A Case for Elastic Quantum Error Correction Decoders

Satvik Maurya
University of Wisconsin-Madison
Madison, WI, USA

Abtin Molavi
University of Wisconsin-Madison
Madison, WI, USA

Aws Albarghouthi
University of Wisconsin-Madison
Madison, WI, USA

Swamit Tannu
University of Wisconsin-Madison
Madison, WI, USA

Abstract

Large-scale quantum computers promise transformative speedups, but their viability hinges on fast and reliable quantum error correction (QEC). At the center of QEC are *decoders*—classical algorithms running on hardware such as FPGAs, GPUs, or CPUs that process error syndromes to detect errors every microsecond to preserve fault-tolerance. Quantum processors, therefore, operate not in isolation, but as accelerators tightly coupled with powerful classical digital hardware. A key challenge is that decoder demand fluctuates unpredictably: bursts of activity can require orders of magnitude more decodes than idle periods. Provisioning hardware for the worst case wastes resources, while provisioning for the average case risks catastrophic slowdowns. We show that this mismatch is a systems problem of capacity planning and scheduling, and propose a two-level framework that treats decoders as shared accelerators managed by the quantum operating system. Our approach reduces decoder requirements by 10–40% across fault-tolerant benchmarks, demonstrating that efficient decoder scheduling is essential to making FTQC practical.

1 Introduction

Quantum computers are not just faster machines; they are transformative domain-specific accelerators designed to solve problems that are fundamentally intractable for classical systems, enabling breakthroughs in cryptography, search, chemistry, and materials discovery [3, 33, 44, 57]. However, their viability depends on continuous, effective quantum error correction (QEC). QEC protects information by repeatedly computing and tracking parities between qubits. QEC is necessary since quantum systems are intrinsically noisy: every gate, idle step, or measurement risks error, and without correction, noise will overwhelm computation. QEC is thus not just a hardware feature but the foundation of the software stack for fault-tolerant quantum computing (FTQC), where user programs run alongside a perpetual loop of error detection and correction. The need for FTQC systems has prompted comprehensive industry roadmaps to build warehouse-scale quantum computers by the end of this decade [32, 36, 51].

At the core of QEC are *decoders*—classical algorithms that process measurement outcomes called *syndromes* (bit-strings indicating which parity checks detected an error) to identify and correct errors. The preparation and measurement of one syndrome constitutes a QEC cycle. Decoding is generally NP-hard [37] yet must finish within microsecond-level QEC cycles. Software alone cannot meet these demands on leading platforms such as superconducting qubits, making *hardware decoders* essential [2, 16, 17, 47, 52, 69, 72, 73]. Industry efforts include NVIDIA's CUDA-QX GPU-accelerated tensor-network decoders [49] and Riverlane's "Deltaflow" FPGA/ASIC stack for real-time decoding across hundreds of qubits [54]. These decoders are costly, shared resources whose performance directly affects system throughput and reliability. Their role can be summarized by the "**four C's**":

- *Classical:* serve quantum systems but run on classical hardware (CPUs/FPGAs/ASICs/GPUs).
- *Configurable:* adapt to diverse codes and noise channels.
- *Critical:* real-time in lockstep with quantum hardware.
- *Catastrophic:* any delay or failure breaks fault-tolerance.

A natural way to accelerate decoding is through parallelism via *windowed decoding*, where a decoding task is divided into smaller regions ("windows") processed simultaneously [10, 25, 59]. We refer to highly parallel variants as *parallel windowed decoding (PWD)*. While this reduces latency, it multiplies hardware demand, since each window needs its own decoder. Even one decoder per logical qubit is costly, but parallelism makes the requirement prohibitive. For example, a modest 32-bit quantum adder uses about 300 logical qubits; parallel decoding inflates demand by 3–5×, pushing the total beyond a thousand decoders even for a small quantum program. The burden is compounded because decoding is required not only for data qubits but also for numerous auxiliary qubits that support computation and movement. In many cases, auxiliaries even outnumber data qubits, further amplifying decoder demand. One major driver of this variability is *lattice surgery*, a technique that implements logical operations by dynamically merging and splitting encoded logical qubits.

All decoding tasks are not created equal – some are more urgent than others. We define a *critical decode* as one whose result is immediately required for program progress. Qubits actively involved in computation (and thus requiring critical decodes) must be decoded immediately to continue program computation, while idle memory qubits can tolerate some

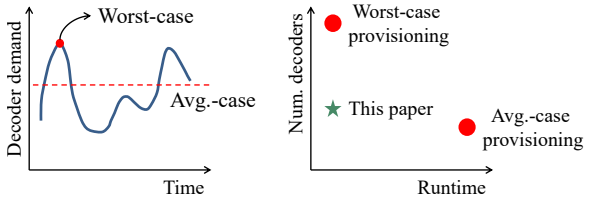


Figure 1. Decoder provisioning highlights the tension between worst-case (wasteful) and average-case provisioning (risky), motivating elastic decoder scheduling.

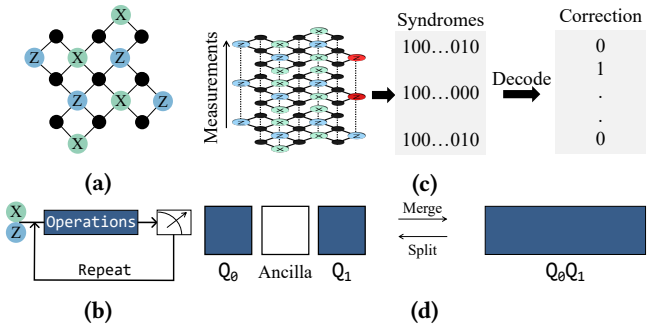


Figure 2. (a) A surface code logical qubit (patch); (b) QEC works by repeatedly performing operations followed by measurements to generate syndromes; (c) Decoding with measurement errors – multiple rounds of measurements are decoded collectively; (d) The fundamental split and merge operations of lattice surgery used for logical computation: each square is a logical qubit shown in (a).

delay. This asymmetry creates an opportunity: we can time-multiplex decoders by prioritizing active qubits on the critical path while buffering and deferring decoding of idle qubits. By sharing a smaller pool of decoders across both active and idle qubits in this manner, we can reduce hardware overhead while meeting the strict real-time deadlines of FTQC.

From a systems perspective, this is a classic utilization problem analogous to accelerator scheduling in classical platforms such as GPUs, where finite compute units must be multiplexed across dynamic workloads. Figure 1 shows that over-provisioning guarantees performance but wastes resources, while average-case provisioning risks severe stalls under bursts. We therefore introduce the notion of *elastic decoders*: a scheduling abstraction in which a fixed physical pool of decoder hardware is dynamically allocated to logical qubits based on demand. This differs from the traditional systems notion of *elastic resources*, which typically assumes that additional hardware can be provisioned on demand. In contrast, decoder elasticity refers strictly to time-multiplexing a fixed hardware budget. Our study shows that decoder demand is highly variable, driven by bursts of lattice surgery operations and sporadic non-Clifford gates. Static allocation fails under these conditions. Instead, we treat decoders as shared accelerators managed by the quantum operating system (QOS) [14, 31], and introduce scheduling policies that balance throughput, latency, and fairness.

This paper is the first to address the *capacity planning* problem for FTQC: given a workload, how many hardware decoders should be provisioned to ensure timely error correction without over-provisioning costly resources? We argue that decoder allocation and scheduling are core responsibilities of the QOS. The decoding infrastructure is the backbone of any FTQC system, and is thus an integral part of the QOS.

This paper makes the following contributions:

- **Workload characterization.** We quantify decoder demand across a range of benchmarks (QFT, Shor, adders, Ising), showing that decoder demand comes in bursts and dominated by a small set of critical operations. This variability makes static allocation ineffective.
- **Capacity planning.** We present the first systematic study of how many hardware decoders are required to sustain FTQC workloads. We show that provisioning for the worst case wastes up to 40% of decoders, while provisioning for the average case risks catastrophic slowdowns.
- **Scheduling framework.** We introduce a two-level scheduling design for the quantum operating system (QOS): coarse-grained scheduling to prioritize critical-path and spatially-coupled decodes, and fine-grained scheduling to fairly allocate remaining decoders across qubits.
- **Scheduling policies.** We evaluate several fine-grained policies—Most Frequently Decoded (MFD), Round-Robin (RR), and Minimize Longest Undecoded Sequence (MLS). MLS consistently achieves the best tradeoffs, reducing decoder needs by up to 19% compared to RR.
- **Open-source workflow.** We release an open-source workflow [56] that compiles programs into lattice surgery IR, tracks decoder demand, and enables reproducible evaluation of scheduling policies. Using this, we demonstrate that careful scheduling yields 10–40% decoder savings even with long decoder tail latencies.

2 Quantum Error Correction and Decoding

In this section, we cover high-level details of Quantum Error Correction (QEC) and the role of decoders.

2.1 Quantum error correction

Qubits are fragile: a variety of processes can introduce an error in a qubit's state. Such errors occur far more often than in classical storage (e.g., SRAM, DRAM, RAID) [53], preventing reliable execution on today's devices. Quantum Error Correction (QEC) mitigates this by encoding a few logical qubits into many physical ones—similar in spirit to ECC in DRAM but able to correct both bit- and phase-flip errors. Among candidate codes, the surface code [24] is especially compelling for its compatibility with current hardware connectivity and is the focus of this work, though our ideas generalize to other codes.

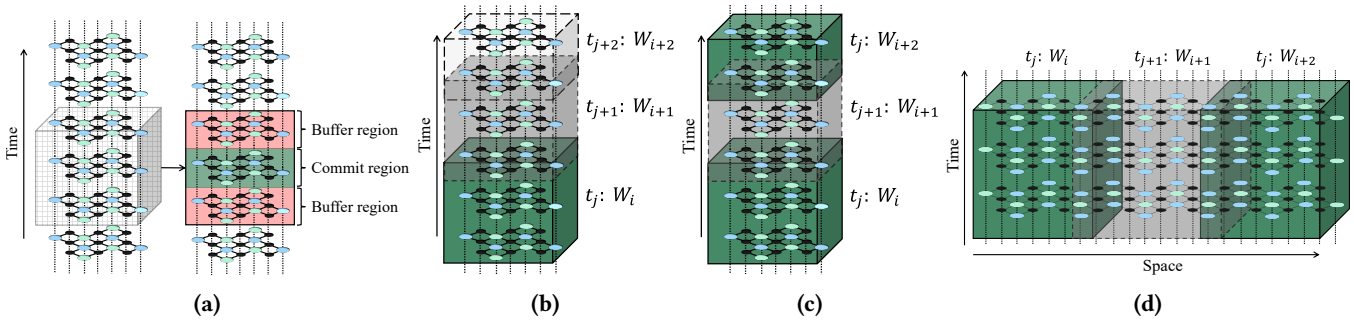


Figure 3. (a) Streams of syndromes are processed in windows represented by *blocks* which are fixed decoding volumes: each block consists of a region in which corrections are committed, while buffer regions are used for merging corrections with overlapping windows; (b) Sliding windowed decoding (SWD) is inherently sequential: window W_{i+1} can only be processed after W_i has been processed; Parallel windowed decoding (PWD) can be (c) temporal and/or (d) spatial: in either case, W_i, W_{i+2} are treated as independent tasks and processed independently in parallel at time step t_j. W_{i+1}, which includes overlaps with W_i, W_{i+2}, is processed in the next time step.

QEC codes are defined by their code distance d, which determines how many errors can be reliably detected and corrected. A surface code logical qubit of distance d, for example, can correct error chains up to length $\frac{d-1}{2}$. Figure 2(a) illustrates a $d = 3$ surface code logical qubit comprising many physical qubits: data qubits (black) and check qubits (X and Z, which detect phase- and bit-flips, respectively). Each QEC cycle applies a sequence of gates (Figure 2(b)) and measures all check qubits, producing syndromes that reveal the presence and type of errors for correction. Every cycle takes T_{cycle} time units.

2.2 Decoding

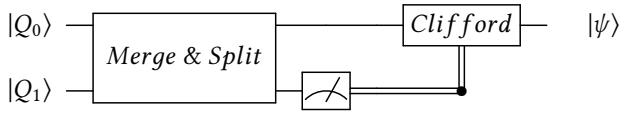
A QEC code is defined by its parity-check matrix H, which specifies the relationship between check and data qubits in the encoded logical qubit(s). Decoding can be expressed as solving a linear system for F:

$$HF = \sigma$$

Where σ is the measured syndrome and F is a correction that restores the logical state. Since H typically has more columns than rows, this system is *under-determined*, yielding many possible corrections—a challenge unique to quantum codes, known as *quantum degeneracy*. QEC decoders thus rely on maximum likelihood decoding (MLD), selecting the correction most probable for a given set of errors that can affect the system, making it an NP-Hard problem. Since every operation involving qubits is susceptible to errors, measurements of the check qubits can be erroneous too. For this reason, syndromes are collected over multiple measurement rounds and decoded collectively, as shown in Figure 2(c).

2.3 Logical operations

Once logical qubits are encoded, computation proceeds through logical operations between them. In the surface code, the leading method is *lattice surgery* (LS) [23, 35, 40], which consists of two primitives (Figure 2(d)): *merge*, which combines multiple logical qubit patches into one, and *split*, which divides a patch into two. Sequencing these primitives enables a wide range of logical operations.



In the circuit above, Q_1 is measured after lattice surgery and its outcome drives a corrective *feed-forward* operation on Q_0. Since Q_1 is a logical qubit, its measurement outcome cannot be read directly — it must first be inferred by a classical *decoder* processing noisy syndrome data. This places the decoder on the critical path of execution: computation cannot proceed until decoding completes. We call such decodes **critical decodes**. This imposes a hard real-time requirement: the average decoding latency T_{decode} must satisfy $T_{\text{decode}} < T_{\text{cycle}}$, where T_{cycle} is the QEC cycle time. If violated, unprocessed syndromes accumulate and computation suffers an exponential slowdown — the **backlog problem** [63].

2.4 Processing streams of syndromes

Logical qubits continuously generate syndromes that must be decoded. Instead of decoding all syndromes at once, decoding is performed on *windows* of syndromes. Each window of size n_W forms a 3-D decoding volume, visualized as a block in Figure 3(a). A block is divided into buffer and commit regions. Buffer regions are required to handle corrections spanning multiple overlapping blocks. Corrections fully contained in a commit region are finalized, while buffer-region corrections are deferred. With this block-based structure, syndrome streams can be decoded using the following paradigms.

2.4.1 Sliding windowed decoding (SWD). SWD is inherently sequential, as shown in Figure 3(b). At time step t_j, the decoder processes window W_i. Once complete, the

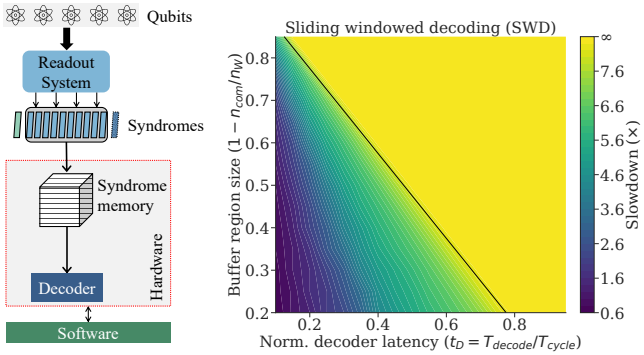


Figure 4. (a) A full system using hardware (FPGA/GPU/ASIC) decoders: qubits are measured using the readout system, which produces syndromes that are buffered until a window has been collected. Then, the decoder uses these collected syndromes to determine a correction, which is communicated to software; (b) Effect of the normalized decoder latency t_D on the slowdown in processing $5d$ rounds of syndromes using SWD. The buffer region size is the normalized size of the buffer regions required ($n_W = 3d, n_{com} = d$ would yield an overlap of 0.66).

next window is processed at the following step. The overlap between W_i and W_{i+1} corresponds to their buffer regions.

2.4.2 Parallel windowed decoding (PWD).

PWD [59] extends SWD by exploiting temporal (Figure 3(c)) and spatial (Figure 3(d)) parallelism. In Figure 3(c), windows W_i and W_{i+2} are independent and can be decoded simultaneously with two decoders. Window W_{i+1} is processed in the next step, with buffers overlapping those of W_i and W_{i+2}. Similarly, in Figure 3(d), a large patch formed by lattice surgery can be decoded using spatial parallelism [25]. PWD increases decoding throughput but requires more decoders. As shown in Figures 3(c), 3(d), compared to SWD, at least one extra decoder is needed. In general, decoder requirements grow linearly with the size of the decoding task in both temporal and spatial dimensions.

3 Decoder Allocation is Hard

In this section we will discuss the need for accelerated, hardware decoders, and how it is non-trivial to allocate these decoders for a system in a resource-efficient manner.

3.1 Hardware decoders

The requirement for syndromes to be processed faster than they are generated has motivated research to build fast and accurate hardware decoders for the surface code [2, 5, 52, 60, 69, 72, 73], especially for systems using superconducting qubit architectures due to their fast gate times. GPU-accelerated decoding has also been demonstrated for neutral atom systems [48]. Figure 4(a) shows the general structure of a real-time decoding system that uses accelerated hardware decoders. Syndromes are generated after readout, and these syndromes are buffered until a decoder can operate on them. Such hardware systems are necessary for ensuring that the backlog problem is avoided [63]. Figure 4(a) also shows that T_{decode} is dependent on not just the actual decoder latency but also the communication/memory access latencies required for storing and providing the syndromes to the decoder.

3.2 SWD is not scalable

As discussed in Section 2.4, decoding in the SWD paradigm is inherently sequential. This makes SWD too slow and fundamentally unscalable for large systems. Figure 4(b) illustrates this issue for $5d$ rounds of pending syndromes from a single logical qubit. The figure shows slowdown as a function of buffer region size and normalized decoder latency $t_D = T_{decode}/T_{cycle}$. Slowdown accounts for both the initial $5d$ rounds of pending syndromes and the additional syndromes generated during their processing; we measure the time required for the decoder to catch up to the live syndrome stream. Buffer regions are expressed in multiples of code distance d, so a buffer of size d corresponds to d rounds of overlap between adjacent windows. Larger buffer regions improve accuracy in windowed decoding [25, 59], but also increase slowdown. The results are clear: SWD avoids the backlog problem only for unrealistically small buffer sizes and very optimistic t_D. For instance, $t_D = 0.2$ in a superconducting system with $T_{cycle} \simeq 1\,\mu s$ requires an average decoder latency of 200 ns — far below what any proposed hardware decoder has achieved. When $t_D > 0.8$, the slowdown diverges, meaning the decoder can never catch up with the incoming syndrome stream.

3.3 PWD introduces non-determinism

Figure 5(a) shows the same initial problem used for the SWD paradigm in Figure 4(b): even for values of t_D close to one, the slowdown is still finite. This example clearly highlights the utility and scalability of the PWD paradigm – Figure 5(a) would be the same for any initial size of pending syndromes.

This instance of PWD exploits *temporal* parallelism: windows in time are executed in parallel to increase decoder throughput. However, as discussed in Section 2.4, PWD can also have a spatial interpretation. We now discuss how the spatial interpretation of PWD is non-deterministic and its interplay with the temporally PWD.

3.3.1 The impact of gate routing.

Logical operations in the surface code are performed via lattice surgery, where qubits are merged through ancilla regions that form routing paths. The size of these ancilla bridges varies, leading to unpredictable decoding volumes. Figure 5(b) illustrates this in the EDPC layout [8], where blue squares are logical qubits and white squares are ancillas. Ancilla 1 connects nearby

qubits with a short bridge of length 3. Ancilla 2 connects similarly adjacent qubits but requires a longer path of length 7 to avoid Ancilla 1. Ancilla 3, unaffected by others, also has length 7 since it spans distant qubits. Such routing conflicts and varying patch sizes create non-deterministic decoder demands, as the number of required decoders scales linearly with patch size (Section 2.4).

3.3.2 Interplay of temporal and spatial requirements.

Consider the logical program shown in Section 2.3. This lattice surgery operation could be between qubits that have either a large temporal decoding task (pending syndromes), a large spatial decoding task (due to the physical distance and the subsequent routing between them), or both. We attempt to capture this interplay between the spatial and temporal decoder requirements in Figure 5(c). In the simplest case, both qubits do not have any pending syndromes, and they are next to each other on the device, which means that they need at most a decoder each to decode all generated syndromes. However, if there is a spatial and/or temporal element to the decoding task, the number of decoders required can explode to up to 50×, just for two logical qubits!

3.4 Need for decoder scheduling

In the sections above, we showed that the number of decoders required under the PWD paradigm can be highly variable and, in some cases, untenable when both spatial and temporal decoding demands grow. Figure 6 highlights this variability in the spatial dimension: different benchmarks (see Section 6.2 for more information on benchmarks) exhibit a wide range in the number of merged patches, with significant outliers. This directly affects the number of logical qubits active at any given time, as illustrated in Figure 7, which shows the distribution of active qubits across benchmarks. Since every logical qubit continuously produces syndromes that must be decoded, the key challenge becomes:

> How do we allocate decoders for benchmarks with such diverse and dynamic demands?

The simplest allocation strategies are to provision (a) for the worst-case demand (i.e., the maximum number of active qubits from Figure 7), or (b) for the average-case demand. Option (a) is prohibitively resource-intensive, since each decoder may require nearly all the resources of a single FPGA [72, 73]. Option (b) risks substantial slowdowns: spatial decoding tasks of large size would be delayed, reducing throughput and degrading program performance.

Thus, static allocation is insufficient. The problem of decoder allocation is better viewed as a *scheduling problem*, where a finite pool of decoders must be dynamically assigned to decoding tasks over time. Like CPU scheduling in classical computers, decoder scheduling must balance throughput, latency, and fairness, while avoiding bottlenecks from workload spikes. In this paper, our goal is to determine how many decoders are needed and how they should be scheduled to ensure that benchmarks execute without slowdowns.

4 Coarse-Grained Scheduling

We define decoder scheduling at two levels—coarse-grained and fine-grained—as illustrated in Figure 8. Coarse-grained scheduling prioritizes critical decodes and resolves conflicts between spatial and temporal tasks on the same qubits. For example, in Figure 8, the spatial decode on qubits Q_0, Q_1 from a lattice surgery operation takes precedence over the temporal decode. After coarse-grained allocation, remaining decoders are assigned to other qubits. In the figure, Q_2 and Q_N receive decoders while Q_3 does not, demonstrating fine-grained scheduling, which selects among tasks from individual qubits. Because critical and spatial decodes impose rigid constraints, we adopt a single coarse-grained policy

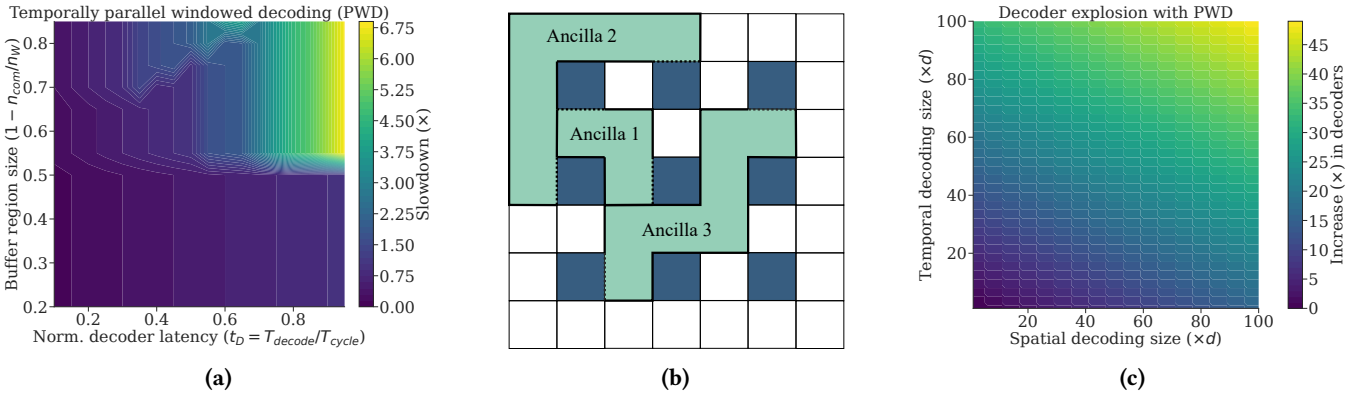


Figure 5. (a) Using simulations, effect of the normalized decoder latency t_D on the slowdown in processing $5d$ rounds of syndromes using Temporally parallel windowed decoding (PWD); (b) Logical qubits arranged in the EDPC layout with ancilla bridges of varying sizes to facilitate LS operations; (c) PWD is applicable in both space *and* time – this leads to an a variable and sudden increase in the number of decoders required for implementing PWD (assuming $t_W = 3d$).

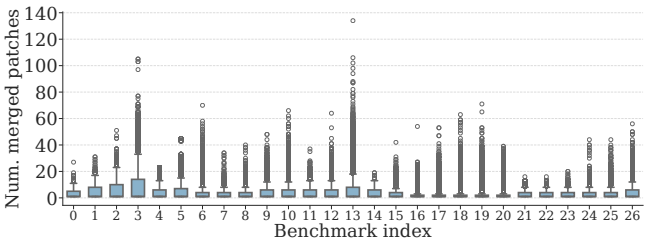


Figure 6. Distribution of the number of patches merged to perform lattice surgery-based logical measurements for different workloads – most workloads require short-range interactions on average, but some large outliers are possible due to routing constraints.

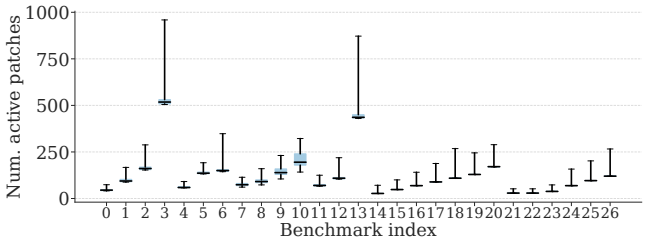


Figure 7. Distribution of the decoder demand over time for different benchmarks – these error bars capture the fluctuations in active logical patches that require decoding during the computation.

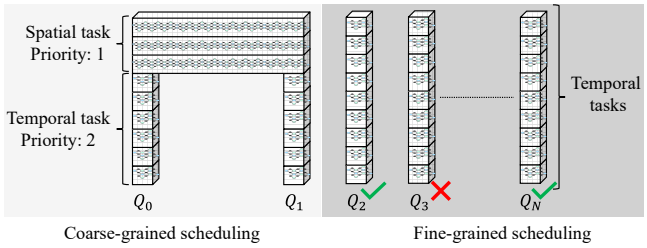


Figure 8. Distinction between coarse- and fine-grained scheduling. (Left) Coarse-grained decoder scheduling handles the prioritization between spatial and temporal decoding tasks. This includes handling critical decodes, which will always have some spatial component to them due to lattice surgery. (Right) Any decoders left over after handling critical decoding tasks are assigned to the temporal decoding tasks associated with logical qubits not involved in any operation in that time step. Scheduling decoders among this subset of qubits is what we refer to as fine-grained scheduling.

to maximize utilization, while Section 5 presents multiple fine-grained policies that operate in tandem.

4.1 Decoding space-time volumes

Figures 9(a), 9(b), and 9(c), show space-time decoding volumes for three distinct scenarios under the PWD paradigm. Decoding volumes could be simple in both space and time, or require multiple decoders for a large volume in time, or have a large volume in *both* space and time. Each case shows how the decoding volume can be different in the same program, which requires decoders to be *elastic* such that an increase or decrease in the demand can be dealt with without any detrimental effects on the performance of a program.

4.2 Prioritizing between spatial and temporal parallelism

As shown in Figure 9(c), lattice surgery can require decoders for both spatial and temporal parallelism, raising a key scheduling question: **Which type of parallelism should take priority when both are needed simultaneously?**

Consider first the effect of decoder scarcity on spatial PWD. Suppose a large patch requires three decoders: the first layer uses all three for non-overlapping windows, while the second layer reuses two to handle overlaps and boundary errors [10, 25]. If only one decoder is available, the protocol cannot proceed as designed, forcing one of three fallbacks: (i) switch to a monolithic decoder, (ii) serialize window decoding, or (iii) defer decoding to the next slice. Option (iii) risks slowdown if followed by a critical non-Clifford operation. As Figure 10 shows, spatial PWD and monolithic decoding yield similar logical error rates (LERs), but monolithic decoding incurs far higher latency for large patches[1]. While demonstrated using PyMatching [34], the same trends hold for hardware decoders. Thus, fallback options (i) and (ii) degrade performance, and spatial PWD loses its latency advantage without concurrent decoder availability.

In contrast, temporally PWD is more resilient to decoder constraints. As a parallelized form of sliding windowed decoding (SWD), it can always fall back to SWD when decoders are scarce. Available decoders are simply assigned to the oldest undecoded windows, ensuring correctness. Once more decoders free up, temporally PWD clears accumulated backlogs in *constant time*, provided sufficient parallelism is available. This adaptability is its main strength: **as long as delays do not exceed the system's recovery capacity, efficient scheduling remains feasible despite temporary decoder shortages.**

4.3 Distinguishing between spatial decoding tasks

Lattice surgery operations that require spatially PWD can either be due to critical decodes arising due to non-Clifford gates, or due to other operations between Y-states [70]. Critical decodes must be serviced, which is why those spatial decoding tasks are given the highest priority, followed by any temporal decoding tasks needed for the critical decode. If there are non-critical spatial decoding tasks required at the same time step, they are given a priority lower than critical decodes but higher than that of decodes needed by logical qubits not involved in any operation in that time step. This

[1] Spatial PWD is only beneficial beyond $3d$ data qubits (or ~ 3 patches); smaller patches are treated as monolithic in our evaluations.

allows spatial decoding tasks, critical or not, to be prioritized over all other temporal decoding tasks.

5 Fine-Grained Scheduling Policies

In this section, we define fine-grained decoder scheduling policies. These policies act on decoders that are available after all other higher priority spatial decoding tasks have been allocated decoders. These fine-grained policies are similar to process scheduling policies used by modern operating systems where decoding tasks of individual qubits are analogous to processes and decoders are analogous to CPU cores.

5.1 Connection with real-time scheduling systems research

Decoder scheduling maps naturally to a **mixed-criticality, soft real-time model** [6, 68]: decoding windows are sporadic tasks with bursty arrivals; critical decodes (from non-Clifford gates) impose firm deadlines on dependent computation; non-critical decodes tolerate bounded delay; and the decoder pool represents a set of identical parallel processors. Our two-level scheduler mirrors hierarchical scheduling architectures [18]: coarse-grained scheduling handles criticality levels (critical > spatial > temporal), analogous to priority-based admission control; fine-grained policies allocate residual capacity among admitted tasks for backlog control.

While conventional real-time systems research often provides hard schedulability guarantees (e.g., rate-monotonic analysis [42]), this approach is unattainable for QEC decoders due to fundamental differences:

1. **Non-stationary arrivals:** Task generation depends on dynamic program structure—lattice surgery operations, gate scheduling, and routing decisions—rather than fixed periodic or sporadic arrival models with known minimum inter-arrival times.
2. **Data-dependent execution:** Decoding latency varies with syndrome patterns and error distributions, making tight worst-case execution time (WCET) bounds impractical. Even for fixed code distance, execution time can vary 2-3× based on error chain complexity.
3. **Elastic resource requirements:** Spatial decoding tasks require anywhere from 1 to 50+ decoders depending on patch size and routing distance—a form of moldable parallelism [45] uncommon in traditional real-time systems.
4. **Temporal dependencies:** Overlapping decoding windows create precedence constraints between consecutive tasks on the same logical qubit, violating the task independence assumption central to most schedulability analyses.

Despite these constraints, certain scheduling principles from queuing theory and soft real-time systems remain applicable. Work-conserving policies that account for task backlog and waiting time can prevent starvation and control queue growth under bursty, non-stationary arrivals [62]. Similarly, policies that prioritize tasks based on accumulated service deficits can provide fairness guarantees without requiring detailed arrival models [58]. These insights inform our fine-grained policy design, which we describe next.

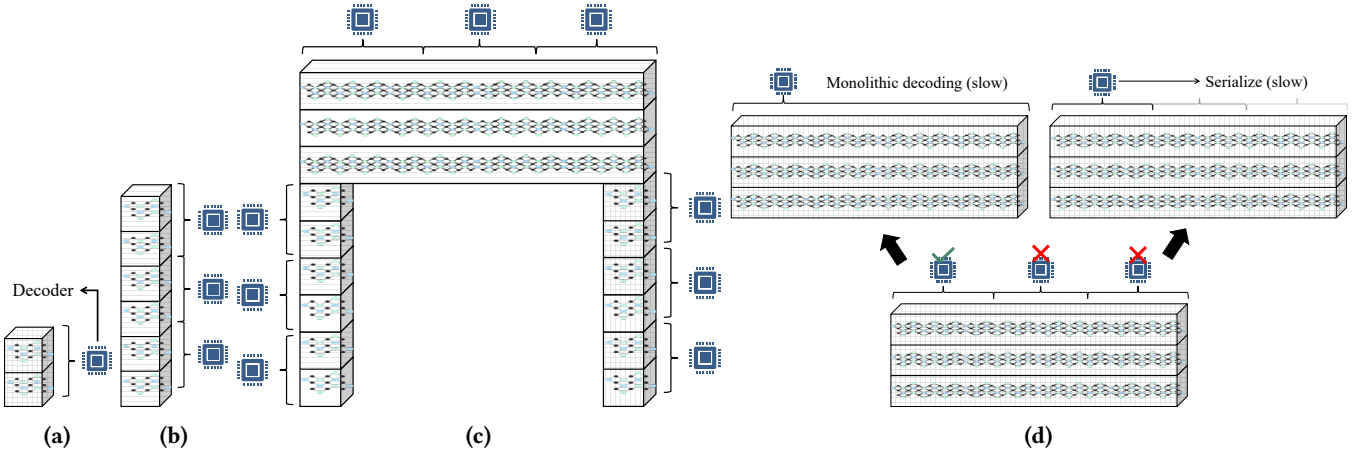


Figure 9. (a) Decoding volume of a single logical patch idling for a few rounds—only one decoder is needed; (b) Idling for multiple rounds creates a linear growth in windows, requiring temporally parallel windowed decoding (PWD); (c) A lattice surgery merge between distant patches temporarily increases decoder demand due to spatial parallelism, which may coincide with temporal parallelism (overlapping windows omitted for clarity); (d) Spatial PWD is not robust to decoder shortages—if two of three required decoders are unavailable, one must either decode the merged patch monolithically or serialize the protocol, both slow and undesirable. Deferring the task is also problematic, as the next slice may contain a critical decode that must wait for the deferred task, further slowing execution.

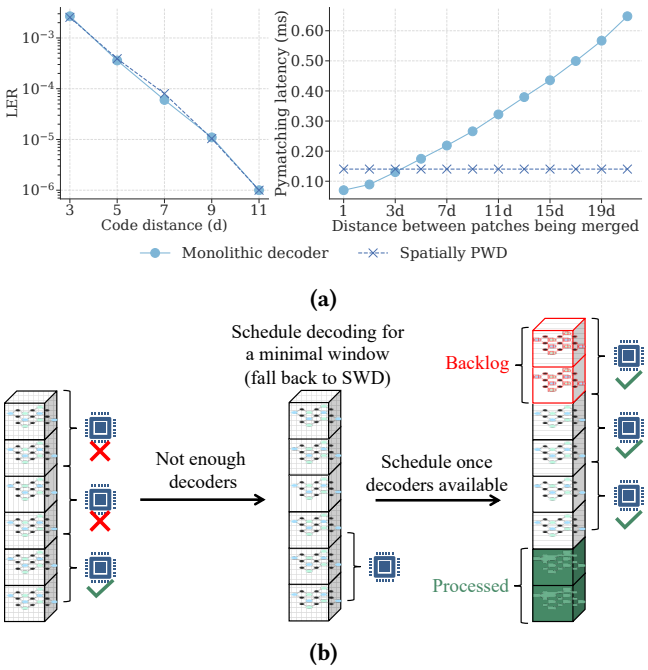


Figure 10. (a) (Left) Logical error rates (LER) for various lattice surgery configurations with and without spatially parallel window decoding (PWD) at circuit-level noise $p = 0.1\%$. (Right) Spatial parallelism does not affect LER but reduces decoder latency when merged patches are separated by $\geq 3d$ data qubits, assuming sufficient decoders for all distances (for $d = 11$, window size = $3d$). (b) Temporal PWD remains robust under decoder resource constraints: with only one of three decoders available, the decoding task can be processed incrementally by the single decoder. This creates a qubit backlog, which can be rapidly cleared once additional decoders become available via temporal parallelism.

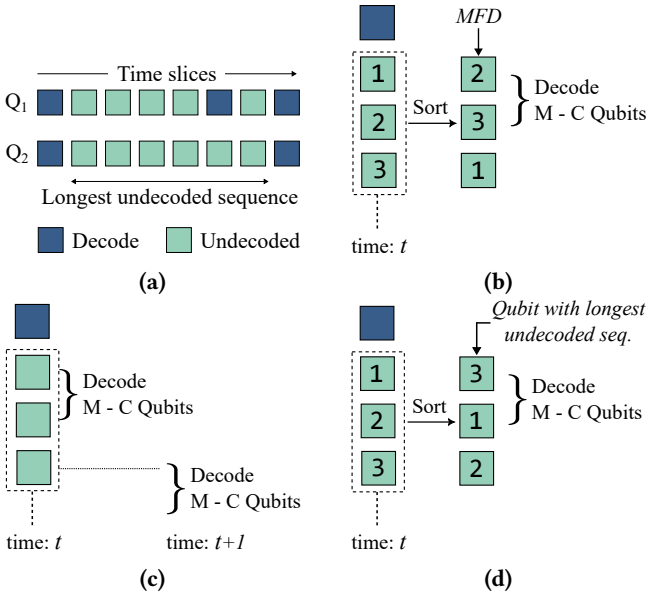


Figure 11. (a) Illustration of the longest undecoded sequence – Q_1 has the longest undecoded sequence before the last decode (each square is a decoding block/window); (b) `MFD` policy – undecoded qubits are sorted according to the number of critical decodes they are involved in after time slice t and $M - C$ qubits are selected from this sorted list; (c) `RR` policy – the $M - C$ qubits decoded in time slice t are not decoded in time slice $t + 1$; (d) `MLS` policy – undecoded qubits are sorted according to their undecoded sequence lengths and $M - C$ qubits are selected from this sorted list.

5.2 Scheduling for individual qubits

Fine-grained scheduling determines when each logical qubit's syndromes (current and pending) are provided access to a decoder. If a policy starves qubits of decoding, its measured syndromes will accumulate, leading to longer decoding latencies (since the total size of the decoding problem increases with time). This thus necessitates fine-grained policies to ensure fair access to decoders for all qubits that are not involved in lattice surgery operations.

Longest undecoded sequence: To quantify the fairness of a fine-grained scheduling policy, we use a metric '*Longest Undecoded Sequence*', which measures how well the decoders are servicing all logical qubits. A large undecoded sequence length implies that a qubit has been left undecoded for a long time. A qubit left undecoded for a long time will also require more processing from the decoder for it to be up to date with the latest syndrome. Figure 11(a) shows an example of determining the longest undecoded sequence length.

Consider an arbitrary time slice t in the execution of a quantum program. There are N logical qubits and M hardware decoders ($N > M$). All decoding scheduling policies will have two components: The first will assign the decoders necessary for all critical decodes C in the time slice t. The second will assign all the remaining $M - C$ hardware decoders to the $N - C$ qubits based on the scheduling policy used. We now discuss three decoder scheduling policies (all policies are illustrated in Figure 11(b) – Figure 11(d)):

5.2.1 Most frequently critically decoded (`MFD`). A logical qubit that consumes a significant number of T-states during the execution of a program would have a frequent requirement of critical decodes – leaving such a logical qubit undecoded for more than a few slices would make subsequent critical decodes take longer, thus slowing down computation. This motivates the `MFD` scheduling policy that prioritizes decoding of logical qubits that have numerous critical decodes in the future at any given time (Figure 11(b)). The `MFD` policy will ensure that future critical decodes have a minimized number of undecoded syndromes for qubits with frequent critical decodes. The `MFD` policy is predicated on

quantum programs being static in nature. While logical operations to perform S corrections during magic state consumption represent 'dynamic' instructions, they do not add any additional non-Clifford gates to the program, thus ensuring that any statically compiled program using the MFD policy will always work.

5.2.2 Round-robin (RR). Derived from CPU scheduling policies used in operating systems, the RR policy does not prioritize any specific logical qubits – rather, it chooses $M-C$ qubits in a round-robin manner (Figure 11(c)) in every time slice to ensure fairness for all qubits in the system. This policy was also alluded to in prior works [11, 17] for scheduling decoders for different logical qubits.

5.2.3 Minimize longest undecoded sequence (MLS). The longest undecoded sequence length at any time slice is an indicator of how well the fine-grained scheduling policy is servicing all qubits in the system. We use this as the base for the MLS policy, which greedily minimizes the longest undecoded sequence at every time slice (Figure 11(d)). The MLS policy works as follows: at any time slice t, qubits are sorted on the basis of their current undecoded sequence lengths. Then, $M - C$ qubits with the largest undecoded sequence lengths are assigned hardware decoders. The MLS policy mimics longest-queue-first (LQF) scheduling [21], known to prevent starvation and control backlog growth under bursty demand without strong arrival distribution assumptions.

5.3 Magic state cultivation factories

Magic state cultivation [30] is a protocol that generates high-quality non-Clifford (magic) states by using one or more smaller, more error-prone logical qubits. These cultivation protocols do not require high-performance decoders during the state generation process. Instead, they rely on **post-selection**, where attempts are *discarded* as soon as a syndrome bit-flip is detected. Post-selection is well-suited for non-deterministic state preparation, providing exponential error suppression at the cost of success probability [30, 41]. While some protocols support active error correction [13], we assume fully post-selected cultivation protocols for our evaluation, resulting in no decoder overhead during factory operation. However, the final output magic state is stored in a high-distance logical qubit for long-term use, and thus requires decoding to ensure reliability after successful preparation. We include these magic state storage patches in our analysis.

5.4 Scheduling policy overheads

While fine-grained scheduling policies provide performance benefits, they introduce computational and memory overheads that must be quantified. Table 1 summarizes the time and space complexity of each policy.

Table 1. Time and space overheads for all fine-grained scheduling policies.

Policy	Per-slice time complexity	Metadata per qubit	Total metadata storage
RR	O(1)	1-bit flag	$N_{decoders}$ bits
MFD	O(1)	1 counter (static)	$N_{decoders} \times log$(Total time steps) bits
MLS	O(NlogN)	1 counter (dynamic)	$N_{decoders} \times log$(d × Total time steps) bits

5.4.1 Time complexity. RR and MFD both achieve $O(1)$ scheduling decisions per time slice by maintaining simple state: RR uses a round-robin counter, while MFD accesses a precomputed static priority ordering based on future critical decode counts (determined at compile time). In contrast, MLS requires sorting N qubits by their undecoded sequence lengths, requiring $O(N \log N)$ comparisons. For typical workloads with $N \approx 100$–1000 active qubits, this translates to ∼700–10,000 comparisons per slice.

5.4.2 Space complexity. All policies maintain per-qubit metadata. RR requires only a single bit indicating whether each qubit was decoded in the previous slice. MFD stores a counter of remaining critical decodes for each qubit, requiring $\lceil \log_2 T \rceil$ bits to represent counts up to the total program length T. MLS tracks the current undecoded sequence length, which can grow up to dT in the worst case (accumulating d syndrome rounds per time slice for T slices), requiring $\lceil \log_2(dT) \rceil$ bits per qubit. For representative parameters ($N = 800$, $T = 10^5$, $d = 11$), total metadata ranges from 100 bytes (RR) to ∼2.4 KB (MLS)—negligible compared to syndrome buffer memory requirements (multiple MBs).

5.4.3 Off-critical-path execution. Critically, scheduling latency does not impact end-to-end execution time because it occurs *off the critical path*. The decoding pipeline operates as follows:

1. At time slice t, the quantum hardware executes operations and generates syndromes for all active qubits
2. While syndromes are being read out and buffered (requiring $T_{\text{readout}} \sim 100$–$500$ ns per qubit), the classical control processor computes the scheduling decision for slice $t + 1$
3. Once M decoders become available (having completed their previous windows), they are immediately assigned to qubits according to the computed schedule
4. Decoding proceeds in parallel across all M decoders

The key insight is that syndrome readout, buffering, and decoder processing all occur in parallel with scheduling for the *next* slice. As long as scheduling completes before decoders finish their current windows—i.e., $T_{\text{schedule}} < t_D \cdot T_{\text{cycle}}$—there is no additional latency overhead because of scheduling.

5.4.4 Concrete overhead analysis. For MLS with $N = 800$ qubits and $t_D = 0.5$ (normalized decoder latency), each decoder processes a window in $0.5 \times 1\ \mu s = 500$ ns. With $M = 400$ decoders operating in parallel, aggregate decoding

throughput is 400 windows per 500 ns. Meanwhile, sorting 800 integers on a modern CPU (2 GHz, ∼1 cycle per comparison for integer comparisons) requires ∼8000 comparisons × 0.5 ns/cycle ≈ 4 μs. Since this is $\ll t_D \cdot T_{\text{cycle}} = 500$ ns per individual decoder but occurs in parallel with all decoder execution, the effective overhead is negligible.

For FPGA implementations, priority queues based on heap structures reduce per-qubit updates to $O(\log N)$ operations. At typical FPGA clock rates (100–200 MHz), maintaining a heap of 1000 elements requires ∼10 cycles × 10 ns/cycle = 100 ns per insertion/update—well within the syndrome readout time budget.

6 Methodology

We now describe the methodology used to evaluate different decoder scheduling policies for the PWD paradigm.

6.1 Compiler

We use the Lattice Surgery Compiler (LSC) [70] to generate Lattice Surgery Instructions (LLI) Intermediate Representations (IR) of workloads that can be executed on an error-corrected quantum computer using the Surface code with Lattice Surgery. LSC can generate IR that denote Lattice Surgery instructions from the QASM [15] representation of a workload. The Lattice Surgery instructions generated by LSC are a combination of Clifford operations and multi-body measurements used for implementing Pauli-Product Rotations (PPR) [40]. LSC handles mapping and routing based on the layout provided to the compiler. We configure LSC to use a 'wave' scheduling that maximizes the number of concurrent instructions executed in every time slice. LSC uses Gridsynth [55] to synthesize arbitrary rotations.

6.1.1 Layout. We perform all evaluations using the Edge-Disjoint Path Compilation layout (EDPC) [8] with a single routing lane.

6.1.2 Magic states. LSC abstracts distillation factories away – it is assumed that the magic state storage sites arranged along the borders of the EDPC layout get magic states within a specified time period. We assume that the magic state production throughput is high enough for generating magic states every alternate slice.

6.1.3 Other configurations. Y-states are produced using the twist-based initialization [28].

6.2 Simulation Framework

Using the IR generated by LSC, we build a framework [56] that can parse the IR, determine the critical decodes in every slice, generate a timeline of all operations, and assign decoders to all logical qubits depending on the scheduling policy.

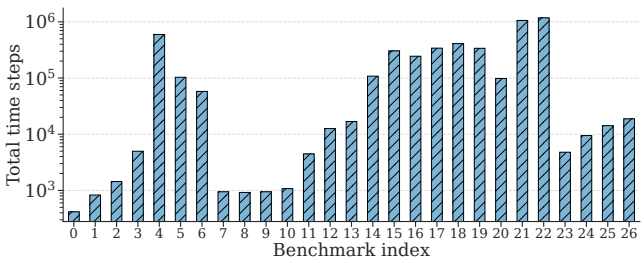


Figure 12. Total slices executed for each benchmark.

Benchmarks: We select benchmarks that are representative of large-scale programs that will run on FTQC systems. A majority of these will be building blocks for larger algorithms. We use the following benchmarks (benchmark indices annotated in parentheses, benchmark size in terms of number of qubits specified after the hyphen):

```
adder-28 (0), adder-64 (1), adder-118 (2), adder-433 (3),
↪ bwt-37 (4), bwt-97 (5), heisenberg-100 (6), ising-34 (7),
↪ ising-42 (8), ising-66 (9), ising-98 (10), multiplier-45
↪ (11), multiplier-75 (12), multiplier-350 (13), qft-20
↪ (14), qft-40 (15), qft-60 (16), qft-80 (17), qft-100
↪ (18), qft-120 (19), qft-160 (20), shor-9 (21), shor-15
↪ (22), wstate-20 (23), wstate-40 (24), wstate-60 (25),
↪ wstate-80 (26)
```

Other Software: Stim [27] was used for simulating stabilizer circuits to generate syndromes and error rates. Azure QRE [9] was used for resource estimations.

7 Evaluations

In this section, we simulate the effectiveness of different decoder scheduling policies for the parallel windowed decoding (PWD) paradigm.

7.1 Research Questions

The key research questions we would like to answer for the PWD paradigm is:

- Q1. What is the minimum number of decoders required to avoid slowdowns in the program runtime?
- Q2. How does this minimum number of decoders compare with the baseline system?
- Q3. How do the fine-grained scheduling policies compare with each other?
- Q4. How do decoder latencies affect decoder scheduling?

7.2 Benchmark statistics and baselines

7.2.1 Benchmark runtimes. LSC was configured to timeout after 10 minutes to keep simulation times tractable. Figure 12 shows the number of time steps for all benchmarks.

7.2.2 Concurrent critical decodes. Since all scheduling policies prioritize decoder allocation for any critical decode in a slice, the concurrency of critical decodes plays an important role in determining the number of decoders required for a benchmark. Figure 13 shows the difference between

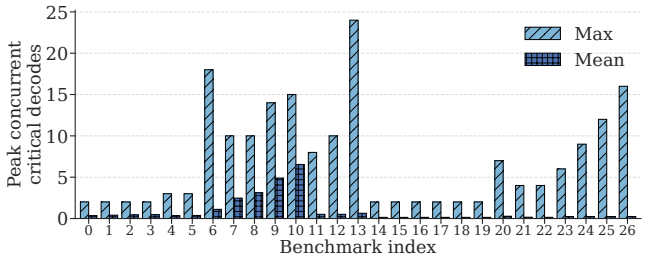


Figure 13. Peak and average concurrency for non-Clifford operations for all benchmarks – concurrency is sporadic.

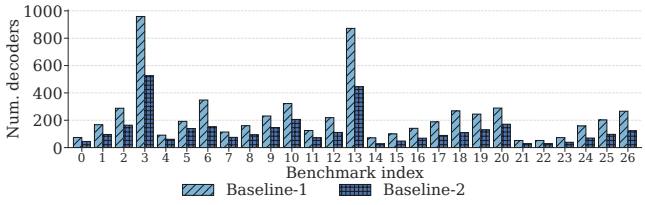


Figure 14. Number of decoders allocated for the baselines: Baselines-1 and 2 provision for the worst- and average-case decoder demands respectively.

the peak and average critical decode concurrency. This highlights the importance and need for performant decoder scheduling policies – the limited concurrency of critical decodes in quantum programs makes efficient scheduling of decoders important for high decoder utilization.

7.2.3 Baselines. We assume a baseline system to allocate decoders for every logical qubit. Since the number of active logical qubits can vary through the course of a program, we consider the following baselines:

1. Baseline-1: Provisions for the worst-case number of active logical qubits.
2. Baseline-2: Provisions for the average-case number of active logical qubits.

Baseline-1 caters for all possible qubits that could be active for a given benchmark, and thus requires a very simple scheduling policy wherein syndromes for a logical qubit are passed to its pre-assigned decoder. On the other hand, Baseline-2 will require some scheduling policy for choosing which logical qubits are decoded at any given instant. We consider the Round-Robin (RR) policy as the baseline policy since its use has been suggested in prior works that allude to scheduling decoders to reduce resource overheads [11, 17]. Figure 14 shows the number of decoders configured for the two baselines for all benchmarks. As is to be expected, Baseline-1 requires significantly more decoders than Baseline-2.

7.3 Scheduling for the PWD paradigm

In our simulations of the PWD paradigm, we first assume that the normalized decoder latency is fixed at $t_D = 0.5$. This is in line with existing high-performance hardware decoders [2, 73]. Further research will likely reduce this latency further. We assess the impact of t_D on the simulation results in subsequent sections. For all evaluations, we assume both temporal and spatial windows have the same size of $n_W = 3d$. We use the coarse-grained policy presented in Section 4 in all evaluations and compare the MFD, RR, and MLS fine-grained scheduling policies with each other.

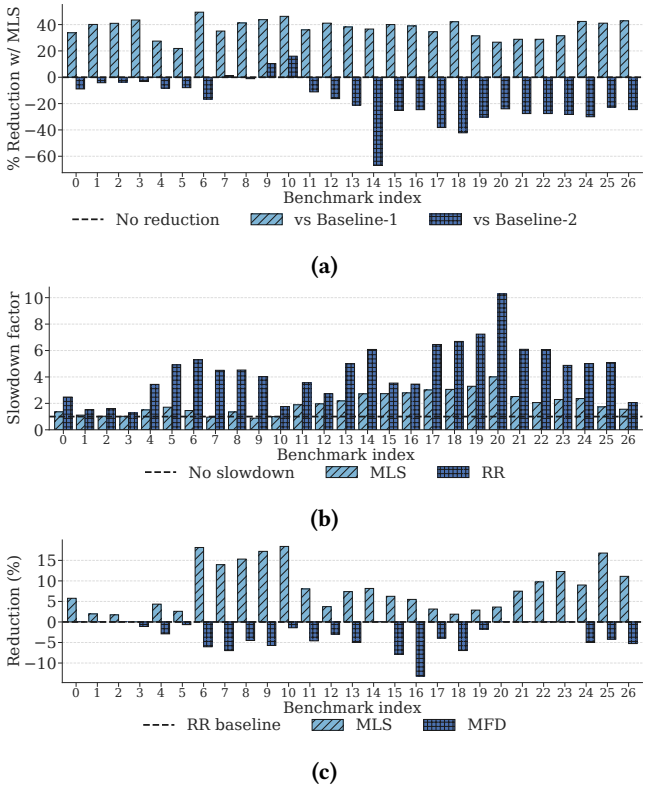


Figure 15. (a) The relative reduction (higher is better) in the number of decoders required by the MLS policy to prevent any slowdowns compared to the two baselines. Baseline-2 provisions for fewer decoders than what is required by the MLS policy; (b) Slowdown in computation when setting the number of decoders to Baseline-2 for the MLS and RR fine-grained policies; (c) The relative reduction in the minimum number of decoders required (higher is better) when using the MFD and MLS policies over the baseline RR policy. The MFD policy requires more decoders to prevent any slowdowns, while the MLS policy can reduce the number of required decoders by up to 19%;

7.3.1 Minimum number of decoders needed to prevent any slowdowns. We first determine the minimum number of decoders required for all fine-grained policies and benchmarks to prevent any slowdowns in the computation. We determine this optimal number by performing a binary search on the number of decoders such that the final backlog for all logical qubits is zero.

Figure 15(a) shows the reduction in the number of decoders required when the optimal number is determined for the MLS policy relative to the two baselines. Compared to Baseline-1, the optimal decoder count is about **40% lower** than the counts determined for Baseline-1 across most workloads. However, compared to Baseline-2, the optimal decoder count is greater by up to 60% – highlighting that the optimal number of decoders must be more than the average-case number of logical qubits active for any workload.

7.3.2 Baseline-2 introduces significant slowdowns. As Baseline-2 has fewer decoders allocated than the optimal number, it is expected that Baseline-2 will incur slowdowns in computation due to decoder scarcity, regardless of the scheduling policies used. Figure 15(b) shows the slowdowns incurred by Baseline-2 across all benchmarks when the MLS and RR policies are used. For the RR policy, **the slowdown can be up to 10×**, highlighting the sub-optimality of Baseline-2 compared to the optimal number of decoders. Unsurprisingly, the slowdowns are higher for benchmarks which have a greater difference between the optimal number of decoders and the number allocated for Baseline-2.

Slowdowns are an important factor to consider when designing fault-tolerant quantum computing systems. Despite the speed-up they offer, FTQC systems still require considerable time to finish computations. For example, it is estimated that factoring a 2048-bit number will take about a week [29]. Slowdowns must thus be avoided to further increase this computing time – a 2× slowdown would increase that time to two weeks. Figure 15(b) also highlights that the MLS policy is more resilient to decoder pressure/scarcity compared to the RR policy. Also, Baseline-1 cannot incur any slowdowns since the number of decoders allocated to it is always more than the optimal number determined across all benchmarks.

7.3.3 Comparing the fine-grained scheduling policies. To further illustrate the differences between the fine-grained scheduling policies, we determine the reduction in the optimal number of decoders when the MLS and RR policies are used compared to the baseline RR policy, as shown in Figure 15(c). The MLS policy achieves a relative reduction of **up to 19%** compared to the RR policy, while the MFD policy is either as good or worse than the RR policy. This is likely because the MFD policy starves some qubits of decoding, thus requiring more decoders. Combined with the results of Figure 15(b), these results clearly show the benefits of the MLS policy over the other fine-grained policies.

Another important observation from Figure 15(c) is its correlation with Figure 13 – benchmarks that exhibit higher concurrency experience greater benefits with the MLS policy compared to benchmarks that are more serial.

The answers for research questions $Q1 - Q3$ can be summarized from the results above as follows:

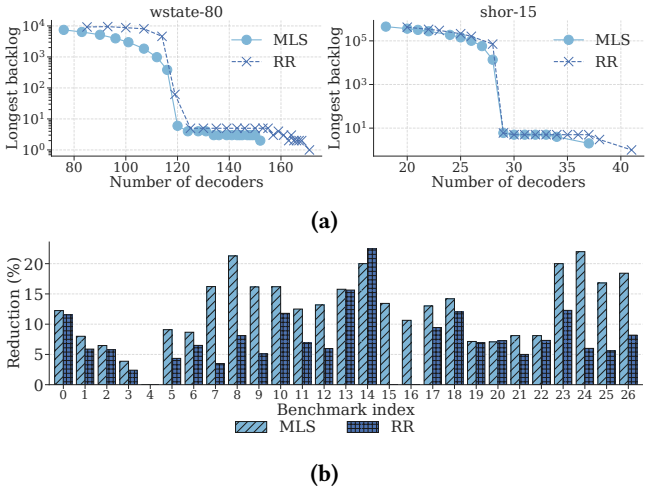
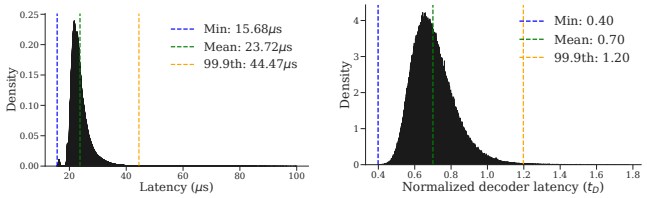


Figure 16. (a) Effect of the number of decoders on the backlog at the end of a benchmark for workloads exhibiting high concurrency (wstate-80) and low concurrency (shor-15). When the number of decoders is less than a threshold, the backlog grows exponentially; (b) Reduction in the number of decoders relative to the optimal number determined in Figure 15(a) for the MLS and RR policies when a slowdown of 10% or less is tolerable. The MLS policy yields a higher reduction for most workloads.

> The optimal number of decoders is 40% less than the worst-case decoder allocation of Baseline-1 while avoiding the slowdown of up to 10× experienced by Baseline-2. The MLS policy outperforms both RR and MFD policies by up to 19%.

7.3.4 Relaxing the slowdown constraint. The optimal decoder count was determined to ensure that there would be no slowdown during the execution of the program. However, what if this slowdown constraint was relaxed? To understand the effect of relaxing the slowdown constraint, we first show the effect of reducing the number of decoders on the longest backlog of syndromes for two benchmarks: the first being a highly concurrent (wstate-80) and the second being very serial (shor-15). Figure 16(a) shows the effect of reducing the number of decoders on the longest backlog of syndromes for both benchmarks: after a certain decoder count, the backlog increases exponentially, corresponding to significant slowdowns (since these backlogs would need extra time to be processed). This exponential increase would also increase the memory requirements to store undecoded syndromes. Additionally, as was shown before, the MLS policy is more resilient to the scarcity of decoders compared to the RR policy. We omit the MFD policy from these evaluations for brevity, since we showed in Figure 15 that it is inferior to both MLS and RR policies.



(a) (b)

Figure 17. (a) Density histogram of latencies sampled from the PyMatching [34] decoder: the distribution can be approximated by a log-normal distribution with long tail latencies ($d = 9, p = 0.1\%$); (b) The density histogram used for our simulations with t_D sampled from a log-normal distribution.

In Figure 16(b), we determine the reduction in the number of decoders relative to the optimal number determined in Figure 15(a) when slowdowns up to 10% are tolerable. For all benchmarks, we see a reduction in the optimal number of decoders required, especially for `wstate`. Furthermore, the `MLS` policy achieves a higher reduction than the `RR` policy. These results show that there is a trade-off between decoder allocation and program performance: if limited slowdowns are tolerable, the classical resource requirements can be significantly reduced with good scheduling policies.

7.4 Effect of decoder tail latencies

So far, we have performed all evaluations with a fixed normalized decoder latency $t_D = 0.5$ (where $t_D = \frac{T_{decode}}{T_{cycle}}$). However, decoders do not have a deterministic latency, and some decoding problems take longer to decode than others. In particular, some decoding tasks can have a long *tail latency*. This can result in higher decoder allocations, because slower decoders are unavailable for a longer period, reducing the throughput at which decoding tasks as processed. Figure 17(a) shows the distribution of latencies from the state-of-the-art software decoder for the surface code, PyMatching [34] for a distance 9 ($d = 9$) surface code logical qubit and a circuit-level physical error rate of 0.1% (p). The distribution is centered around the mean but tail latencies are significant long. We approximate this distribution with a log-normal probability density function [20] of the form:

$$PDF(x; \mu, \sigma) = \frac{1}{x\,\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right), \quad x > 0$$

We model hardware decoder latencies with a log-normal distribution (see Figure 17(b)), which produces worst-case tail latencies longer than the syndrome generation cycle ($t_D > 1.0$). These long tails shift the optimal decoder count for all benchmarks compared with Figure 15(a): most benchmarks require more decoders under the `MLS` policy, while `qft-20 (14), qft-80 (17)` require fewer because some critical decodes sampled latencies below $t_d = 0.5$, reducing

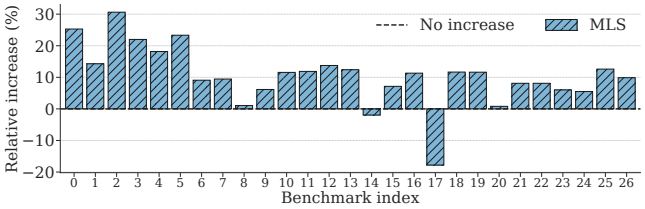


Figure 18. Relative increase in the number of optimal decoders required by the `MLS` policy to prevent slowdowns with longer non-uniform decoder latencies.

their decoder needs. **Overall, despite these longer, non-uniform latencies, we can still reduce decoder counts by 10–40% across all benchmarks**.

7.5 Limitations

Magic states: We assume that magic states are produced deterministically and are always available. This represents the most resource-intensive scenario, since it maximizes the potential number of concurrent non-Clifford operations and thus critical decodes. In practice, fewer distillation factories or variable throughput would reduce concurrency and lower decoder demand. We do not evaluate these effects, so these results are an upper bound on decoder requirements.

Fine-grained scheduling policies: Our results show that the `MLS` policy is more effective than `RR` at reducing backlogs. However, both `MLS` and `MFD` require continuous ranking of logical qubits, which may incur non-trivial overhead in a hardware controller. While ranking can be pre-computed for statically compiled programs under deterministic magic state availability, this assumption does not hold in general. By contrast, `RR` is simple to implement and has predictable overhead. We view our evaluation as a first step, and expect future work to explore more practical, low-overhead policies that combine the efficiency of `MLS` with the simplicity of `RR`.

7.6 Real system evaluation

At present, no publicly available quantum computer supports the tight classical control loop required for evaluating the ideas presented in this work. This control loop requires fast transmission of syndromes from the quantum processor to the decoding hardware, an ability to route syndromes to specific decoders in the system, and fast transmission of the decoder result to the classical control software. Furthermore, the smallest workload in our study requires 20 logical qubits. Even if we make the most optimistic assumption that one logical qubit is encoded in a distance-3 ($d = 3$) surface code patch, 20 logical qubits will require at least $20 \times 17 = 340$ physical qubits. Overheads due to magic state production, routing, ancillas will further inflate the qubit requirements. This is far beyond the capacity of current, cloud-accessible quantum computers. This limitation on the number of qubits

also prevents offline studies where syndromes can be collected from a system and then post-processed.

8 Related Work

This is one of the first works to perform a workload-oriented study of the classical processing requirements and system-level scheduling policies for error-corrected quantum computers. The Round-Robin (RR) decoder scheduling policy was alluded to in [11, 17] for allocating decoders to different logical qubits but the efficacy of the policy and its strengths and limitations were not explored. In particular, XQ-Sim [11] proposed *patch-sliding window decoding* that reduced resource and power requirements for a cryogenic control system . This patch-sliding mechanism is essentially the round-robin scheduling policy. Similarly, AFS [17] reduced decoder resource requirements by using the conjoined decoder architecture (CDA) that shared decoder compute blocks among multiple logical qubits, with arbitration done via a round-robin policy.

Spatial and temporal parallel window decoding are discussed in [10, 25, 59]. [59] introduced parallel windowed decoding in both space and time and showed how sliding windowed decoding is not scalable. [25] further refined and discussed microarchitectural considerations for spatial windowed decoding, including scheduling different decoding windows within a decoding task. Bombín et al. [10] introduced modular decoding, which incorporates both spatial and temporal parallelism in decoding without sacrificing accuracy. No work focuses on scheduling decoders from a windowed decoding approach. Other works that are broadly connected to this work are summarized below.

System-level Studies: Network integrated FPGA-based decoding systems were introduced in [43, 71]. Delfosse et al. [20] studied the speed vs. accuracy tradeoff for decoders used in FTQC. Stein et al. [61] proposed a heterogeneous architecture for FTQC. Lin et al. [39] explored modular architectures for error-correcting codes, and scheduling for distillation factories was proposed in [22]. [38] described a blueprint of a fault-tolerant quantum computer.

Decoder Designs: Neural network-based decoders [1, 4, 7, 26, 46, 50, 66, 67], LUT-based decoders [16, 64], decoders based on the union-find algorithm [5, 73], and optimized MWPM decoders [2, 69, 72] have been proposed. In general, neural network decoders are slower and therefore not ideal for fast qubit technologies such as superconducting qubits. Other predecoders [19, 60] and partial decoders [12] have also been proposed. Decoders based on superconducting logic [52, 65] target cryogenic implementations.

9 Conclusions

Decoders form the backbone of fault-tolerant quantum computing (FTQC) systems. They are classical algorithms designed to continuously detect and correct errors in the quantum hardware, and must be accelerated using custom/reconfigurable hardware to meet microsecond-scale latency requirements. For such hardware decoders, *capacity planning* and *scheduling* are key to ensure optimal resource utilization and performance. Capacity planning prevents wasteful over-provisioning or risky under-provisioning, while scheduling ensures that decoders are assigned tasks such that there is no degradation to program runtime. Capacity planning for hardware decoders is not trivial – the use of spatial and temporal parallel windowed decoding (PWD) results in non-deterministic decoder demand throughout the course of a program's execution. This paper is the first to address these system-level challenges for FTQC. We introduce a two-level scheduling framework that enables accurate decoder provisioning and efficient task allocation across diverse workloads. Our evaluation shows that this approach reduces decoder hardware needs by 10–40% while sustaining fault-tolerant performance.

A Artifact Appendix

A.1 Abstract

This artifact contains program traces, benchmarks, and software to generate the main results and insights presented in the paper. Lattice Surgery Compiler [70] has been used to generate program traces, and while the artifact provides the traces for ease of evaluation, the traces can be generated using instructions in the README.

This artifact can be used to reproduce Figures 6, 7, 12, 13, 14, 15(a), 15(b), 15(c), and 18 of the main text. Note that due to the inherent randomness of the sampling process used for generating decoder latencies, Figure 18 might not exactly match the generated results.

A.2 Description & Requirements

A.2.1 How to access. We provide the code and datasets (program traces) to reproduce the results of this paper as a Zenodo repository: https://doi.org/10.5281/zenodo.18555903.

The code for generating program traces and scheduling decoders is also available on Github: https://github.com/satvikmaurya/decoder-resources.

A.2.2 Hardware dependencies. ≥16-core workstation/server node (any Linux distro), ≥64 GB RAM[2], and ≥20 GB storage.

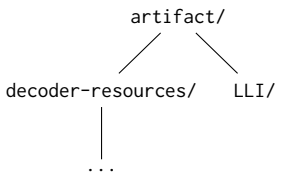
A.2.3 Software dependencies. Docker.

[2]Our evaluations on a 32C/64T machine did not require more than 96 GB of RAM.

A.2.4 Benchmarks. None (all benchmarks and program traces included in artifact).

A.3 Set-up

This artifact utilizes Docker for a seamless installation of all underlying (C++, Python, Haskell) dependencies. To start off, ensure that the directory structure looks something similar to this (assuming `artifact` is the top-level directory):



To build the Docker image, run:

```
$ cd decoder-resources/
$ docker build -t $USER/vader .
```

We recommend using the interactive shell of the Docker container for this artifact. To prevent loss of progress, `tmux` is recommended:

```
$ tmux new -s artifact129
$ docker compose run app
```

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): We show that the demand for decoders in fault-tolerant programs is bursty and non-deterministic.*
- *(C2): We show that with careful capacity planning, the number of decoders can be reduced by up to 40% compared to the worst-case provisioning, saving system costs and complexity.*
- *(C3): We show that the MLS fine-grained scheduling policy under this bursty, non-deterministic achieves a 19% reduction in the number of decoders compared to the baseline scheduling policy.*

A.4.2 Experiments.

Experiment (E1): [End-to-end evaluation] [30 human-minutes + 24 compute-hours]: To simplify evaluations, we have consolidated all scripts into a single bash script that can be run from within the Docker container. We provide descriptions of some important constituent scripts below:

1. `histogram.sh`: This script characterizes the patch-to-patch distances between logical qubits involved in an operation. This generates data for Figure 6, which supports *(C1)*.
2. `spatial_wc.py`: This script characterizes the worst-case number of concurrently active patches as a function of time for all workloads. This generates data for Figure 7, which supports *(C1)*.
3. `baseline.sh`: This script determines the slowdown when the average-case baseline (Baseline-2) is used for provisioning decoders. This generates data for Figure 15(b). This script supports *(C2)*.
4. `optimize.sh`: This script runs an optimization loop to determine the optimal (minimum) number of decoders required for a workload and scheduling policy to prevent any program slowdowns. This generates data for Figures 15(a), 15(c), and 18. This script supports *(C2 and C3)*.

[How to]
To run all required scripts, simply run the `run_all.sh` script from the `scripts/` directory:

```
$ cd scripts/
$ bash run_all.sh
```

The total expected runtime is at most 48 hours. Each script will generate results in a pickled format, which are then used by the plotting script for generating figures.
[Results]
All generated figures will be available in the `artifact/decoder-resources/figs/` directory.

A.5 Notes on Reusability

This artifact provides pre-generated program traces of various quantum computing workloads to analyze decoder provisioning and scheduling. However, the code provided can be used to generate program traces of other workloads too. Furthermore, any other surface code compiler can be used for generating the program traces required by this artifact, provided the same IR is used for defining logical operations. A change in IR will require significant refactoring of the parsing components of this artifact, but the scheduling logic and setup should generalize.

A.6 Methodology

Submission, reviewing, and badging methodology:

- http://ctuning.org/ae/submission-20201122.html
- http://ctuning.org/ae/reviewing-20201122.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] Rajeev Acharya, Laleh Aghababaie-Beni, Igor Aleiner, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, Ryan Babbush, Dave Bacon, Brian Ballard, Joseph C. Bardin, Johannes Bausch, Andreas Bengtsson, Alexander Bilmes, Sam Blackwell, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, David A. Browne, Brett Buchea, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Anthony Cabrera, Juan Campero, Hung-Shen Chang, Yu Chen, Zijun Chen, Ben Chiaro, Desmond Chik, Charina Chou, Jahan Claes, Agnetta Y. Cleland, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben

Curtin, Sayan Das, Alex Davies, Laura De Lorenzo, Dripto M. Debroy, Sean Demura, Michel Devoret, Agustin Di Paolo, Paul Donohoe, Ilya Drozdov, Andrew Dunsworth, Clint Earle, Thomas Edlich, Alec Eickbusch, Aviv Moshe Elbag, Mahmoud Elzouka, Catherine Erickson, Lara Faoro, Edward Farhi, Vinicius S. Ferreira, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, Suhas Ganjam, Gonzalo Garcia, Robert Gasca, Elie Genois, William Giang, Craig Gidney, Dar Gilboa, Raja Gosula, Alejandro Grajales Dau, Dietrich Graumann, Alex Greene, Jonathan A. Gross, Steve Habegger, John Hall, Michael C. Hamilton, Monica Hansen, Matthew P. Harrigan, Sean D. Harrington, Francisco J. H. Heras, Stephen Heslin, Paula Heu, Oscar Higgott, Gordon Hill, Jeremy Hilton, George Holland, Sabrina Hong, Hsin-Yuan Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Stephen Jordan, Chaitali Joshi, Pavol Juhas, Dvir Kafri, Hui Kang, Amir H. Karamlou, Kostyantyn Kechedzhi, Julian Kelly, Trupti Khaire, Tanuj Khattar, Mostafa Khezri, Seon Kim, Paul V. Klimov, Andrey R. Klots, Bryce Kobrin, Pushmeet Kohli, Alexander N. Korotkov, Fedor Kostritsa, Robin Kothari, Borislav Kozlovskii, John Mark Kreikebaum, Vladislav D. Kurilovich, Nathan Lacroix, David Landhuis, Tiano Lange-Dei, Brandon W. Langley, Pavel Laptev, Kim-Ming Lau, Loïck Le Guevel, Justin Ledford, Kenny Lee, Yuri D. Lensky, Shannon Leon, Brian J. Lester, Wing Yan Li, Yin Li, Alexander T. Lill, Wayne Liu, William P. Livingston, Aditya Locharla, Erik Lucero, Daniel Lundahl, Aaron Lunt, Sid Madhuk, Fionn D. Malone, Ashley Maloney, Salvatore Mandrá, Leigh S. Martin, Steven Martin, Orion Martin, Cameron Maxfield, Jarrod R. McClean, Matt McEwen, Seneca Meeks, Anthony Megrant, Xiao Mi, Kevin C. Miao, Amanda Mieszala, Reza Molavi, Sebastian Molina, Shirin Montazeri, Alexis Morvan, Ramis Movassagh, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Chia-Hung Ni, Thomas E. O'Brien, William D. Oliver, Alex Opremcak, Kristoffer Ottosson, Andre Petukhov, Alex Pizzuto, John Platt, Rebecca Potter, Orion Pritchard, Leonid P. Pryadko, Chris Quintana, Ganesh Ramachandran, Matthew J. Reagor, David M. Rhodes, Gabrielle Roberts, Eliott Rosenberg, Emma Rosenfeld, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus, Christopher Schuster, Andrew W. Senior, Michael J. Shearn, Aaron Shorter, Noah Shutty, Vladimir Shvarts, Shraddha Singh, Volodymyr Sivak, Jindra Skruzny, Spencer Small, Vadim Smelyanskiy, W. Clarke Smith, Rolando D. Somma, Sofia Springer, George Sterling, Doug Strain, Jordan Suchard, Aaron Szasz, Alex Sztein, Douglas Thor, Alfredo Torres, M. Mert Torunbalci, Abeer Vaishnav, Justin Vargas, Sergey Vdovichev, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Steven Waltman, Shannon X. Wang, Brayden Ware, Kate Weber, Theodore White, Kristi Wong, Bryan W. K. Woo, Cheng Xing, Z. Jamie Yao, Ping Yeh, Bicheng Ying, Juhwan Yoo, Noureldin Yosri, Grayson Young, Adam Zalcman, Yaxing Zhang, Ningfeng Zhu, and Nicholas Zobrist. 2024. Quantum error correction below the surface code threshold. doi:10.48550/ARXIV.2408.13687

[2] 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. doi:10.48550/ARXIV.2404.03136

[3] Alan Aspuru-Guzik, Anthony D. Dutoi, Peter J. Love, and Martin Head-Gordon. 2005. Simulated Quantum Computation of Molecular Energies. *Science* 309, 5741 (Sept. 2005), 1704–1707. doi:10.1126/science.1113479

[4] J. Pablo Bonilla Ataides, Andi Gu, Susanne F. Yelin, and Mikhail D. Lukin. 2025. Neural Decoders for Universal Quantum Algorithms. doi:10.48550/ARXIV.2509.11370

[5] 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. doi:10.48550/ARXIV.2309.05558

[6] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. 2011. *Mixed-Criticality Scheduling of Sporadic Task Systems.* Springer Berlin Heidelberg, 555–566. doi:10.1007/978-3-642-23719-5_47

[7] Johannes Bausch, Andrew W Senior, Francisco J H 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 Neven, and Pushmeet Kohli. 2023. Learning to Decode the Surface Code with a Recurrent, Transformer-Based Neural Network. doi:10.48550/ARXIV.2310.05900

[8] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. 2022. Surface Code Compilation via Edge-Disjoint Paths. *PRX Quantum* 3, 2 (May 2022). doi:10.1103/prxquantum.3.020342

[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. doi:10.48550/ARXIV.2211.07629

[10] Héctor Bombín, Chris Dawson, Ye-Hua Liu, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. 2023. Modular decoding: parallelizable real-time decoding for quantum computers. doi:10.48550/ARXIV.2303.04846

[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 (ISCA '22)*. ACM. doi:10.1145/3470496.3527417

[12] Laura Caune, Brendan Reid, Joan Camps, and Earl Campbell. 2023. Belief propagation as a partial decoder. doi:10.48550/ARXIV.2306.17142

[13] Christopher Chamberland and Kyungjoo Noh. 2020. Very low overhead fault-tolerant magic state preparation using redundant ancilla encoding and flag qubits. *npj Quantum Information* 6, 1 (Oct. 2020). doi:10.1038/s41534-020-00319-5

[14] Henry Corrigan-Gibbs, David J. Wu, and Dan Boneh. 2017. Quantum Operating Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, 76–81. doi:10.1145/3102980.3102993

[15] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. doi:10.48550/ARXIV.1707.03429

[16] 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 (ASPLOS '22)*. ACM. doi:10.1145/3503222.3507707

[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. doi:10.1109/hpca53966.2022.00027

[18] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *Comput. Surveys* 43, 4 (Oct. 2011), 1–44. doi:10.1145/1978802.1978814

[19] Nicolas Delfosse. 2020. Hierarchical decoding to reduce hardware requirements for quantum computing. doi:10.48550/ARXIV.2001.11427

[20] Nicolas Delfosse, Andres Paz, Alexander Vaschillo, and Krysta M. Svore. 2023. How to choose a decoder for a fault-tolerant quantum computer? The speed vs accuracy trade-off. doi:10.48550/ARXIV.2310.15313

[21] Antonis Dimakis and Jean Walrand. 2006. Sufficient conditions for stability of longest-queue-first scheduling: Second-order properties using fluid limits. *Advances in Applied probability* 38, 2 (2006), 505–521.

[22] Yongshan Ding, Adam Holmes, Ali Javadi-Abhari, Diana Franklin, Margaret Martonosi, and Frederic Chong. 2018. Magic-State Functional Units: Mapping and Scheduling Multi-Level Distillation Circuits for Fault-Tolerant Quantum Architectures. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. doi:10.1109/micro.2018.00072

[23] Austin G. Fowler and Craig Gidney. 2018. Low overhead quantum computation using lattice surgery. doi:10.48550/ARXIV.1808.06709

[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* 86, 3 (Sept. 2012). doi:10.1103/physreva.86.032324

[25] 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 (April 2025), 035007. doi:10.1088/2058-9565/adc6b6

[26] Spiro Gicev, Lloyd C. L. Hollenberg, and Muhammad Usman. 2023. A scalable and fast artificial neural network syndrome decoder for surface codes. *Quantum* 7 (July 2023), 1058. doi:10.22331/q-2023-07-12-1058

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

[28] Craig Gidney. 2024. Inplace Access to the Surface Code Y Basis. *Quantum* 8 (April 2024), 1310. doi:10.22331/q-2024-04-08-1310

[29] Craig Gidney. 2025. How to factor 2048 bit RSA integers with less than a million noisy qubits. *arXiv preprint arXiv:2505.15917* (2025).

[30] Craig Gidney, Noah Shutty, and Cody Jones. 2024. Magic state cultivation: growing T states as cheap as CNOT gates. (2024). doi:10.48550/ARXIV.2409.17595

[31] Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia. 2025. {QOS}: Quantum Operating System. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 429–447.

[32] Google Quantum AI. 2024. *Google Quantum Roadmap*. https://quantumai.google/qecmilestone Accessed: September 18, 2025.

[33] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. doi:10.1145/237814.237866

[34] Oscar Higgott and Craig Gidney. 2023. Sparse Blossom: correcting a million errors per core second with minimum-weight matching. doi:10.48550/ARXIV.2303.15933

[35] 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 (Dec. 2012), 123011. doi:10.1088/1367-2630/14/12/123011

[36] IBM Quantum. 2024. *IBM Quantum Roadmap*. https://www.ibm.com/quantum/roadmap Accessed: September 18, 2025.

[37] Pavithran Iyer and David Poulin. 2015. Hardness of decoding quantum stabilizer codes. *IEEE Transactions on Information Theory* 61, 9 (2015), 5209–5223.

[38] Junpyo Kim, Dongmoon Min, Jungmin Cho, Hyeonseong Jeong, Ilkwon Byun, Junhyuk Choi, Juwon Hong, and Jangwoo Kin. 2024. A Fault-Tolerant Million Qubit-Scale Distributed Quantum Computer. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. doi:10.1145/3620665.3640388

[39] Sophia Fuhui Lin, Joshua Viszlai, Kaitlin N. Smith, Gokul Subramanian Ravi, Charles Yuan, Frederic T. Chong, and Benjamin J. Brown. 2023. Codesign of quantum error-correcting codes and modular chiplets in the presence of defects. doi:10.48550/ARXIV.2305.00138

[40] Daniel Litinski. 2019. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum* 3 (March 2019), 128. doi:10.22331/q-2019-03-05-128

[41] Daniel Litinski. 2019. Magic State Distillation: Not as Costly as You Think. *Quantum* 3 (Dec. 2019), 205. doi:10.22331/q-2019-12-02-205

[42] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[43] Namitha Liyanage, Yue Wu, Emmet Houghton, and Lin Zhong. 2025. Network-Integrated Decoding System for Real-Time Quantum Error Correction with Lattice Surgery. *arXiv preprint arXiv:2504.11805* (2025).

[44] Seth Lloyd. 1996. Universal Quantum Simulators. *Science* 273, 5278 (Aug. 1996), 1073–1078. doi:10.1126/science.273.5278.1073

[45] Lukasz Masko, Pierre–Francois Dutot, Gregory Mounie, Denis Trystram, and Marek Tudruj. 2006. *Scheduling Moldable Tasks for Dynamic SMP Clusters in SoC Technology*. Springer Berlin Heidelberg, 879–887. doi:10.1007/11752578_106

[46] Kai Meinerz, Chae-Yeun Park, and Simon Trebst. 2022. Scalable Neural Decoder for Topological Surface Codes. *Physical Review Letters* 128, 8 (Feb. 2022). doi:10.1103/physrevlett.128.080505

[47] Tristan Müller, Thomas Alexander, Michael E Beverland, Markus Bühler, Blake R Johnson, Thilo Maurer, and Drew Vandeth. 2025. Improved belief propagation is sufficient for real-time decoding of quantum memory. *arXiv preprint arXiv:2506.01779* (2025).

[48] NVIDIA. 2025. *NVIDIA and QuEra Decode Quantum Errors with AI*. NVIDIA. https://developer.nvidia.com/blog/nvidia-and-quera-decode-quantum-errors-with-ai/ Accessed: June 18, 2025.

[49] NVIDIA. 2025. Streamlining Quantum Error Correction and Application Development with CUDA-QX 0.4. https://developer.nvidia.com/blog/streamlining-quantum-error-correction-and-application-development-with-cuda-qx-0.4/. GPU-accelerated tensor-network decoders, BP+OSD enhancements, automated detector error models.

[50] Ramon W. J. Overwater, Masoud Babaie, and Fabio Sebastiano. 2022. Neural-Network Decoders for Quantum Error Correction Using Surface Codes: A Space Exploration of the Hardware Cost-Performance Tradeoffs. *IEEE Transactions on Quantum Engineering* 3 (2022), 1–19. doi:10.1109/tqe.2022.3174017

[51] Quantinuum. 2025. *Quantinuum Unveils Accelerated Roadmap to Achieve Universal, Fully Fault-Tolerant Quantum Computing by 2030*. https://www.quantinuum.com/press-releases/quantinuum-unveils-accelerated-roadmap-to-achieve-universal-fault-tolerant-quantum-computing-by-2030 Accessed: September 18, 2025.

[52] 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 (ASPLOS '23)*. ACM. doi:10.1145/3575693.3575733

[53] Salonik Resch and Ulya R. Karpuzcu. 2021. Benchmarking Quantum Computers and the Impact of Quantum Noise. *Comput. Surveys* 54, 7 (July 2021), 1–35. doi:10.1145/3464420

[54] Riverlane. 2025. Deltaflow: The Quantum Error Correction Stack. https://www.riverlane.com/quantum-error-correction-stack. Real-time error correction for up to 250 qubits using FPGA/ASIC decode hardware.

[55] Neil J. Ross and Peter Selinger. 2014. Optimal ancilla-free Clifford+T approximation of z-rotations. doi:10.48550/ARXIV.1403.2975

[56] Satvik Maurya and Abtin Molavi and Aws Albarghouthi and Swamit Tannu. 2026. *Decoder Resource Estimator*. https://doi.org/10.5281/zenodo.18555904 Available at https://doi.org/10.5281/zenodo.18555904.

[57] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (oct 1997), 1484–1509. doi:10.1137/S0097539795293172

[58] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 4, 3 (1996), 375–385.

[59] 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 (Nov. 2023). doi:10.1038/s41467-023-42482-1

[60] Samuel C. Smith, Benjamin J. Brown, and Stephen D. Bartlett. 2023. Local Predecoder to Reduce the Bandwidth and Latency of Quantum Error Correction. *Physical Review Applied* 19, 3 (March 2023). doi:10.1103/physrevapplied.19.034050

[61] Samuel Stein, Sara Sussman, Teague Tomesh, Charles Guinn, Esin Tureci, Sophia Fuhui Lin, Wei Tang, James Ang, Srivatsan Chakram, Ang Li, Margaret Martonosi, Fred Chong, Andrew A. Houck, Isaac L. Chuang, and Michael Demarco. 2023. HetArch: Heterogeneous Microarchitectures for Superconducting Quantum Systems. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. ACM. doi:10.1145/3613424.3614300

[62] Leandros Tassiulas and Anthony Ephremides. 1990. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *29th IEEE Conference on Decision and Control*. IEEE, 2130–2132.

[63] Barbara M. Terhal. 2015. Quantum error correction for quantum memories. *Reviews of Modern Physics* 87, 2 (April 2015), 307–346. doi:10.1103/revmodphys.87.307

[64] Yu Tomita and Krysta M. Svore. 2014. Low-distance surface codes under realistic quantum noise. *Physical Review A* 90, 6 (Dec. 2014). doi:10.1103/physreva.90.062320

[65] 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. doi:10.1109/dac18074.2021.9586326

[66] Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki, and Yutaka Tabuchi. 2022. NEO-QEC: Neural Network Enhanced Online Superconducting Decoder for Surface Codes. doi:10.48550/ARXIV.2208.05758

[67] Savvas Varsamopoulos, Ben Criger, and Koen Bertels. 2017. Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology* 3, 1 (Nov. 2017), 015004. doi:10.1088/2058-9565/aa955a

[68] Steve Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, 239–243. doi:10.1109/rtss.2007.47

[69] 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 (ISCA '23)*. ACM. doi:10.1145/3579371.3589037

[70] George Watkins, Hoang Minh Nguyen, Keelan Watkins, Steven Pearce, Hoi-Kwan Lau, and Alexandru Paler. 2024. A High Performance Compiler for Very Large Scale Surface Code Computations. *Quantum* 8 (May 2024), 1354. doi:10.22331/q-2024-05-22-1354

[71] Yue Wu, Namitha Liyanage, and Lin Zhong. 2024. LEGO: QEC Decoding System Architecture for Dynamic Circuits. *arXiv preprint arXiv:2410.03073* (2024).

[72] Yue Wu, Namitha Liyanage, and Lin Zhong. 2025. Micro Blossom: Accelerated Minimum-Weight Perfect Matching Decoding for Quantum Error Correction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. ACM, 639–654. doi:10.1145/3676641.3716005

[73] Abbas B. Ziad, Ankit Zalawadiya, Canberk Topal, Joan Camps, György P. Gehér, Matthew P. Stafford, and Mark L. Turner. 2024. Local Clustering Decoder: a fast and adaptive hardware decoder for the surface code. doi:10.48550/ARXIV.2411.10343