

Predictive Window Decoding for Fault-Tolerant Quantum Programs

Joshua Vizlai*[†]
University of Chicago
Chicago, Illinois, USA

Jason D. Chadwick*
University of Chicago
Chicago, Illinois, USA

Sarang Joshi
University of Chicago
Chicago, Illinois, USA

Gokul Subramanian Ravi
University of Michigan
Ann Arbor, Michigan, USA

Yanjing Li
University of Chicago
Chicago, Illinois, USA

Frederic T. Chong
University of Chicago
Chicago, Illinois, USA

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 schemes can scalably meet throughput requirements in the presence of increasing decoding times, given enough classical resources. However, windowed decoding schemes 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 a *speculative* window decoding scheme. Taking inspiration from branch prediction in classical computer architecture our decoder 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 simulator, we find that speculation reduces application runtimes by 40% on average compared to prior parallel window decoders.

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 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. [19], known as sliding window decoding, data is passed forward in time, creating sequentially-dependent decoding problems.

While a decoding system is necessary for enabling QEC, its efficiency also has strong implications for 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 [53] in the sliding window setting due to an accumulating backlog of decoding tasks that all rely on

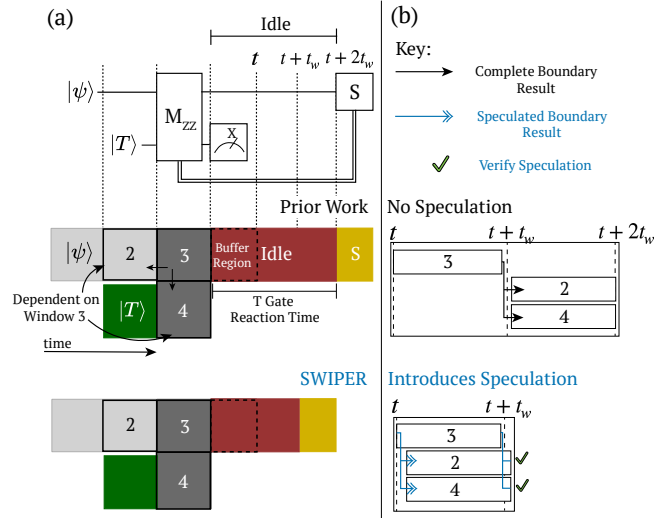


Figure 1: (a) Decoding a lattice surgery T gate teleportation (top left) using prior parallel window decoders (middle) and using SWIPER (bottom). (b) Resulting decoding pipelines. SWIPER improves decoding throughput with a lightweight speculation step, allowing dependent windows to begin decoding earlier.

their predecessors. In response, research has broadly improved decoder performance through a combination of algorithmic innovations [17, 32, 56] and tailored optimizations [3, 44, 55]. When addressing the backlog problem, however, 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 they will, in turn, increase throughput. However, as discussed both by Skoric et al. [51] and Tan et al. [52] the decoding backlog is chiefly a problem of throughput. 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

*Both authors contributed equally to this research.

[†]Correspondance: vizlai@uchicago.edu

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 sub-optimal. 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, removing 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 only exist 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 a window’s total syndrome data. ② Error chains crossing the boundaries between windows will be short and sparsely distributed in an 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 SWIPER provides a 40% reduction in fault-tolerant program runtime compared to prior 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, 28]. Our results utilize our decoding simulator, SWIPER-SIM, that 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 prior parallel window decoders.
- We develop **SWIPER-SIM, a round-level lattice surgery decoding simulator**, enabling program-level simulations of parallel window decoding and allowing 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 relaxes decoder latency requirements consistently by over 2 – 5×**, enabling the development of more powerful and accurate decoders.
- We evaluate SWIPER with realistic decoder parameters on an assortment 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 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.

2 Background

For in-depth background we refer to [21, 41, 46] for quantum computing fundamentals and [24, 29] for QEC and stabilizer codes. In this section, we give 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 in 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 [33], in which adjacent surface code patches are *merged* and *split* to perform logical operations. These primitives enable universal computation [23, 38] which has led to a growing set of resource estimates for large quantum programs [9, 28].

2.2 Quantum Error Correction Decoding

We presume 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 [17, 25, 31]. 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 non-zero 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.

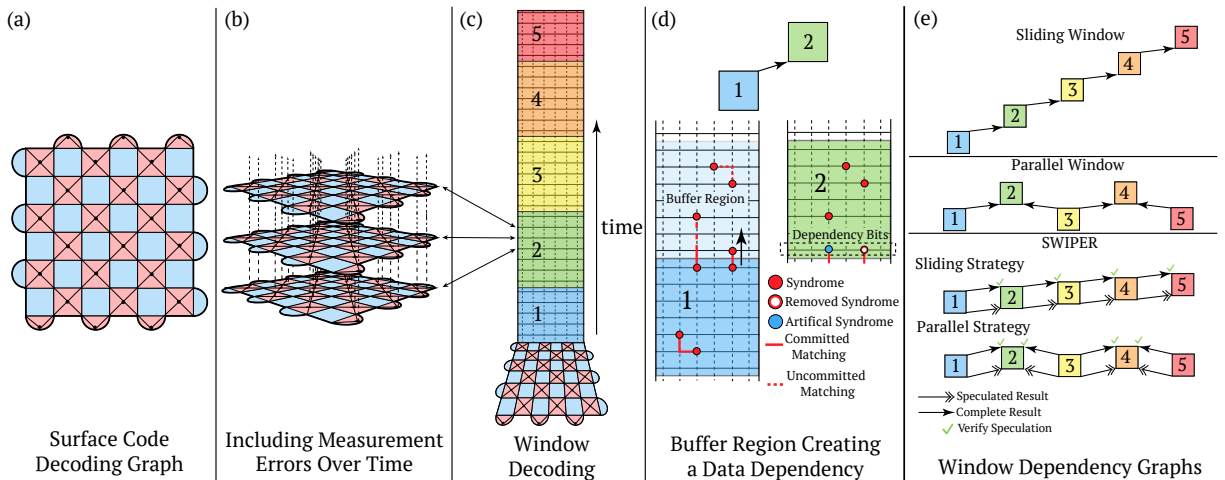


Figure 2: (a) A $d = 7$ surface code patch with $Z(X)$ stabilizers indicated by red(blue) faces. Drawn is the decoding graph for Z stabilizers, with vertices of the graph indicating stabilizers and edges indicating error-prone data qubits. (b) Repeated rounds of measuring stabilizers. The decoding graph 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).

2.3 Blocking Operations

Decoding results can be tracked in software for some time using Pauli frames [34, 47], 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 in order to continue past 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 [22, 53]. 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 [19], 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 contain 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 [37]. 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 “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 the overall decoding performance.

2.5 Sliding Window Approach

The overlapping recovery method [19] 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 a sliding window decoder uses a similar, feed-forward approach for windows sliding along a spatial dimension.

2.6 Parallel Window Approach

In parallel window decoding [37, 51, 52], 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 of 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 [37].

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 to generate a decoding window. Regimes exist where this is avoided, particularly smaller code distances with hardware-based implementations [6, 15, 39], however, in all of these the latency scales with the code distance d with the highest distance achieved being $d = 23$, still below distances expected for large-scale applications like factoring [28]. In Figure 3 we highlight this point by

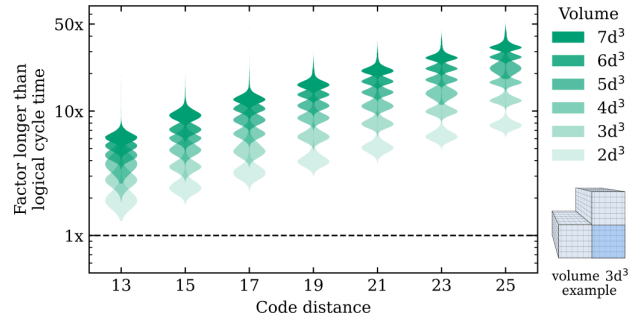


Figure 3: PyMatching decoding latency distributions for varying code distances and window sizes. Latencies are relative to a logical cycle consisting of d measurement rounds where each measurement round is assumed to take $1 \mu\text{s}$. Lower right: example of a $3d^3$ decoding volume in the style of Figure 2c.

plotting decoding latencies using PyMatching, a state-of-the-art software decoder [31], and Stim [26], a detailed surface code simulator, with an assumed syndrome measurement round time of $1 \mu\text{s}$ 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 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, 50] and decoders for qLDPC codes [42] 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 a quantum program running on a surface-code-based system will be decomposed into Clifford+T gates, where 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 [38]. 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.

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 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 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.

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 we use and present benchmark evaluations comparing SWIPER to prior, 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 complete, we then verify the speculations were correct. The resulting reaction time is reduced, allowing the program to continue past the blocking operation earlier than with prior work. Importantly, if we find that any speculations were incorrect upon completing 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, 44, 55]. 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,

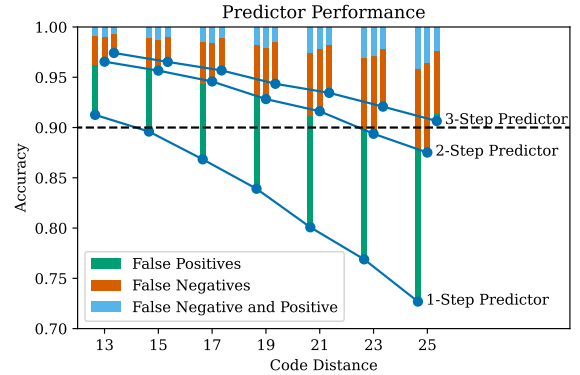


Figure 4: Accuracy of the three different predictors for increasing values of code distance. Bars indicate a breakdown of different failure cases.

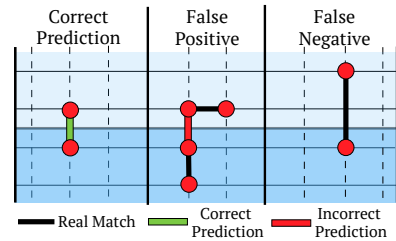


Figure 5: Common cases for the 1-step predictor.

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 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 crossing the boundary was predicted but did not actually exist and a false negative is when a matching that did cross the boundary was not predicted. We generate circuit-level surface code windows at a physical qubit error rate of 0.1% using Stim [26] and we use PyMatching [32] as the reference decoder to determine misprediction. Despite its simplicity, the 1-step predictor can still reach $> 70\%$ accuracy for all code distances we consider.

The 1-step predictor can be further improved by studying common failure cases. In Figure 5 we describe the most common cases the 1-step predictor encounters. While 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 from 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

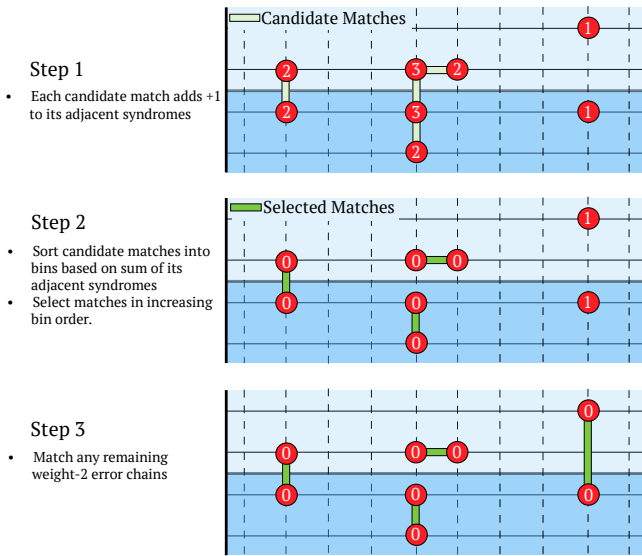


Figure 6: An overview of the final, 3-step predictor logic.

the peeling decoder introduced in [18] where we prioritize matching syndrome bits with the fewest viable matches, akin to a leaf node.

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 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 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 the 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. Similar 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 $O(d^2)$. Since this occurs after the 2-step predictor, we expect all simple, weight 1 error chains to already be matched. We summarize the logic of this final, 3-step predictor in Figure 6.

In Figure 4 we can see the 3-step predictor reduces the amount of false negatives mispredicted by the 1 and 2-step predictors. As a result, we reach a prediction accuracy of $> 90\%$ for all code distances we simulate.

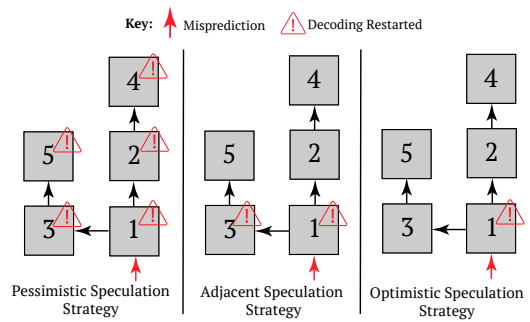


Figure 7: Three different speculation strategies for handling a misprediction which has poisoned window 1. “Pessimistic” restarts all descendants of a poisoned window. “Adjacent” restarts any window that used a prediction on an adjacent boundary to the misprediction. “Optimistic” restarts only the poisoned window itself.

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 [6, 39]. We performed a behavioral simulation of the 3-step predictor on 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 to the full decoding volume of $O(d^3)$, we also expect area costs to scale favorably compared to the full decoder. This separation increases as d increases. For example, at the smallest window volume of $2d^3$, with $d = 13$ the predictor examines $\sim 15\%$ of the total syndrome data whereas with $d = 25$ the predictor only examines $\sim 8\%$ of the total syndrome data.

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 exhibits an increased demand on the number of concurrent decoders, as SWIPER collapses the dependency structure of windows. 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. As such, we believe an increase in classical compute to reduce the required amount of quantum compute is a beneficial trade. However, here we study how to minimize the required classical costs of SWIPER. Particularly, an incorrect speculation can cause wasted classical computation as it requires restarting some decoding tasks that were based on flawed dependency bits. SWIPER mitigates this cost through an *optimistic speculation strategy*.

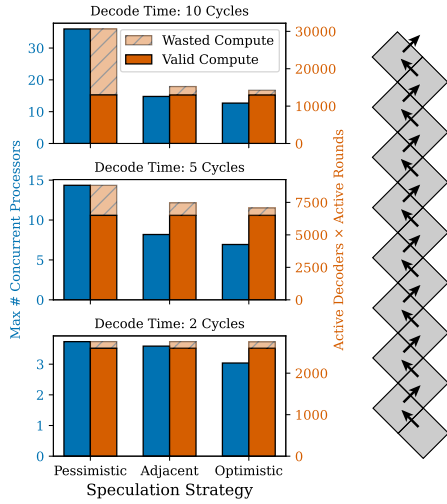


Figure 8: Estimating classical decoder costs of the three different proposed speculation strategies. All values are averaged over 10,000 shots. The specific case simulated is 100 windows in a “zig-zag” shape (right) using a sliding schedule with speculation. In the breakdown, “valid compute” indicates compute that finished and was correct, whereas “wasted compute” indicates compute that was restarted early or found to be incorrect.

4.3.1 Handling Mispredictions. A waiting decoding window begins decoding after its incoming dependencies have passed boundary speculations to it. Once each dependency is fully decoded, however, 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 the predictor was incorrect, we have a *misprediction*. A misprediction means that 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, the effect of a misprediction does not necessarily stop here. 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 an 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 7, the *pessimistic speculation strategy* (left) restarts all descendants that already began 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. On the other hand, 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. An *optimistic speculation strategy* (Figure 7, right) therefore 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 7 (center).

In Figure 8, we show classical costs for decoding a sequence of 100 windows with speculation failures using the three speculation strategies detailed above. To study a regime where we expect the most variation between the three strategies, we use a sliding window schedule with a “zig-zag” dependency structure such that successive boundaries are always adjacent. 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 strategy unnecessarily restarts more windows, wasting classical compute. In all cases, we find the optimistic strategy performs the best, which we attribute to the high prediction accuracy and minimal downstream effects in the case of mispredictions.

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 9 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 [30, 36, 43, 49] which can be compiled down to Clifford+RZ gates in OpenQASM [14] using Cirq [20]. We then use Gridsynth [48] 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 [35] to create a mapped and routed 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 9). Based on the lattice surgery program, the DeviceManager generates a set of syndrome rounds every $1 \mu\text{s}$ (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

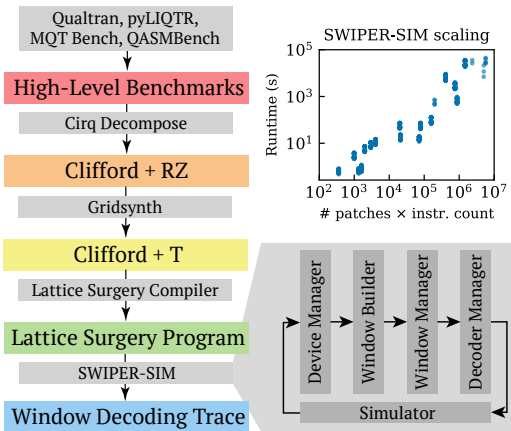


Figure 9: The pipeline used to evaluate high-level benchmark programs with different window decoders. *Right*: details of SWIPER-SIM and its runtime versus program size on a single core of an Intel Xeon Gold 6248R processor.

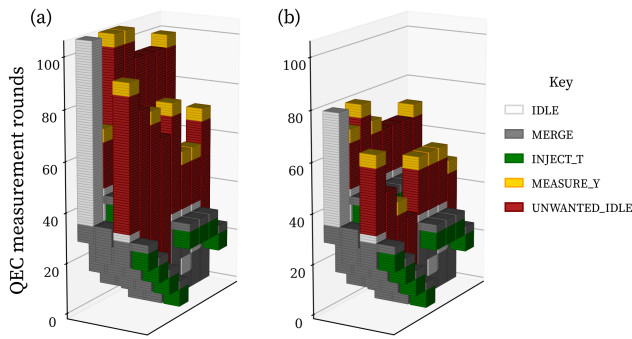


Figure 10: SWIPER-SIM program traces for a 15-to-1 magic state distillation in a distance-7 code using the construction from [23] (Fig. 17). Time advances vertically, and each horizontal slice represents a batch of syndrome data (colored by instruction type). Decoding time is fixed to be the same as 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).

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,

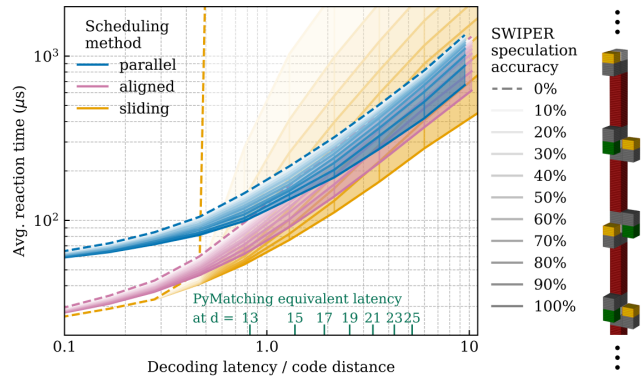


Figure 11: Sensitivity of reaction time to decoding latency and speculation accuracy. Speculation accuracy of 0% corresponds to baseline (no speculation) decoding. Decoder latency is $t_{\text{dec}}(v) = rv/d^2$ rounds, where v is the volume of the decoding problem (in units of d^3) and the relative latency factor r is varied along the x axis. Equivalent latency factors extracted from linear fits to PyMatching latency data (Figure 3) are shown along the x axis. *Right*: simulator trace of repeated T gates on an idling logical qubit. A similar experiment with 1000 T gates was used to collect the data in this figure.

even if the reaction time is not a multiple of the window size. As an example, Figure 10 shows a spacetime program trace generated by SWIPER-SIM for a hand-specified lattice surgery program for 15-to-1 magic state distillation [10, 23, 45]. We can compare prior work (Figure 10a) with SWIPER (Figure 10b) 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 [27].

5.2 Studying reaction times with SWIPER

In Figure 11 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 $r > 0.5$, where the reaction time begins to grow exponentially with each successive blocking operation. SWIPER mitigates this problem with speculation, but we find 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.

5.2.1 T Gate Alignment. We also see that at small decoding latencies the sliding window strategy outperforms the parallel window strategy by up to 50%. We can explain this in part by noticing 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 12, 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.

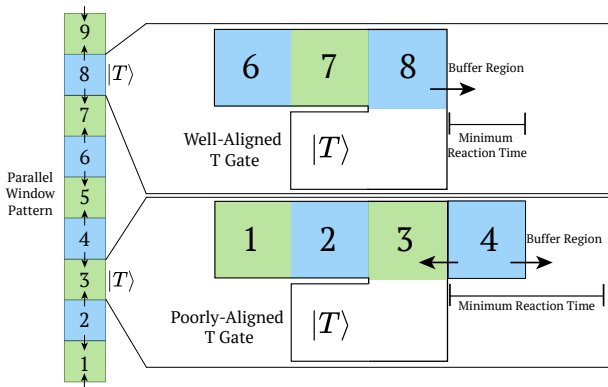


Figure 12: Comparing a well-aligned T gate (top) and a poorly-aligned T gate (bottom).

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 a minimal reaction time. Since a sliding window schedule always ends with source boundaries, it is always aligned. However the 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 11, we find 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 to 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. We can also analyze the data from Figure 11 from a different angle: in a setting with a fixed runtime budget, SWIPER allows for significantly longer inner decoder

latencies. For fixed reaction time values, we compare high-accuracy SWIPER to the default parallel window scheme and find that speculated aligned and sliding schemes allow upwards of $5\times$ increased latency for reaction times near $100\ \mu\text{s}$ and all three (parallel, aligned, and sliding) speculated methods allow over $2\times$ increased latency in the limit of large reaction time ($500\ \mu\text{s}+$). Again, we can understand this limit by remembering that SWIPER collapses the two-layer dependency structure of parallel windows.

This relaxation in decoder latency could be translated to an increase in program fidelity by using recurrent, transformer-based decoders [7], tensor network decoders [1, 13], or ensemble decoders [50] 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 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 logical error rates below 10^{-12} . This is often referred to as the “teraquop regime” where a trillion logical operations can be executed. We again assume that each QEC syndrome round takes $1\ \mu\text{s}$ and we sample decoding latencies from the distributions shown in Figure 3, rounding up if the window volume is not an integer multiple of d^3 . Speculation is assumed to take $1\ \mu\text{s}$ 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” (μ Benchmarks) 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, simulating 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 13 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 more than 1000 T gates: without SWIPER, using the aligned scheduling method achieves 4.3% to 8.5% (geomean 5.8%) reduction in runtime compared to parallel window. With SWIPER, all three scheduling methods (parallel, aligned, and

Benchmark	Short Name	Space footprint	Compiled # T's	Source	Domain
Toffoli	toffoli	9	7	[41]	μ Benchmark
Magic State Distillation	msd_15to1	32	15	[23]	Resources, μ Benchmark
RZ(ϕ), $\epsilon = 10^{-10}$	rz	4	100	[48]	μ Benchmark
Quantum Read Only Memory	qrom	153	48	[30]	Data I/O
8-bit Adder	adder_8bit	111	112	[36]	Factoring subroutine
Carleman Encoding	carleman	264	394	[49]	ODE
H ₂ Qubitized Walk Operator	H2_molecule	92	3084	[49]	Chemistry
Fermi Hubbard 4 × 4 Lattice	fermi_hubbard	225	3898	[49]	Chemistry
Quantum Phase Estimation	qpe	85	11378	[43]	Common subroutine
Quantum Fourier Transform	qft	86	11484	[43]	Factoring subroutine
Grover's	grover	34	13577	[43]	Data Search

Table 1: Selected benchmark applications. Space footprint is the number of $d \times d$ surface code patches used.

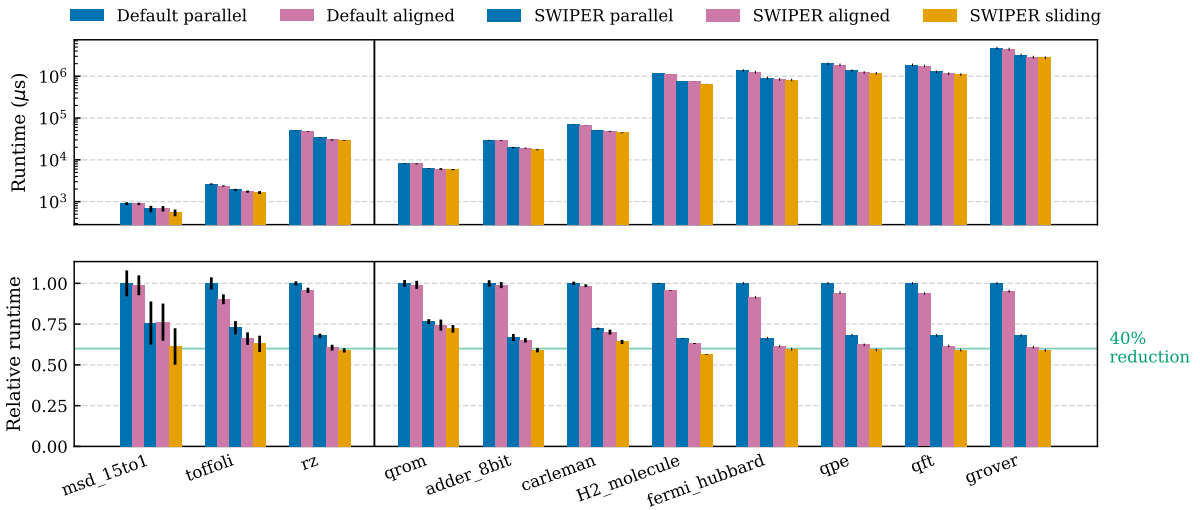


Figure 13: Top: Program runtime for selected benchmarks under different window scheduling methods, with and without SWIPER. Classical processor resources are unlimited. Vertical black lines indicate standard deviation over 10 randomized trials for smaller benchmarks. Bottom: Relative runtimes normalized to default parallel window runtime.

sliding) improve significantly. SWIPER-parallel achieves 31.8% to 33.9% (geomean 32.7%) reduction in runtimes, SWIPER-aligned achieves 36.9% to 39.2% (geomean 38.1%) reduction in runtimes, and SWIPER-sliding achieves 40.4% to 43.6% (geomean 41.4%) 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 ($\sim 40\%$ compared to $\sim 70\text{-}80\%$ for the other benchmarks), so the benefit of reducing reaction time is slightly less impactful.

5.3.3 Runtime uncertainty and extrapolation. It's 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 13 have error bars showing the standard deviation of results. In Figure 14, we show that this standard deviation (relative to mean) decreases as 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 occurs with 50% probability whereas mispredictions occur only with 10% probability.

Looking forward, because SWIPER's main benefit is reducing the reaction time for T gates, which are typically the main cost of a program [9, 38], the magnitude of runtime improvement is broadly

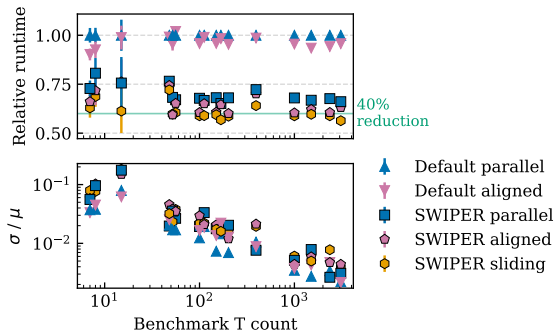


Figure 14: Top: Performance improvement of SWIPER appears to be consistent across benchmarks size. Bottom: Uncertainty in runtime, showing that relative uncertainty decreases for larger programs.

independent of program size, so we expect we would see similar improvements for full application-level benchmarks.

5.3.4 Limiting classical processors. 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 an 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 15 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 31% more 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.

6 Related Work

While QEC decoding has broadly been an active area of research [11, 16, 40, 54], related work on decoder performance has largely focused on achieving a decoding latency within $1 \mu\text{s}$ as a requirement for real-time decoding, motivated by the fact that superconducting systems generate a round of syndrome measurements every $\sim 1 \mu\text{s}$. LILLIPUT [15] proposes a look-up table that can decode up to $d = 5$ within $1 \mu\text{s}$. Astrea [55] extends this to $d = 9$ in $1 \mu\text{s}$ via

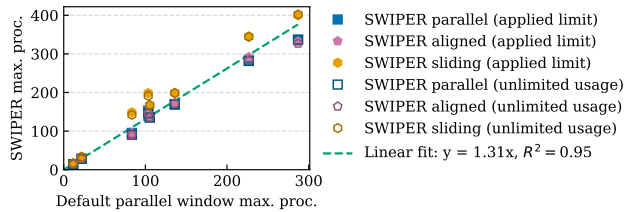


Figure 15: Comparing classical compute cost between baseline method and SWIPER.

brute-force searching low-Hamming-weight bitstrings. Clique [44] is a lightweight pre-decoder that specifically tackles isolated errors and bears some resemblance to the 1-Step Predictor proposed in this work. More recently, Promatch [3] proposes an adaptive pre-decoding step to lift this limit to $d = 13$ within $1 \mu\text{s}$ 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 $1 \mu\text{s}$. Helios [39] demonstrates an impressive implementation of the Union-Find decoder [17] 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. While 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 $1 \mu\text{s}$ as a hard requirement for decoding latency. By relaxing the constraints on the inner decoder, this also enables decoders which historically have had prohibitive latencies, such as higher accuracy decoders [7, 50] and decoders for qLDPC codes [42].

Prior work specifically addressing parallel window decoders is still limited. The original proposals for parallel window decoding [51, 52], work analyzing their performance for spatial windows during lattice surgery [37], and work extending this to transversal gate computation for high-connectivity systems [57] all assume 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 adjacent 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 [32, 56] 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 [51]. 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.

Acknowledgements

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 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.

Data Availability

This manuscript is currently under peer review. We will fully open-source the code and supporting data when it is published.

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 Heide, 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 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.
- [17] Nicolas Delfosse and Naomi H Nickerson. 2021. Almost-linear time decoding algorithm for topological codes. *Quantum* 5 (2021), 595.
- [18] 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.
- [19] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (2002), 4452–4505.
- [20] Cirq Developers. 2024. *Cirq*. <https://doi.org/10.5281/zenodo.11398048>
- [21] 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.
- [22] David P DiVincenzo and Panos Aliferis. 2007. Effective fault-tolerant quantum computation with slow measurements. *Physical review letters* 98, 2 (2007), 020501.
- [23] Austin G Fowler and Craig Gidney. 2018. Low overhead quantum computation using lattice surgery. *arXiv preprint arXiv:1808.06709* (2018).
- [24] 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.
- [25] 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.
- [26] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.
- [27] Craig Gidney. 2024. Inplace access to the surface code y basis. *Quantum* 8 (2024), 1310.
- [28] 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.
- [29] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction*. California Institute of Technology.
- [30] 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).
- [31] 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.
- [32] 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).
- [33] 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.
- [34] Emanuel Knill. 2007. Quantum computing with very noisy devices. *arXiv preprint quant-ph/0410199* (2007).

- [35] 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.
- [36] 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.
- [37] Sophia Fuhui Lin, Eric C Peterson, Krishanu Sankar, and Prasahnt Sivarajah. 2024. Spatially parallel decoding for multi-qubit lattice surgery. *arXiv preprint arXiv:2403.01353* (2024).
- [38] Daniel Litinski. 2019. A game of surface codes: Large-scale quantum computing with lattice surgery. *Quantum* 3 (2019), 128.
- [39] 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.
- [40] Satvik Maurya and Swamit Tannu. 2024. Managing Classical Processing Requirements for Quantum Error Correction. *arXiv preprint arXiv:2406.17995* (2024).
- [41] Michael A Nielsen and Isaac L Chuang. 2001. Quantum computation and quantum information. *Phys. Today* 54, 2 (2001), 60.
- [42] Pavel Panteleev and Gleb Kalachev. 2021. Degenerate quantum LDPC codes with good finite length performance. *Quantum* 5 (2021), 585.
- [43] 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/>.
- [44] 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.
- [45] Ben W Reichardt. 2004. Improved magic states distillation for quantum universality. *arXiv preprint quant-ph/0411036* (2004).
- [46] Eleanor G Rieffel and Wolfgang H Polak. 2011. *Quantum computing: A gentle introduction*. MIT Press.
- [47] Leon Rieseboos, 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.
- [48] 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.
- [49] rroodll, jbelarge, elenewski, and zmorrell. 2024. *isi-usc-edu/pyLIQTR: Release 1.1.1*. <https://doi.org/10.5281/zenodo.10913397>
- [50] Noah Shutty, Michael Newman, and Benjamin Villalonga. 2024. Efficient near-optimal decoding of the surface code through ensembling. *arXiv preprint arXiv:2401.12434* (2024).
- [51] 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.
- [52] 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.
- [53] Barbara M Terhal. 2015. Quantum error correction for quantum memories. *Reviews of Modern Physics* 87, 2 (2015), 307–346.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] Jiakuan 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).