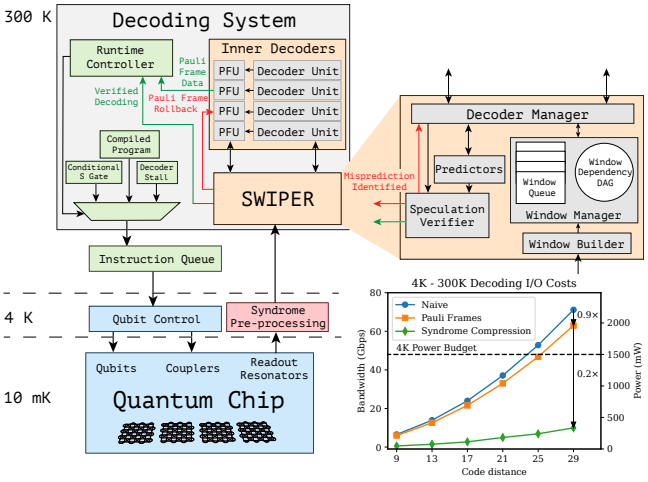

Figure 9: Overview of SWIPER's interaction with a decoding system for superconducting quantum computers. The bottom right plot estimates the bandwidth and power used to communicate between 4K and 300K for decoding. Utilizing Pauli frames and a pre-processing step such as sparse syndrome compression significantly reduce costs.

strategy unnecessarily restarts more windows, wasting classical compute. In all cases, we find that the optimistic strategy performs the best, which we attribute to the high prediction accuracy and minimal downstream effects in the case of mispredictions.

4.4 System Overview

Figure 9 gives an overview of how SWIPER interacts with a decoding system for a superconducting quantum computer. Since SWIPER is designed to be agnostic to the inner decoder, we assume that decoder units simply operate out-of-fridge. In such a system, an important consideration is the impact that sending real-time data between the fridge (4K) and room temperature (300K) has on scalability. We discuss this here briefly, but emphasize that SWIPER is compatible with prior and future work optimizing the underlying decoder, such as operating decoders in-fridge [11, 35, 48]. Using SWIPER with such decoders improves their scalability, removing 1 μs as a latency constraint, which, as described in Section 3.1, proves to be exceedingly difficult for the large code distances needed in factoring and chemistry applications.

4.4.1 Bandwidth and Power. Communicating data between 4K and 300K can stress the cooling capacity of a dilution refrigerator due to potentially many communication cables that will dissipate heat [38]. In this context a large contributor can be the quantum control itself [57] where physical gate pulses have an order of magnitude larger power than readout [38]. Fortunately, this can be alleviated by placing the quantum controller in-fridge and instead sending *logical-level* instructions [11, 57]. The use of Pauli frames further improves over this baseline by eliminating the need to send physical corrections back into the fridge. Additionally, since corrections take the form of deciding whether an S gate is applied after each T gate teleportation, which should occur with a 50% chance, the instruction bandwidth is not affected by SWIPER.

Syndrome readout can also stress the system. However, since syndromes are typically sparse and low-weight we can make use of cheap preprocessing steps [16, 48] to reduce bandwidth. In the bottom right plot of Figure 9 we estimate the reduction in I/O costs when applying two optimizations: Pauli frames and an example syndrome preprocessing step, sparse syndrome compression. Simulations are based on 50 simulated surface codes with an error rate of 0.1%. We assume a bandwidth per coaxial cable of 1 Gbps and power per coaxial cable of 31 mW. This is a conservative assumption based on prior work assuming 10 Gbps with 31 mW [11, 32] and 1 Gbps with 11.5 mW [38, 60]. We find bandwidth and power scale favorably, and we note that costs can be further reduced by using higher bandwidth cables such as optical fibers [65].

Beyond bandwidth and power, previous justification for in-fridge decoders has been to keep decoder *latency* below 1μs [11]. However, since SWIPER is designed to tolerate large latencies while meeting throughput demands, the ubiquitous 1μs goal is no longer required, removing a key constraint underlying prior work.

***Key Insight:* SWIPER does not increase 4K-300K I/O costs, as corrections are tracked via Pauli frames and are 'applied' by deciding if an S gate should occur after a T gate.**

4.4.2 Mispredictions. In Section 4.3.1 we describe SWIPER's policy for handling misprediction. Here, we clarify how mispredictions interact with the broader decoding system. Specifically, if SWIPER encounters a misprediction, the optimal speculation strategy dictates we restart only the poisoned window. However, if the poisoned window was already complete, then it committed an invalid value to the corresponding Pauli frame. SWIPER therefore issues a *Pauli frame rollback*. The Pauli frame rollback removes the Pauli record corresponding to the poisoned window from the Pauli Frame Unit (PFU). Furthermore, since deciding whether an S gate is applied after a T gate requires that all decoding results be verified, *mispredictions do not propagate to the quantum device.*

***Key Insight:* Before deciding if an S gate occurs after a T gate, all mispredictions are resolved as Pauli frame rollbacks and all decoding results are verified. Mispredictions therefore do not affect 4K-300K I/O costs.**

5 Methodology

To evaluate the impact of SWIPER on the overall program latency, we perform benchmark evaluations using state-of-the-art compilation techniques. To do so, we develop our own simulator, SWIPER-SIM, to evaluate window decoders given lattice surgery programs.

5.1 Simulation Software

Figure 10 gives an overview of the procedure by which we compile and evaluate benchmark applications. We source relevant benchmarks for the fault-tolerant regime from recent repositories [31, 40, 47, 53] which can be compiled down to Clifford+RZ gates in OpenQASM [14] using Cirq [21]. We then use Gridsynth [52] to approximate RZ gates as a sequence of H, S, and T gates with a precision of 10^{-10}. We feed the resulting Clifford+T circuit into the Lattice Surgery Compiler [39] to create a mapped and routed

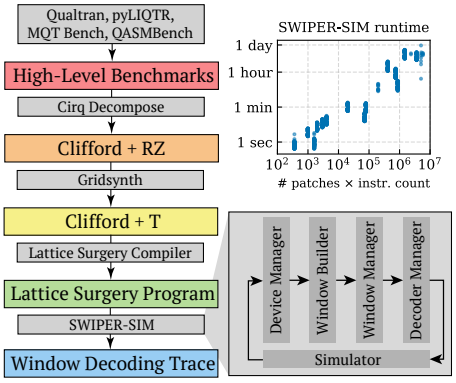

Figure 10: The pipeline used to evaluate high-level benchmark programs with different window decoders, including details of SWIPER-SIM. *Inset:* SWIPER-SIM runtime versus program size on a single core of an Intel Xeon Gold 6248R processor.

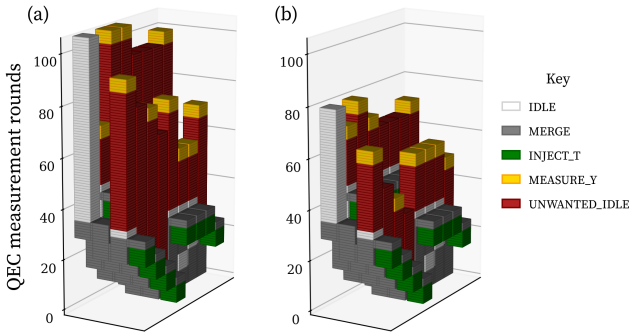

Figure 11: SWIPER-SIM program traces for a 15-to-1 magic state distillation in a distance-7 code using the construction from [24] (Fig. 17). Time advances vertically, and each horizontal slice represents a batch of syndrome data (colored by instruction type). Decoding time is fixed to be the same as the time to generate 2d rounds, about twice the window generation rate. Device traces are shown for (a) baseline parallel window method (15.1d QEC rounds) and (b) SWIPER aligned window (11.3d QEC rounds, a 25% improvement).

lattice surgery program using the Edge-Disjoint Paths Compilation (EDPC) surface code layout [8].

SWIPER-SIM takes in a compiled lattice surgery program and performs a round-level simulation of syndrome generation, windowing, and decoding (the final step in Figure 10). Based on the lattice surgery program, the DeviceManager generates a set of syndrome rounds every 1 µs (based on recent experimental timing [2]) for all currently active surface code patches. These syndrome rounds are collected by the WindowBuilder and assembled into windows by the WindowManager, with source/sink boundaries decided based on a window decoding strategy (e.g. sliding, parallel). Completed windows are sent to the DecoderManager, which initiates speculation and decoding tasks when the relevant dependencies are satisfied, manages classical compute resources, and handles mispredictions. Speculation uses the optimistic strategy described in Section 4.3.1 to handle mispredictions. For blocking instructions, the DeviceManager delays the conditional instruction until the DecoderManager signals that it has fully decoded (and verified) the instruction history up to and including the blocking operation.

By simulating at the granularity of rounds, windows do not necessarily need to align with instructions. As a result, idling while a blocking T gate is being decoded can complete as soon as possible, even if the reaction time is not a multiple of the window size. As an example, Figure 11 shows a spacetime program trace generated by SWIPER-SIM for a hand-specified lattice surgery program for 15-to-1 magic state distillation [10, 24, 49]. We can compare prior work (Figure 11a) with SWIPER (Figure 11b) to see the reduction in reaction time for blocking operations when using speculation. Slices are colored by instruction type. Y-basis measurement and S gates are modeled according to [28].

5.2 Studying reaction times with SWIPER

In Figure 12 we plot the sensitivity of SWIPER to decoding latency and speculation accuracy. Decoding latency is proportional to window volume with relative factor r. For sliding windows of size $2d^3$ (d^3 commit, d^3 buffer), $r = 0.5$ corresponds to a latency of d rounds, matching the window generation rate; as expected, we see that the backlog problem for default sliding window decoding (yellow dashed line) therefore begins when $r > 0.5$. SWIPER mitigates this problem with speculation, but we find that reaction time for sliding windows is particularly sensitive to speculation accuracy. This is due to the depth of the dependency tree, which is unbounded in sliding window decoding.

Equivalent latency factors extracted from linear fits to PyMatching latency data (Figure 3) are shown along the x axis in green for code distances corresponding to the so-called *gigaquop* regime (10^{-9} logical error rate, $d = 15$), the *teraquop* regime (10^{-12}, $d = 21$), and the *petaquop* regime (10^{-15}, $d = 27$) evaluated at $p = 10^{-3}$. We additionally highlight these three regimes for a hypothetical high-accuracy RNN decoder inspired by [7], which we assume to have similar latency to PyMatching for a fixed code distance but 4× reduced logical error rate, reducing the required code distances to $d = 13, 19$, and 25 respectively.

5.2.1 T Gate Alignment. We also notice that, in parallel window decoding, *the type of window a T gate aligns with affects its reaction time.* As shown in the bottom example of Figure 13, if the merge operation in a T gate teleportation ends with a sink boundary, the reaction time can be further delayed by the time to generate the source boundary's window in the future. In this specific example, since window 3 depends on window 4, it must wait for window 4 to be generated before it can even begin decoding. However, if instead the merge ends in a source boundary, as shown with window 8 in the top example, the soonest window 8 can begin decoding is after its buffer region is generated, which is only d rounds.

More generally, we conclude that a blocking operation should always be "aligned" (have a future-facing source boundary) to ensure minimal reaction time. Since a sliding window schedule always ends with source boundaries, it is always aligned. However, the

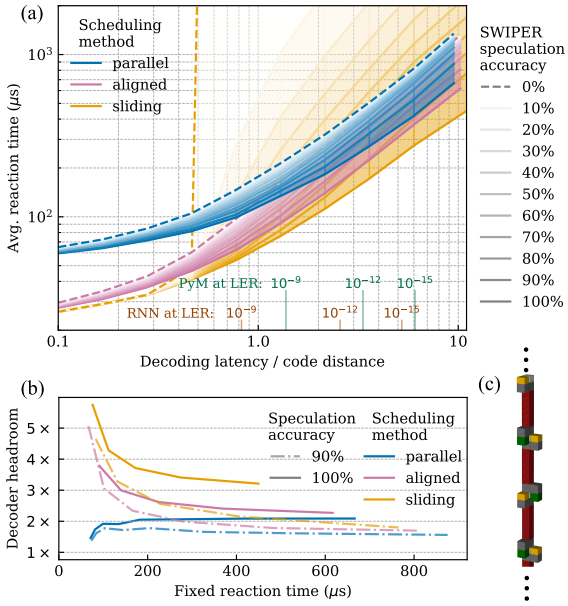

Figure 12: (a) Sensitivity of reaction time to decoding latency and speculation accuracy. Speculation accuracy of 0% corresponds to baseline (no speculation) decoding. Decoder latency is $t_{dec}(v) = rv/d^2$ rounds, where v is the volume of the decoding problem (in units of d^3) and the relative latency factor r is varied along the x axis. Equivalent latency factors extracted from linear fits to PyMatching latency data and a hypothetical high-accuracy RNN decoder at varying target logical error rates are also shown for comparison. (b) For a fixed reaction time, SWIPER allows for up to 5× longer decoder latency compared to the parallel-window baseline. (c) Simulator trace of repeated T gates and S corrections on an idling logical qubit. A similar experiment with 1000 T gates was used to collect the data in this figure.

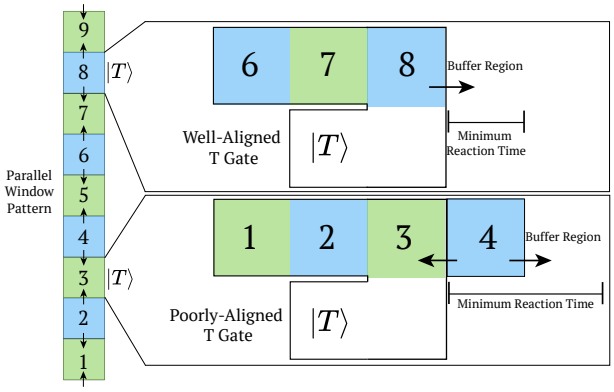

Figure 13: Comparing a well-aligned T gate (top) and a poorly-aligned T gate (bottom).

sliding window schedule is much more sensitive to speculation accuracy, leaving it vulnerable to the backlog problem. To address this, we also introduce an *aligned* window strategy, which is a parallel window strategy with forced alignment of blocking operations.

***Key Insight:* Using SWIPER-SIM, we find that parallel window decoders suffer from dependency-induced delays, as already discussed, but also *alignment*-induced delays.**

5.2.2 SWIPER's effect on reaction times. In Figure 12 a, we find that the aligned strategy retains the misprediction resiliency of parallel window decoding while reducing reaction time. We see that for short decoding latency, the aligned strategy yields over 50% shorter reaction times than the parallel strategy. As decoder latency increases, parallel and aligned strategies become increasingly similar; reaction time in this regime is dominated by waiting for dependencies to resolve rather than generating windows. For both aligned and parallel, we see that SWIPER can reduce reaction times by roughly 50% when decoding latency is long; we attribute this to SWIPER effectively "flattening" the two-layer dependency structure of the parallel window strategy. A speculation accuracy of 90% is sufficient to get most of the benefit of SWIPER. Finally, we see that if the speculation is reliable enough, sliding window decoding performs best at all decoder latencies, which we attribute to the use of smaller window sizes (typically $2d^3$, compared to the $3d^3$-volume windows in parallel window decoding) leading to lower inner decoder latencies.

5.2.3 Relaxing Inner Decoder Latency. Figure 12b shows that, in a setting with a fixed runtime budget, SWIPER allows for significantly longer inner decoder latencies. As shown in Figure 13, well-aligned T gates have their reaction times shortened by dt_{cycle} compared to poorly-aligned T gates; in the regime where decoding time t_{dec} is fast compared to t_{cycle}, the same overall reaction time can be achieved with a significantly slower decoder. For fixed reaction time values, we compare high-accuracy SWIPER to the default parallel window scheme and find that the speculated aligned and sliding schemes allow upward of 5× increased latency for reaction times near 100 μs and all three (parallel, aligned, and sliding) speculated methods allow 2× increased latency in the limit of large reaction time (500 μs+).

This relaxation in decoder latency requirements could be translated into an increase in program fidelity by using recurrent, transformer-based decoders [7], tensor network decoders [1, 13], or ensemble decoders [54] which have demonstrated improvements to decoder accuracy at the cost of decoder latency.

5.3 Benchmark Simulation

To further evaluate the efficacy of SWIPER, we select a suite of fault-tolerant benchmark applications and use the methodology described in Section 5.1 to simulate the program runtime when using different window decoding strategies. To evaluate at the scales expected for large, fault-tolerant applications, we consider a physical error rate of $p = 10^{-3}$ and code distance $d = 21$, which corresponds to the *teraquop regime* with logical error rates below 10^{-12}. We again assume that each QEC syndrome round takes 1 μs and we sample decoding latencies from the distributions shown in Figure 3, rounding up if the window volume is not an integer

Benchmark	Short name	Footprint	Physical qubits	T count	Source	Domain
Magic State Distillation	msd_15to1	32	28192	15	[24]	Resources, μBenchmark
Toffoli	toffoli	9	7929	7	[45]	μBenchmark
RZ(ϕ), $\epsilon = 10^{-10}$	rz	4	3524	100	[52]	μBenchmark
Quantum Read Only Memory	qrom	153	134793	48	[31]	Data I/O
8-bit Adder	adder_8bit	111	97791	112	[40]	Factoring subroutine
Carleman Encoding	carleman	264	232584	394	[53]	ODE
H_2 Qubitized Walk Operator	H2_molecule	92	81052	3084	[53]	Chemistry
Fermi Hubbard (2, 2) Kagome	fh_kagome	181	159461	3340	[53]	Chemistry
Fermi Hubbard 4×4 Grid	fh_square	225	198225	3898	[53]	Chemistry
Quantum Phase Estimation	qpe	85	74885	11378	[47]	Common subroutine
Quantum Fourier Transform	qft	86	75766	11484	[47]	Factoring subroutine
Heisenberg Encoding	heisenberg	107	94267	12833	[53]	Chemistry
Grover's Algorithm	grover	34	29954	13577	[47]	Data Search

Table 1: Selected benchmark applications. Footprint is the number of $d \times d$ surface code patches used. Physical qubit count is calculated at $d = 21$.

multiple of d^3. Speculation is assumed to take 1 µs with an accuracy of 90% based on our results in Section 4.2.

5.3.1 Selected Benchmarks. Table 1 summarizes the lattice surgery program benchmarks we simulate. We identify three "microbenchmarks" (μBenchmarks) which are small, consistently-used primitive operations in the fault-tolerant domain whose performance can be extrapolated to estimate that of programs beyond what we include in Table 1.

For the other benchmarks, we include exact quantum phase estimation on 5 qubits due to its presence as a common subroutine in many quantum algorithms. We include Quantum Read Only Memory (QROM) with 15 data bits and 15 select bits to represent data I/O in many quantum algorithms. Grover's algorithm on 5 qubits is chosen to represent data search applications. We include block encoding for Carleman linearization with 4 truncation steps to represent quantum algorithms for nonlinear differential equations. We include the Quantum Fourier Transform (QFT) on 10 qubits and an 8-bit adder to represent subroutines in factoring applications. Finally, simulation of the Fermi-Hubbard model on a 4×4 lattice and performing qubitized ground-state energy estimation of H_2 [5] are included to represent chemistry applications.

5.3.2 Results. Figure 14a shows the program runtimes for the selected benchmarks relative to the baseline parallel window method. Due to the uncertainty in runtime for smaller benchmarks (discussed in the following subsection), we report aggregate results for benchmarks with at least 1000 T gates: without SWIPER, the aligned scheduling method achieves a 4.3% to 11.8% (geomean 6.3%) reduction in runtime compared to parallel window. With SWIPER, all three scheduling methods (parallel, aligned, and sliding) improve significantly. SWIPER-parallel achieves **31.5% to 35.4% (geomean 32.9%)** reduction in runtimes, SWIPER-aligned achieves **36.9% to 40.6% (geomean 38.4%)** reduction in runtimes, and SWIPER-sliding achieves **40.4% to 43.6% (geomean 41.5%)** reduction in runtimes compared to baseline parallel window. We observe that the QROM and Carleman encoding benchmarks exhibit slightly less performance improvement compared to the other benchmarks; we conclude that this is because these two benchmarks have a lower fraction of T instructions (~40% compared to ~70-80% for the other benchmarks), so the benefit of reducing T reaction time is slightly less impactful.

5.3.3 Overhead due to missed speculations. We can determine the impact of missed speculations on runtime by comparing the runtimes of SWIPER for the default 90% and an idealized 100% speculation accuracy (the difference is shown as the ligher portion of the bars in Figure 14a). For benchmarks with at least 1000 T gates, we observe 12.5% to 28.7% (geomean 20.2%) runtime overhead due to speculation failures. The overhead is higher for SWIPER-sliding (geomean 26.2%) and lower for SWIPER-aligned (geomean 22.9%) and SWIPER-parallel (geomean 13.7%), which can be understood by recognizing that higher-depth window dependency structures lead to higher missed speculation costs, as was explored in Figure 12.

5.3.4 Runtime uncertainty and extrapolation. It is important to note that there is uncertainty in the runtime of the benchmark programs we simulate. Randomness is introduced in two ways in our simulations: ① conditional S gates, which are applied 50% of the time after a T gate teleportation, and ② mispredictions in SWIPER. To capture this uncertainty, we run 10 trials of each benchmark with fewer than 3,000 T gates. For these benchmarks, the runtimes in Figure 14a have error bars showing the standard deviation of the results. In Figure 14b-c, we show that this standard deviation (relative to mean) decreases as the T count increases while the relative improvement over the baseline remains constant. We therefore claim that in the limit of the large T counts we expect in future fault-tolerant algorithms, SWIPER will show performance improvements similar to those in the larger benchmarks in our study. We also note that the amount of uncertainty does not appear to depend strongly on the window strategy being used, indicating that most of the variation stems from ① (the conditional S gate) rather than ② (mispredictions in SWIPER). This can be explained by the fact that conditional S gates occur with 50% probability, whereas mispredictions occur only with 10% probability.

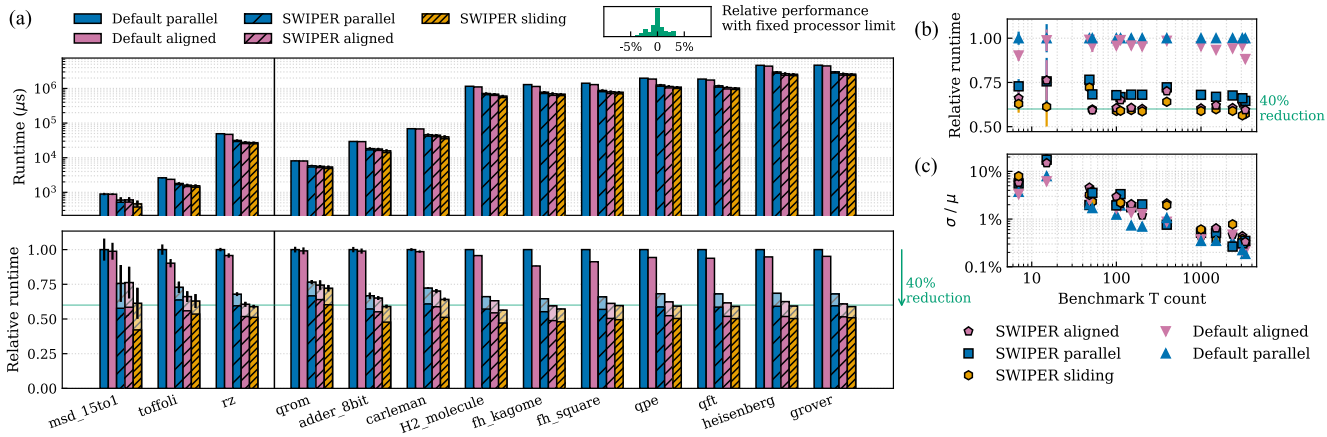

Figure 14: (a) Program runtime for selected benchmarks under different window scheduling methods, with and without SWIPER. Bottom histogram shows relative runtimes normalized to default parallel window runtime. Classical processor resources are unlimited. Vertical black lines indicate standard deviation over 10 randomized trials for smaller benchmarks. For SWIPER, lighter portion of bar shows the contribution of missed speculations to runtime. *Top right:* Relative differences in runtime when using auto-limited processor count (Section 5.3.5) for benchmarks with runtime longer than 10^4 μs. (b) Performance improvement of SWIPER appears to be consistent across benchmarks size. (c) Relative standard deviations in runtime over randomized trials decreases as programs get larger.

Looking ahead, because SWIPER's main benefit is reducing the reaction time for T gates, which are typically the main cost of a program [9, 42], the magnitude of runtime improvement is broadly independent of program size, so we expect we would see similar improvements for full application-level benchmarks.

5.3.5 Classical overhead of SWIPER. SWIPER takes the perspective that a modest increase in classical workload is worth an improvement in quantum computer speed. Nevertheless, it is important to quantify the extra classical resources required to implement SWIPER. While in the majority of this work we assume the number of classical decoders is not limited, in practice we must instantiate a finite number of decoders to implement SWIPER on real hardware. As speculation failures occur probabilistically, we cannot know the exact optimal number of classical decoding processors to provision. We suggest a simple heuristic to provision an appropriate number of processors: we simulate the program of interest in the perfect-speculation, unlimited-processor case and track $P_{\max}$ (maximum number of parallel processes) and P_{mean} (mean number of parallel processes) over all rounds of the program. We then allocate $P_{\max} + \epsilon_{\text{spec}} P_{\text{mean}}$ processors to SWIPER, where ϵ_{spec} is the probability of a misprediction (10% in our evaluation). The intuition for this heuristic is that $P_{\max}$ is the number of processors required by the program and $\epsilon_{\text{spec}} P_{\text{mean}}$ is the expected number of processors needed to handle mispredicted windows that are being redecoded.

We do not observe any significant variation in simulated program runtime with this limit imposed. Figure 15a compares the unlimited-processor usage to this auto-set processor limit, showing that the heuristic is able to accurately estimate the true number of required processors without bottlenecking the decoder, setting the processor limit very near to the actual maximum required. A linear fit to this data also reveals that SWIPER uses approximately 24% more

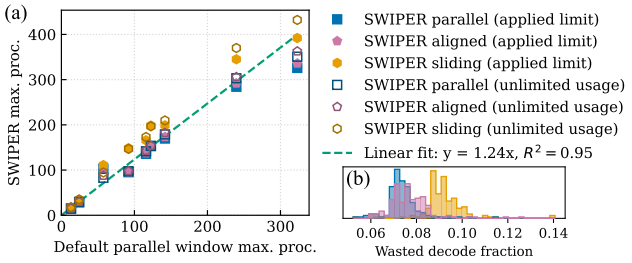

Figure 15: (a) Comparing classical compute cost between baseline method and SWIPER for selected benchmarks. (b) Fraction of total decoder processing time spent on tasks that were discarded.

simultaneous decoding processors than the default method. We argue that this extra cost is worth the significant improvement in quantum program runtime, as classical hardware is inexpensive compared to a large-scale quantum computer.

Furthermore, we quantify the amount of classical decoder computation that is "wasted" on decoding tasks that are based on incorrect speculations. A histogram of the fraction of wasted decoding computation cycles is shown in Figure 15b for the three scheduling methods. We see that SWIPER-sliding (yellow) incurs more wasted computation than parallel and aligned, as expected considering the deeper dependency structure of windows in a sliding scheme.

6 Related Work

While QEC decoding has broadly been an active area of research [11, 17, 44, 59], related work on decoder performance has

largely focused on achieving a decoding latency within 1 µs as a requirement for real-time decoding, motivated by the fact that superconducting systems generate a round of syndrome measurements every ∼ 1 µs [2]. LILLIPUT [15] proposes a look-up table that can decode up to $d = 5$ within 1 µs. Astrea [62] extends this to $d = 9$ in 1 µs via brute-force searching low-Hamming-weight bitstrings. Clique [48] is a lightweight predecoder that specifically addresses isolated errors and bears some resemblance to the 1-Step Predictor proposed in this work. More recently, Promatch [3] proposes an adaptive predecoding step to lift this limit to $d = 13$ within 1 µs in regimes of low physical error rate (0.01%). Barber et al. [6] design a Collision Clustering decoder and an FPGA implementation that can reach $d = 23$ in under 1 µs. Helios [43] demonstrates an impressive implementation of the Union-Find decoder [18] on an FPGA and achieves latencies of 11.5 ns for one round of a $d = 21$ surface code under a simpler phenomenological error model. Although all of these works improve decoding latencies, the advent of parallel window decoders presents a scalable "outer level" solution to real-time decoding that removes 1 µs as a hard requirement for decoding latency. Relaxing the constraints on the inner decoder also allows decoders which historically have had prohibitive latencies, such as higher-accuracy decoders [7, 54, 61] and decoders for qLDPC codes [46].

Prior work specifically addressing parallel window decoders is still limited. The original proposals for parallel window decoding [55, 56], work analyzing their performance for spatial windows during lattice surgery [41], and work extending this to transversal gate computation for high-connectivity systems [66] all assume that windows with dependencies wait until their dependencies are completely decoded. SWIPER, however, proposes key differences. Our novel introduction of a speculation step allows us to reduce the reaction time of T gates and in turn fault-tolerant program runtimes by 40%. We are also the first work to simulate general lattice surgery programs in the context of window decoding.

7 Conclusion and Future Work

In this work, we proposed SWIPER, a parallel window decoder that introduces a light-weight speculation step to resolve data dependencies between adajcent surface code decoding windows. There are a number of interesting directions for future work to explore. While in this work we design an independent predictor, exploring whether iterative decoding algorithms [34, 64] could admit a high-quality prediction from an intermediate state could further reduce classical resources. Additionally, SWIPER focuses on surface codes, but parallel window decoding is also compatible with a broader set of codes, including qLDPC codes [55]. Designing a predictor in this setting could further extend the benefits of SWIPER.

Author contributions

J.V. conceived of the idea, designed the 3-step predictor, designed the aligned scheduling strategy, and wrote the compilation pipeline. J.D.C. developed SWIPER-SIM and ran benchmark simulations. S.J. performed FPGA evaluations of the predictor. G.S.R., Y.L., and F.T.C. advised the project. All authors revised the manuscript.

Acknowledgments

We thank Pranav Gokhale, Kevin Gui, and Tina Oberoi for feedback on an earlier version of this work.

This work is funded in part by EPiQC, an NSF Expedition in Computing, under award CCF-1730449; in part by STAQ under award NSF Phy-1818914/232580; in part by NSF award 2340516; in part by the US Department of Energy Office of Advanced Scientific Computing Research, Accelerated Research for Quantum Computing Program; and in part by the NSF Quantum Leap Challenge Institute for Hybrid Quantum Architectures and Networks (NSF Award 2016136), in part based upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, and in part by the Army Research Office under Grant Number W911NF-23-1-0077. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Accelerated Research in Quantum Computing under Award Number DE-SC0025633. This work was completed in part with resources provided by the University of Chicago's Research Computing Center. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. FTC is the Chief Scientist for Quantum Software at Infleqtion and an advisor to Quantum Circuits, Inc.

References

[1] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681.

[2] Google Quantum AI and Collaborators. 2024. Quantum error correction below the surface code threshold. *arXiv preprint arXiv:2408.13687* (2024).

[3] Narges Alavisamani, Suhas Vittal, Ramin Ayanzadeh, Poulami Das, and Moinuddin Qureshi. 2024. Promatch: Extending the Reach of Real-Time Quantum Error Correction with Adaptive Predecoding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 818–833.

[4] AMD. 2024. *AMD Vivado™ Design Suite*. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[5] Ryan Babbush, Craig Gidney, Dominic W Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. 2018. Encoding electronic spectra in quantum circuits with linear T complexity. *Physical Review X* 8, 4 (2018), 041015.

[6] Ben Barber, Kenton M Barnes, Tomasz Bialas, Okan Buğdaycı, Earl T Campbell, Neil I Gillespie, Kauser Johar, Ram Rajan, Adam W Richardson, Luka Skoric, Canberk Topal, Mark L Turner, and Abbas B Ziad. 2023. A real-time, scalable, fast and highly resource efficient decoder for a quantum computer. *arXiv preprint arXiv:2309.05558* (2023).

[7] Johannes Bausch, Andrew W Senior, Francisco JH Heras, Thomas Edlich, Alex Davies, Michael Newman, Cody Jones, Kevin Satzinger, Murphy Yuezhen Niu, Sam Blackwell, George Holland, Dvir Kafri, Juan Atalaya, Craig Gidney, Demis Hassabis, Sergio Boixo, Hartmut Neve, and Pushmeet Kohli. 2023. Learning to decode the surface code with a recurrent, transformer-based neural network. *arXiv preprint arXiv:2310.05900* (2023).

[8] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. 2022. Surface code compilation via edge-disjoint paths. *PRX Quantum* 3, 2 (2022), 020342.

[9] Michael E Beverland, Prakash Murali, Matthias Troyer, Krysta M Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. 2022. Assessing requirements to scale to practical quantum advantage. *arXiv preprint arXiv:2211.07629* (2022).

[10] Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A—Atomic, Molecular, and Optical Physics* 71, 2 (2005), 022316.

[11] Ilkwon Byun, Junpyo Kim, Dongmoon Min, Ikki Nagaoka, Kosuke Fukumitsu, Iori Ishikawa, Teruo Tanimoto, Masamitsu Tanaka, Koji Inoue, and Jangwoo Kim.

2022. XQsim: modeling cross-technology control processors for 10+ K qubit quantum computers. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 366–382.

[12] Christopher Chamberland, Pavithran Iyer, and David Poulin. 2018. Fault-tolerant quantum computing in the Pauli or Clifford frame with slow error diagnostics. *Quantum* 2 (2018), 43.

[13] Christopher T Chubb and Steven T Flammia. 2021. Statistical mechanical models for quantum codes with correlated noise. *Annales de l'Institut Henri Poincaré D* 8, 2 (2021), 269–321.

[14] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, and Blake R Johnson. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–50.

[15] Poulami Das, Aditya Locharla, and Cody Jones. 2022. Lilliput: a lightweight low-latency lookup-table decoder for near-term quantum error correction. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 541–553.

[16] Poulami Das, Christopher A Pattison, Srilatha Manne, Douglas Carmean, Krysta Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2020. A scalable decoder micro-architecture for fault-tolerant quantum computing. *arXiv preprint arXiv:2001.06598* (2020).

[17] Poulami Das, Christopher A Pattison, Srilatha Manne, Douglas M Carmean, Krysta M Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2022. Afs: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 259–273.

[18] Nicolas Delfosse and Naomi H Nickerson. 2021. Almost-linear time decoding algorithm for topological codes. *Quantum* 5 (2021), 595.

[19] Nicolas Delfosse and Gilles Zémor. 2020. Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel. *Physical Review Research* 2, 3 (2020), 033042.

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

[21] Cirq Developers. 2024. *Cirq*. https://doi.org/10.5281/zenodo.11398048

[22] Yongshan Ding and Frederic T Chong. 2020. Quantum computer systems: Research for noisy intermediate-scale quantum computers. *Synthesis lectures on computer architecture* 15, 2 (2020), 1–227.

[23] David P DiVincenzo and Panos Aliferis. 2007. Effective fault-tolerant quantum computation with slow measurements. *Physical review letters* 98, 2 (2007), 020501.

[24] Austin G Fowler and Craig Gidney. 2018. Low overhead quantum computation using lattice surgery. *arXiv preprint arXiv:1808.06709* (2018).

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

[26] Austin G Fowler, Adam C Whiteside, and Lloyd CL Hollenberg. 2012. Towards practical classical processing for the surface code. *Physical review letters* 108, 18 (2012), 180501.

[27] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.

[28] Craig Gidney. 2024. Inplace access to the surface code y basis. *Quantum* 8 (2024), 1310.

[29] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433.

[30] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction*. California Institute of Technology.

[31] Matthew P Harrigan, Tanuj Khattar, Charles Yuan, Anurudh Peduri, Noureldin Yosri, Fionn D Malone, Ryan Babbush, and Nicholas C Rubin. 2024. Expressing and Analyzing Quantum Algorithms with Qualtran. *arXiv preprint arXiv:2409.04643* (2024).

[32] Yoshihito Hashimoto, Shinichi Yorozu, Toshiyuki Miyazaki, Yoshio Kameda, Hideo Suzuki, and Nobuyuki Yoshikawa. 2007. Implementation and experimental evaluation of a cryocooled system prototype for high-throughput SFQ digital applications. *IEEE transactions on applied superconductivity* 17, 2 (2007), 546–551.

[33] Oscar Higgott. 2022. Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–16.

[34] Oscar Higgott and Craig Gidney. 2023. Sparse blossom: correcting a million errors per core second with minimum-weight matching. *arXiv preprint arXiv:2303.15933* (2023).

[35] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshan Ding, Massoud Pedram, and Frederic T. Chong. 2020. NISQ+: Boosting quantum computing power by approximating quantum error correction. arXiv:2004.04794 [quant-ph] https://arxiv.org/abs/2004.04794

[36] Dominic Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. 2012. Surface code quantum computing by lattice surgery. *New Journal of Physics* 14, 12 (2012), 123011.

[37] Emanuel Knill. 2007. Quantum computing with very noisy devices. *arXiv preprint quant-ph/0410199* (2007).

[38] Sebastian Krinner, Simon Storz, Philipp Kurpiers, Paul Magnard, Johannes Heinsoo, Raphael Keller, Janis Luetolf, Christopher Eichler, and Andreas Wallraff. 2019. Engineering cryogenic setups for 100-qubit scale superconducting circuit systems. *EPJ Quantum Technology* 6, 1 (2019), 2.

[39] Tyler LeBlond, Christopher Dean, George Watkins, and Ryan Bennink. 2024. Realistic Cost to Execute Practical Quantum Circuits using Direct Clifford+ T Lattice Surgery Compilation. *ACM Transactions on Quantum Computing* 5, 4 (2024), 1–28.

[40] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing* 4, 2 (2023), 1–26.

[41] Sophia Fuhui Lin, Eric C Peterson, Krishanu Sankar, and Prasahnt Sivarajah. 2025. Spatially parallel decoding for multi-qubit lattice surgery. *Quantum Science and Technology* 10, 3 (2025), 035007.

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

[43] Namitha Liyanage, Yue Wu, Alexander Deters, and Lin Zhong. 2023. Scalable quantum error correction for surface codes using FPGA. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 916–927.

[44] Satvik Maurya and Swamit Tannu. 2024. Managing Classical Processing Requirements for Quantum Error Correction. *arXiv preprint arXiv:2406.17995* (2024).

[45] Michael A Nielsen and Isaac L Chuang. 2001. Quantum computation and quantum information. *Phys. Today* 54, 2 (2001), 60.

[46] Pavel Panteleev and Gleb Kalachev. 2021. Degenerate quantum LDPC codes with good finite length performance. *Quantum* 5 (2021), 585.

[47] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* (2023). MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[48] Gokul Subramanian Ravi, Jonathan M Baker, Arash Fayyazi, Sophia Fuhui Lin, Ali Javadi-Abhari, Massoud Pedram, and Frederic T Chong. 2023. Better than worst-case decoding for quantum error correction. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 88–102.

[49] Ben W Reichardt. 2004. Improved magic states distillation for quantum universality. *arXiv preprint quant-ph/0411036* (2004).

[50] Eleanor G Rieffel and Wolfgang H Polak. 2011. *Quantum computing: A gentle introduction*. MIT Press.

[51] Leon Riesebos, Xiang Fu, Savvas Varsamopoulos, Carmen G Almudever, and Koen Bertels. 2017. Pauli frames for quantum computer architectures. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[52] Neil J Ross and Peter Selinger. 2016. Optimal ancilla-free Clifford+ T approximation of z-rotations. *Quantum Inf. Comput.* 16, 11&12 (2016), 901–953.

[53] rroodll, jbelarge, elenewski, and zmorrell. 2024. *isi-usc-edu/pyLIQTR: Release 1.1.1*. https://doi.org/10.5281/zenodo.10913397

[54] Noah Shutty, Michael Newman, and Benjamin Villalonga. 2024. Efficient near-optimal decoding of the surface code through ensembling. *arXiv preprint arXiv:2401.12434* (2024).

[55] Luka Skoric, Dan E Browne, Kenton M Barnes, Neil I Gillespie, and Earl T Campbell. 2023. Parallel window decoding enables scalable fault tolerant quantum computation. *Nature Communications* 14, 1 (2023), 7040.

[56] Xinyu Tan, Fang Zhang, Rui Chao, Yaoyun Shi, and Jianxin Chen. 2023. Scalable surface-code decoders with parallelization in time. *PRX Quantum* 4, 4 (2023), 040344.

[57] Swamit S Tannu, Zachary A Myers, Prashant J Nair, Douglas M Carmean, and Moinuddin K Qureshi. 2017. Taming the instruction bandwidth of quantum computers via hardware-managed error correction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 679–691.

[58] Barbara M Terhal. 2015. Quantum error correction for quantum memories. *Reviews of Modern Physics* 87, 2 (2015), 307–346.

[59] Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki, and Yutaka Tabuchi. 2021. QECOOL: On-line quantum error correction with a superconducting decoder for surface code. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 451–456.

[60] Yosuke Ueno, Yuna Tomida, Teruo Tanimoto, Masamitsu Tanaka, Yutaka Tabuchi, Koji Inoue, and Hiroshi Nakamura. 2023. Inter-Temperature Bandwidth Reduction in Cryogenic QAOA Machines. *IEEE Computer Architecture Letters* (2023).

[61] Savvas Varsamopoulos, Koen Bertels, and Carmen Garcia Almudever. 2019. Comparing neural network based decoders for the surface code. *IEEE Trans. Comput.* 69, 2 (2019), 300–311.

[62] Suhas Vittal, Poulami Das, and Moinuddin Qureshi. 2023. Astrea: Accurate quantum error-decoding via practical minimum-weight perfect-matching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–16.

[63] Yue Wu, Namitha Liyanage, and Lin Zhong. 2025. Micro Blossom: Accelerated Minimum-Weight Perfect Matching Decoding for Quantum Error Correction. *arXiv preprint arXiv:2502.14787* (2025).

[64] Yue Wu and Lin Zhong. 2023. Fusion blossom: Fast mwpm decoders for qec. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 928–938.
[65] Amir Youssefi, Itay Shomroni, Yash J Joshi, Nathan R Bernier, Anton Lukashchuk, Philipp Uhrich, Liu Qiu, and Tobias J Kippenberg. 2021. A cryogenic electro-optic interconnect for superconducting devices. *Nature Electronics* 4, 5 (2021), 326–332.
[66] Jiaxuan Zhang, Zhao-Yun Chen, Jia-Ning Li, Tian-Hao Wei, Huan-Yu Liu, Xi-Ning Zhuang, Qing-Song Li, Yu-Chun Wu, and Guo-Ping Guo. 2024. Integrating Window-Based Correlated Decoding with Constant-Time Logical Gates for Large-Scale Quantum Computation. *arXiv preprint arXiv:2410.16963* (2024).

A Artifact Appendix

A.1 Abstract

The artifact includes the source code, data, and scripts necessary to reproduce the results in this paper. Additionally, it contains our introduced tool, SWIPER-SIM, a simulator for window decoding of general lattice surgery programs.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Python3.10+, slurm.
- **Hardware:** HPC cluster for benchmark evaluations; PC for other experiments.
- **Metrics:** Prediction accuracy, T gate reaction time, program runtime.
- **Output:** SWIPER-SIM lattice surgery program traces.
- **Experiments:** Sampling decoder latencies, evaluating predictor accuracy, evaluating misprediction recovery strategies, evaluating T gate reaction time, evaluating benchmark program runtime and classical decoding cost.
- **How much disk space required (approximately)?:** 10GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** Varies based on experiment. Largest benchmark simulations take several days to complete on an HPC cluster, but all other data generation can be run in under one hour locally.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Archived (provide DOI)?:** doi: 10.5281/zenodo.15102954

A.3 Description

A.3.1 How to access. The most up-to-date version of the code can be accessed by cloning the repository from Github, or the archived artifact version can be accessed by downloading and unzipping the Zenodo archive.

A.3.2 Hardware dependencies. A modern PC for running most experiments, and an HPC cluster with SLURM for running all SWIPER-SIM benchmark evaluations.

A.3.3 Software dependencies.

(1) `git`
(2) `Python` 3.10 or 3.11.
(3) `CMake` and `gcc` (verified to work with versions 3.31.6 and 9.2.0, respectively)

If you run into issues when building and installing the `tweedledum` package, note that this depends on some C++ libraries that may need to be manually reinstalled if the version on your PC is too old.

A.4 Installation

From the repository root, use `pip` to install dependencies:

```
pip install -r requirements.txt
```

A.5 Experiment workflow

The data generated for this publication is provided in 'artifact/data/'. Each of the results can be reproduced using the following Python scripts (note that two require a SLURM cluster). The final script, `artifact/run_analysis.py`, generates the figures based on the (original or new) data in 'artifact/data/'.

- `artifact/run_pymatching_latencies.py`: Evaluates latency of Pymatching on random decoding problems for different code distances and decoding volumes. Relevant for Fig. 3, and as an input to SWIPER-SIM. Expected runtime: about 1 hour.
- `artifact/run_predictor_accuracy.py`: Simulates 1-, 2-, and 3-step predictor accuracy for different code distances. Relevant for Fig. 4. Expected runtime: about 1 hour.
- `artifact/run_mispredict_strategy.py`: Simulates different strategies for recovering from mispredictions. Relevant for Fig. 8. Expected runtime: about 5 minutes.
- `artifact/run_reaction_time_evals.py`: Runs 300 SWIPER-SIM slurm jobs to evaluate SWIPER on a simple "random-T" schedule with different decoder latencies. Relevant for Fig. 12. Expected runtime: one minute to submit jobs, and several hours for SLURM jobs to complete (assuming sufficient parallelization).
- `artifact/run_benchmark_evals.py`: Runs 3000 SWIPER-SIM slurm jobs to evaluate SWIPER on various application benchmarks. Relevant for Figs. 14 and 15. Expected runtime: one minute to several hours to submit jobs (depending on configured delay between submitting sets of jobs; this can be configured), and up to several days for all SLURM jobs to complete.
- `artifact/run_analysis.py`: **Generates all data-related figures that appear in the paper.** This involves reading pre-generated data from 'artifact/data', as well as performing some simpler simulations/calculations within the script. Expected runtime: 5 minutes.

A.6 Evaluation and expected results

Figures 3, 4, 7, 8b, 9, 10, 11a-b, 12a-c, 14a-c, and 15a-b can be reproduced using this code. Additionally, the minimum, geomean, and maximum speedups achieved with SWIPER on the benchmark programs can be calculated. The results from the various data generation scripts are stored in 'artifact/data/'. The generated figures are stored in 'artifact/figures/', and are named according to their number in the paper.

A.7 Experiment customization

The experiments are fully configurable by directly editing the Python scripts. In particular, the SLURM submission scripts can be edited to include other benchmarks, to evaluate at different code distances, to use different speculation accuracies or decoder latency distributions, etc. Note that the compiled benchmark programs and

decoder latency distribution are loaded from files, so those files must first be updated (using other scripts in the 'artifact/' directory) to customize the experiment.