Astrea: Accurate Quantum Error-Decoding via Practical Minimum-Weight Perfect-Matching

Suhas Vittal
suhaskvittal@gatech.edu
Georgia Tech
Atlanta, USA

Poulami Das
poulami@gatech.edu
Georgia Tech
Atlanta, USA

Moin Qureshi
moin@gatech.edu
Georgia Tech
Atlanta, USA

ABSTRACT

Quantum devices suffer from high error rates, which makes them ineffective for running practical applications. Quantum computers can be made fault tolerant using *Quantum Error Correction* (QEC), which protects quantum information by encoding logical qubits using data qubits and parity qubits. The data qubits collectively store the quantum information and the parity qubits are measured periodically to produce a syndrome, which is decoded by a classical *decoder* to identify the location and type of errors. To prevent errors from accumulating and causing a logical error, decoders must accurately identify errors in real-time, necessitating the use of hardware solutions because software decoders are slow. Ideally, a real-time decoder must match the performance of the *Minimum-Weight Perfect Matching* (MWPM) decoder. However, due to the complexity of the underlying Blossom algorithm, state-of-the-art real-time decoders either use lookup tables, which are not scalable, or use approximate decoding, which significantly increases logical error rates.

In this paper, we leverage two key insights to enable practical real-time MWPM decoding. First, for near-term implementations (with redundancies up to distance $d = 7$) of surface codes, the Hamming weight of the syndromes tends to be quite small (less than or equal to 10). For this regime, we propose *Astrea*, which simply performs a brute-force search for the few hundred possible options to perform accurate decoding within a few nanoseconds (1ns average, 456ns worst-case), thus representing the first decoder to be able to do MWPM in real-time up-to distance 7. Second, even for codes that produce syndromes with higher Hamming weights (e.g. $d = 9$) the search for optimal pairings can be made more efficient by simply discarding the weights that denote significantly lower probability than the logical error-rate of the code. For this regime, we propose a greedy design called *Astrea-G*, which filters high-cost weights and reorders the search from high-likelihood pairings to low-likelihood pairings to produce the most likely decoding within 1μs (average 450ns). Our evaluations show that Astrea-G provides similar logical error-rates as the software-based MWPM for up-to $d = 9$ codes while meeting the real-time decoding latency constraints.

CCS CONCEPTS

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

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589037

KEYWORDS

Quantum error correction, Error decoding, Real-time decoding

ACM Reference Format:
Suhas Vittal, Poulami Das, and Moin 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), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3579371.3589037

1 INTRODUCTION

Quantum computers promise significant speedup over conventional computing systems for many important application domains, such as quantum chemistry and crypto-analysis [8, 12, 20, 25, 29, 30, 37, 42, 43, 47, 49, 53]. However, the high error rates of the quantum devices limit us from running most practical quantum applications. *Quantum Error Correction (QEC)* enables fault-tolerant quantum computations by encoding a *logical qubit* using several physical qubits [5, 10, 11, 17, 22, 28, 36, 41, 52]. If the physical error-rate is lower than a certain threshold, then the error-rate of a logical qubit, or the *logical error rate* decreases exponentially with increasing redundancy or *distance (d)* of the QEC code. With increasing system sizes and improving device qualities, there are growing interests in enabling QEC codes on real quantum hardware. For example, Google recently demonstrated exponential suppression of errors using a limited form of $d = 5$ code [1]. Scaling QEC codes and fault-tolerant operations for larger distances ($d = 7$ and $d = 9$) are significant milestones for quantum computing in the near term.

This paper focuses on *surface code*, which is widely regarded as the most promising QEC code due to its high threshold and simple grid structure [17, 22, 36]. The surface code encodes a logical qubit using an alternating lattice of *data* and *parity* qubits. The data qubits collectively store the quantum state, whereas the parity qubits detect errors on the data qubits. During program execution, QEC operations are interleaved between the program instructions. As shown in Figure 1(a), the control processor sends QEC instructions to perform *syndrome extraction*, a process during which parity qubits interact with their neighboring data qubits to extract information about errors. Errors result in failed parity checks which are obtained when the parity qubits are measured to generate a *syndrome*. The syndrome, which is a bitstring of 0s and 1s, is sent to a classical *decoder* that uses it to identify errors on the data qubits. Finally, the decoder sends a *logical correction* to reverse the errors that have occurred on the data qubits and the control processor updates its future operations.

The logical error rate depends on the physical error rate and the decoder performance [22]. Decoders must decode syndromes in

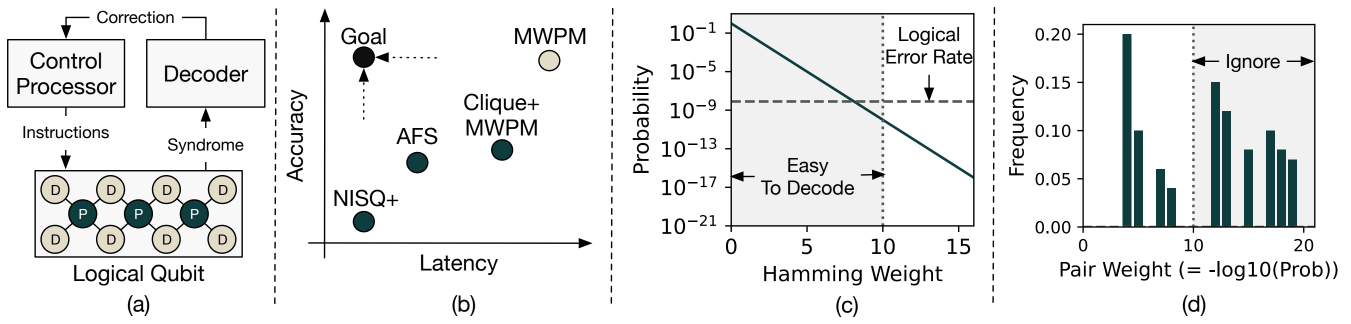

Figure 1: (a) Overview of quantum error correction (b) Trade-off between the accuracy and latency of various decoders (c) The first insight (used in Astrea) is that syndromes with low hamming-weights can be decoded easily in hardware and high hamming-weight syndromes with probability higher than the logical error rates can be ignored without impacting accuracy. (d) The second insight (used in Astrea-G) is that not all pairings are created equal – pairings whose weight denotes a significantly lower probability than the logical error rate can be ignored, leading to a significantly more efficient search.

real-time, typically within *a microsecond* on existing quantum computers, such as Google Sycamore [3], to prevent the accumulation of errors. Moreover, syndromes are imperfect due to operational errors that occur during syndrome extraction. To tolerate these errors and assess errors more accurately, decoders must decode d syndrome rounds simultaneously for a code of distance d in real-time. The decoding problem can be reduced to a *graph matching* problem, where nonzero syndrome bits across d syndrome extraction rounds are uniquely paired with each other to create a *matching*. Each pair of syndrome bits have a *weight* that corresponds to the probability that errors will cause the associated parity checks to fail: a lower weight indicates a higher probability. Thus, pairing up syndrome bits to find a *Minimum-Weight Perfect Matching* (MWPM) corresponds to finding the *highest probability error event* given d syndromes. The MWPM decoder is the state-of-the-art decoder for surface codes owing to its high accuracy, as computing the MWPM guarantees the highest probability error event [17, 22]. *Ideally, we want to achieve the accuracy of MWPM in real-time.*

Unfortunately, software MWPM decoders [31, 32, 38] are too slow for real-time decoding because the underlying Blossom algorithm for obtaining the MWPM is quite complex [18, 19]. Note that in superconducting quantum systems, such as Google Sycamore, the qubits reside at few milli-Kelvins inside a dilution refrigerator. Transmitting the syndromes to general-purpose processors outside the cryogenic setup, running the software MWPM decoder, and then sending the logical correction back to the control processor is too constrained to be done within a few microseconds. To meet this latency constraint, recent works (e.g. AFS [14]) sacrifice accuracy by using approximate algorithms. However, this increases the logical error rate compared to MWPM (almost 200× for AFS), especially in the regime of physical error rates of 10^{-3} to 10^{-4}, which represents the target qubit quality in the next few years. LILLIPUT [13] uses lookup-tables for decoding, but the memory overheads of this design grow significantly, causing it to be limited to $d = 5$ with only two rounds of syndromes, thus making it impractical for larger codes. Other prior decoders, such as Clique and NISQ+, are not only inaccurate but also rely on emerging technologies, making them impractical for adoption on near-term FTQCs [33, 48, 58, 60]. As shown in Figure 1(b), the goal of this paper is to enable real-time and accurate decoding of surface codes in the near term ($d = 7$ and $d = 9$) while relying on commodity hardware (e.g. FPGAs). This enables seamless adoption as FPGAs are currently used to design the control and readout circuitry in most near-term quantum systems.

We develop and design a practical real-time decoder by focusing on the *Hamming weight (HW) or the number of 1s in the syndrome vector* (the concatenation of all syndromes generated across the d rounds). The Hamming weight of the syndrome vector determines the complexity of the MWPM. For example, if the Hamming weight is four, then we are looking for pairings corresponding to four nodes, which has only six possibilities and can easily be done via a brute-force search. Our key insight is that most syndrome vectors have low Hamming weights. For example, syndrome vectors of weight greater than 10 occur with a probability lower than the logical error rate (6×10^{-9}) in a $d = 7$ surface code at a physical error rate of 10^{-4} (default). Our proposed design, *Astrea*, focuses on brute force searching for syndrome vectors of weight up-to 10 and ignores decoding higher Hamming weight syndromes (as not decoding them is unlikely to impact the logical error rate), as shown in Figure 1(c). Astrea uses a hardware circuit (HW6) that can match up-to 6 nodes (15 possible pairings) and uses this to decode syndromes with Hamming weight of 8 (105 matchings, resulting in 7 accesses to the HW6 circuit) and Hamming weight of 10 (945 matchings, resulting in 63 accesses to the HW6 circuit). Astrea accomplishes the goal of accurate real-time decoding for $d = 7$ surface codes with average and worst-case latencies of 1ns and 456ns respectively when implemented on standard off-the-shelf FPGAs from Xilinx. *To the best of our knowledge, Astrea is the first practical decoder that achieves MWPM accuracy while decoding for up-to* $d = 7$ *surface codes in real-time.*

The Hamming weight of a syndrome depends on *the physical error rate* and the distance of a code. At higher error rates or code distances (e.g. $d = 9$), we observe Hamming weights up-to 20. Unfortunately, Astrea's exhaustive search is infeasible for such large Hamming weights due to an exponential increase in the number of possible matchings. For example, a syndrome of Hamming weight 20 has 6.5×10^8 possible matchings.

To decode such syndromes, we propose *Greedy-Astrea* or *Astrea-G* which greedily searches for the MWPM by (1) restricting the

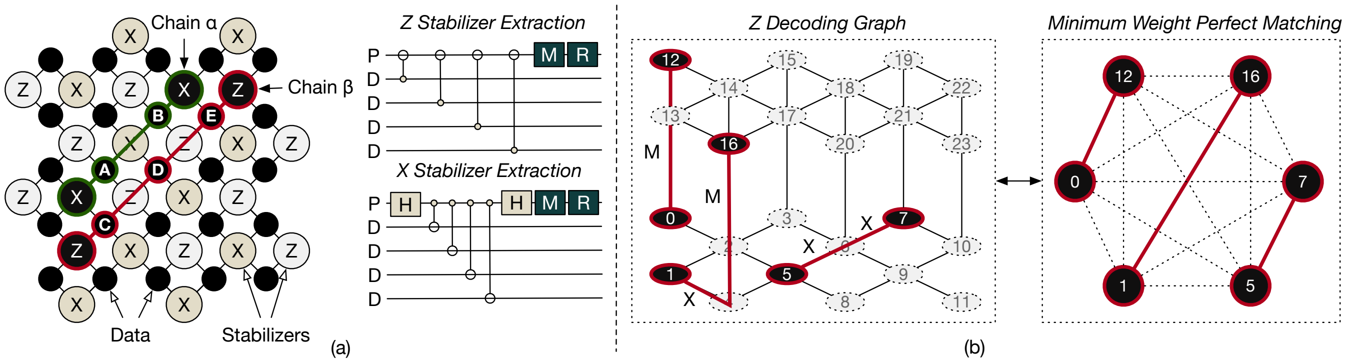

Figure 2: (a) A distance 5 surface code and the syndrome extraction circuits for each stabilizer. (b) Example of decoding Z stabilizer measurements from a distance 5 surface code using MWPM decoding. Two rounds are shown for simplicity (d rounds are needed in practice). The corresponding MWPM is also shown on the right.

search space and (2) searching for the MWPM greedily. Astrea-G leverages the insight that although a syndrome bit has $w-1$ possible pairings for a Hamming weight w, only about two to three of them have low weights (thus occurring with probability higher than the logical error rate). These low-weight pairs have the *highest* probability of appearing in the MWPM. The remaining pairings have high weights and have a very low likelihood of appearing in the MWPM and can be ignored, as shown in Figure 1(d). Astrea-G performs an efficient search by (1) *filtering out* high-weight pairs to reduce the search space and (2) *prioritizing* low-weight pairs during the matching step. Our evaluations with Google's *Stim* framework [24] shows that Astrea-G provides a logical error rate similar to the software-based MWPM (ideal setting), for up-to $d = 9$ codes (with physical error rates between 10^{-3} to 10^{-4}), while meeting the real-time target latency of $1\mu s$ (average is 450ns). Similar to Astrea, Astrea-G also can be implemented using off-the-shelf Xilinx FPGAs, making it a *practical, low-cost solution for near-term FTQCs.*

Overall, this paper makes the following contributions:

(1) We show that efficient MWPM is possible by focusing on the Hamming weight of the syndrome vector – low Hamming weight syndromes are common and easy to decode.

(2) We propose *Astrea*, a real-time decoder that uses exhaustive search to perform MWPM for surface codes up to distance 7. To the best of our knowledge, this is the first real-time decoder that achieves the accuracy of MWPM at $d = 7$.

(3) We show that the search can be made even more efficient by exploiting the non-uniformity in weights and eliminating matchings that have higher weights (based on the logical error rate).

(4) We propose *Astrea-G* that performs MWPM greedily by restricting the search and works even at $d = 9$. To the best of our knowledge, Astrea-G is the first real-time decoder that provides comparable accuracy as the MWPM decoder at $d = 9$.

2 BACKGROUND AND MOTIVATION

2.1 Quantum Error Correction and Surface Code

Many promising quantum applications require error rates orders of magnitude lower than what we expect to achieve on future quantum devices [8, 12, 25, 29, 30, 37, 49, 53]. To close this gap, we expect to use *Quantum Error Correction*, which involves encoding logical qubits from constituent data and parity qubits. Google's recent success in demonstrating a distance 5 surface code has established the surface code as a leading candidate for QEC [1]. In this paper, we focus on real-time decoding for the surface code in the near-term.

Surface codes encode a logical qubit of distance d in a lattice of d^2 data qubits and $d^2 - 1$ parity qubits, as shown in Figure 2(a) [17, 22, 36, 57]. Table 1 shows the redundancy for surface codes. Any error on a data qubit can be projected into a combination of Pauli errors (I, X, Y, Z) and are detected by its adjacent parity qubits by measuring a 4-qubit operator, called a *stabilizer*. Qubits encounter bit-flip (X) or phase-flip (Z) or a combination of both (Y) errors. X errors are detected by Z stabilizers, and Z errors are detected by X stabilizers. The number of errors determines the length of an error chain. A distance d code can correct all error chains of at most length $\lfloor \frac{d-1}{2} \rfloor$. For example, in Figure 2(a), Z errors on data qubits Ⓐ and Ⓑ cause a *correctable* chain α with length 2, whereas the X errors on data qubits Ⓒ, Ⓓ, and Ⓔ cause an *uncorrectable* chain β with length 3. *Irrespective of its length, an error-chain flips at most two syndrome bits flip at its endpoints.*

Table 1: Resources required for surface code logical qubits

Code Distance	Number of Physical Qubits			Syndrome Vector Length (X / Z)
	Data	Parity ($X + Z$)	Total	
3	9	$4 + 4 = 8$	17	16 / 16
5	25	$12 + 12 = 24$	49	72 / 72
7	49	$24 + 24 = 48$	97	192 / 192
9	81	$40 + 40 = 80$	161	400 / 400

2.2 Minimum Weight Perfect Matching (MWPM) Decoding

Decoders must identify (1) errors that accumulate on data qubits and (2) errors in syndrome extraction operations, namely *measurement*, *reset*, and *gate* errors [13, 14, 22]. Data qubit errors manifest as failed parity checks in the same round, whereas errors during syndrome extraction result in failed parity checks across consecutive rounds. At distance d, a decoder must decode d consecutive syndromes simultaneously to tolerate error chains of up to length $\lfloor \frac{d-1}{2} \rfloor$ caused by syndrome extraction in order to preserve the properties of the code. We refer to the collective syndrome from d consecutive rounds as *syndrome vector*, which is used by a decoder.

Surface code decoding can be reduced to a Minimum Weight Perfect Matching (MWPM) problem on a fully-connected weighted graph. Given a syndrome S with nonzero bits $s_1, \ldots, s_k$, we construct a complete graph with $s_1, \ldots, s_k$ as vertices. Each edge (s_i, s_j), $i \neq j$ has a weight based on the probability of an error chain causing both bits, s_i and s_j, to flip; a higher weight indicates lower probability. Then, to decode the syndrome S, we match each s_i with a unique s_j to create a *perfect matching* M which minimizes the aggregated weights of the pairings. For each (s_i, s_j), errors are corrected using the shortest path between the parity qubits for s_i and s_j. Consequently, M corresponds to the *highest probability error event* that would result in the given syndrome. Finally, to tolerate measurement and gate errors, the shortest path travels across multiple rounds. If k is odd, then one syndrome bit s_b is matched to the boundary of the surface code lattice. X syndromes and Z syndromes are decoded independently.

We explain the functionality of MWPM decoding using Figure 2(b). In the Z decoding graph spanning two rounds of a distance 5 surface code, there are 6 flipped Z stabilizers, so we have 3 pairs in the matching. The stabilizers are translated into a weighted graph, where MWPM is performed. Then, the shortest paths between each matched pair are used to identify any errors that have occurred.

Unfortunately, existing software implementations of MWPM such as BlossomV and PyMatching are too slow to decode within the worst-case $1\mu s$ time constraint [31, 32, 38]. For example, BlossomV cannot decode about 96% of nonzero syndromes within $1\mu s$ for a $d = 7$ code, as shown in Figure 3.

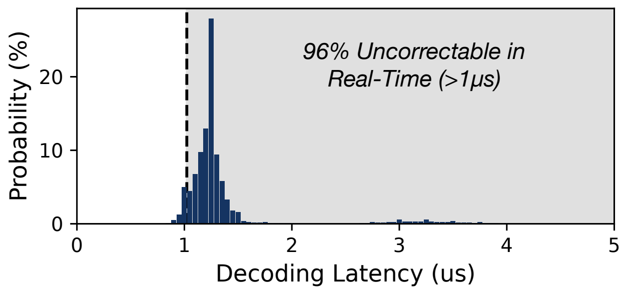

Figure 3: Decoding latencies of MWPM decoding using BlossomV [38], a software implementation (96% of syndromes cannot be decoded in real-time).

2.3 Prior Works on Hardware-Based Decoding

As existing software MWPM implementations are slow, few prior works propose hardware approaches. Unfortunately, these designs do not scale and thus, there has been a paradigm shift towards building approximate decoders that trade off accuracy for latency.

2.3.1 Theoretical MWPM using ASICs. Fowler suggested MWPM using ASIC arrays [21] but this design is impractical due to the complexity of implementing the BlossomV algorithm in hardware.

2.3.2 Lookup-Table MWPM. LILLIPUT uses MWPM to program lookup tables that can be indexed using a syndrome to identify a logical correction [13, 57]. However, they are not scalable due to exponentially increasing memory costs and are not usable beyond distance 5 decoded with 2 rounds only.

2.3.3 Approximate Decoders. They trade-off accuracy to decode syndromes in real time [14, 33, 34, 51, 58, 60]. For example, the AFS decoder uses the fast and simple Union-Find (UF) algorithm to decode syndromes [16]. However, the UF algorithm is less accurate than MWPM [18, 19], resulting in 100×-1000× worse logical error rates as shown in Figure 4. Superconducting decoders like NISQ+, QECOOL, QULATIS make the same trade-offs but worsen accuracy even further as they consider fewer than d syndrome rounds [33, 58, 60]. Moreover, these designs rely on emerging technologies and therefore, are impractical for near-term FTQCs.

2.3.4 Hierarchical Decoders. They classify error events into two types and use a separate decoder for each: *small-events* with few non-zero syndrome bits that are easy to decode and *large-events* otherwise that are hard to decode [9, 15, 48, 54]. Unfortunately, although small-event decoders can quickly decode a subset of syndromes, the overall performance is still limited by the large-event decoder, typically a software MWPM decoder. Also, the logical error rate worsens due to the inaccuracies of small-event decoders [9, 48]. These glaring issues limit the applicability of hierarchical decoders.

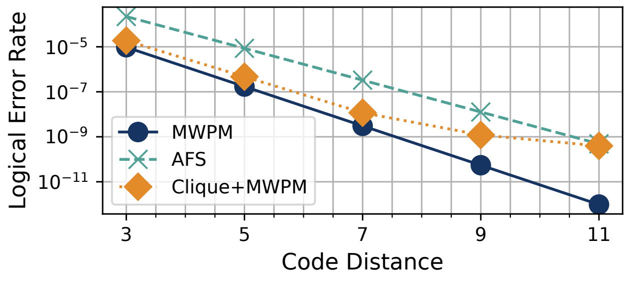

Figure 4: Logical error rate of MWPM, AFS, and Clique decoders for increasing distance at a physical error rate $p = 10^{-4}$. MWPM and Clique+MWPM are not real-time decoders.

Goal: Real-Time Decoding with Accuracy of MWPM

Our goal is to design a practical decoder that decodes surface codes of distance d using d rounds of syndromes in real-time using commodity hardware while achieving MWPM accuracy.

3 EVALUATION METHODOLOGY

We discuss the evaluation methodology before discussing our insights and proposed design.

3.1 Surface Code Parameters

We consider rotated surface codes up-to distance 9 which indicates a target regime for the next few years. The largest demonstrated surface code till date corresponds to distance 5 on Google Sycamore [1].

3.2 Noise Model

In this paper, we consider physical error-rates p up-to an order of magnitude lower than the surface code thresholds, which represents an achievable and target device quality on near-term quantum computers. Thus, p ranges between 10^{-3} to 10^{-4}. We consider a circuit-level noise model where *depolarizing* (X, Y, Z) errors are inserted with a probability p (1) on data qubits at the beginning of every round, (2) on data and parity qubits after syndrome extraction operations, and (3) on parity qubits after measurement and reset operations. This enables us to account for errors during syndrome extraction in addition to errors on data qubits. We also select this error model as it better reflects real-world devices [22, 24, 27, 40, 41].

3.3 Baseline: Software BlossomV implementation of MWPM

We use an MWPM decoder implemented using the BlossomV algorithm as our baseline as it is often regarded as the gold-standard for decoding surface codes [2, 13, 14, 33, 38, 48, 50, 56, 58, 60].

3.4 Simulation Infrastructure

We use a *cycle-accurate* simulator integrated with Google's Stim framework [24], a state-of-the-art Monte Carlo QEC simulator used in prior works [4, 6, 26, 27, 61], to perform state-preservation, or *memory experiments*, which are used to evaluate the logical error rate of a decoder [2, 11, 13, 14, 33, 50, 56, 58, 60]. In each experiment, we (1) prepare an initial state, (2) inject errors for d syndrome extraction rounds, (3) measure the d^2 remaining data qubits to perform a logical measurement. This results in $d(d^2-1)$ syndrome bits for the d syndrome extraction rounds and an additional d^2-1 bits to account for any measurement errors during the logical measurement. Memory experiments can be categorized as Z and X experiments. While both are necessary to evaluate the performance of a QEC code, they are functionally equivalent because the noise model is identical for X and Z errors [7, 13, 14, 22, 33, 48, 58, 60]. We consider Z memory experiments, which involve initializing the logical qubit in the $|0\rangle$ state, measuring Z stabilizers for d rounds, and measuring the logical qubit in the computational (Z) basis.

During each experiment or trial, Astrea receives a syndrome every 1μs which is the syndrome extraction latency on Google Sycamore. After d rounds, where d is the code distance, Astrea applies its correction to the logical qubit and measures the logical state. Decoding is successful if the logical measurement matches the initial state prepared. Otherwise, a logical error has occurred.

We target a 250MHz FPGA implementation for Astrea on Xilinx's Zynq UltraScale+, synthesize our design using Vivado, and verify its correctness using Verilator.

4 ASTREA: KEY INSIGHTS

In this section, we discuss how errors manifest in surface codes and our insights to build a practical real-time decoder.

4.1 How do errors manifest in surface codes?

For high accuracy, a decoder must handle errors that accumulate on data qubits as well as errors that occur during syndrome extraction. Errors on data qubits cause failed parity checks or non-zero syndrome bits during syndrome generation in the same round. These events, shown in Figure 5(a), are known as ***space*** events. Measurement and reset errors during syndrome extraction cause a parity check failure on a parity qubit in two consecutive rounds, resulting in a ***time*** event, as shown in Figure 5(b). Finally, CNOT errors during syndrome extraction can manifest in space, time, or a combination of both: a ***space-time*** event, as shown in Figure 5(c). Errors across d rounds lead to either of the following: (1) an isolated error (an error chain of length 1), (2) a single error chain of length greater than 1 spanning both space and time, or (3) a combination of both isolated errors and error chains. However, any error-chain flips at most two syndrome bits despite their length and thus, the total number of non-zero bits in a syndrome vector is at most twice the total number of error-chains observed in a logical cycle.[1]

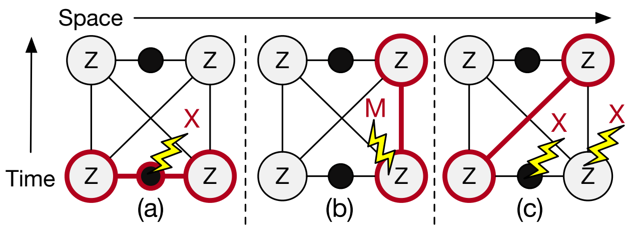

Figure 5: (a) An X data qubit error (space). (b) A measurement error on a Z stabilizer (time). (c) A CNOT error causes an X data qubit error and an X error on a Z stabilizer (space-time).

4.2 Insight #1: Most syndromes are low weight

We analyze the frequency of syndrome vectors by Hamming weight by (1) Using an analytical model for *upper-bound*, and (2) Experimental data using a circuit-level noise model.

4.2.1 Analytical Model for Upper-Bound. In the worst case, every error causes two syndrome bits to flip. There are *five* sources of error during syndrome extraction that causes two syndrome bits to flip (in space, time, or both). These are: (1) X, Y depolarizing errors on the four adjacent data qubits (probability $4\times\frac{p}{2}=2p$), (2) a measurement error on the parity qubit (probability p), (3) a reset error on the parity qubit (probability p), (4) X, Y depolarizing errors on the adjacent data qubits due to CNOT operations (probability $4\times\frac{p}{2}=2p$), and (5) X, Y depolarizing errors on the parity qubit due to CNOT operations (probability $4\times\frac{p}{2}=2p$). Altogether, the syndrome extraction of a parity qubit can cause two syndrome bits

[1]A *syndrome* is the result of stabilizer measurements from one round. A decoder must decode syndromes aggregated from d consecutive rounds (which constitutes a *logical cycle*). We refer to the collection of syndromes from d rounds as a *syndrome vector.*

to flip with probability $2p+p+p+2p+2p = 8p$. We model the number of syndrome extraction errors using a random variable E such that $E \sim \text{Binomial}(D, 8p)$ where $D = (d+1) \times \frac{d^2-1}{2}$ is the number of syndrome bits. Also, as each syndrome extraction error causes *two* syndrome bit flips, we model the Hamming weight of a syndrome vector as a random variable $H = 2E$. Thus, a syndrome vector with Hamming weight h occurs with the probability in Equation (1).

$$\mathbb{P}(H = h) = \binom{D}{\frac{h}{2}}(8p)^{\frac{h}{2}}(1-8p)^{D-\frac{h}{2}} \tag{1}$$

Thus, higher Hamming weight syndrome vectors have an exponentially lower probability than lower Hamming weight ones, so most syndrome vectors have *low Hamming weight.*

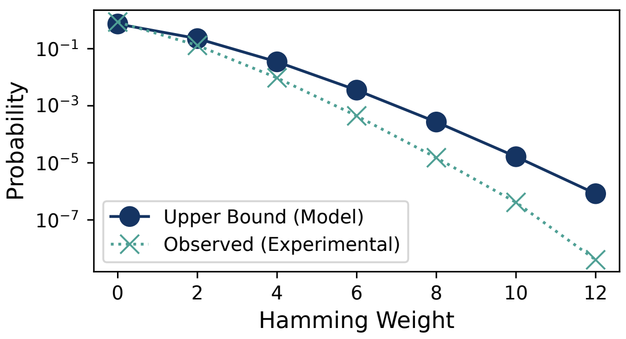

Figure 6: Syndrome-vector probabilities by Hamming weight according to *analytical* and circuit-level noise models.

4.2.2 Experimental Data Using Circuit-Level Simulation. In reality, if multiple errors occur adjacent to each other, they form an *error-chain* which only results in two syndrome bits flipping. Moreover, errors can also cancel each other out. Figure 6 shows the probability of a given Hamming weight using the analytical model (worst-case) and the experimental data. While the real frequencies are significantly lower (as not all errors flip two syndrome bits), the probabilities follow the exponential decay of the analytical model.

Table 2: Syndrome Vector Probability by Hamming Weight

Hamming Weight	Prob. ($d = 3$)	Prob. ($d = 5$)	Prob ($d = 7$)
0	0.99	0.95	0.86
1, 2	1.1×10^{-2}	0.05	0.13
3, 4	4.2×10^{-5}	1.26×10^{-5}	9.5×10^{-3}
5, 6	6.5×10^{-8}	1.9×10^{-5}	4.4×10^{-4}
7 to 10	0	1.9×10^{-7}	1.6×10^{-5}
> 10	0	0	4×10^{-9}
Logical Error Rate	8.1×10^{-6}	1.3×10^{-7}	6×10^{-9}

Table 2 shows the probability of obtaining a syndrome vector of a given Hamming weight for various distances (at p=10^{-4}). Note that the probability of a syndrome vector with Hamming weight greater than 10 is *lower* than the logical error rate for distances up to 7. *Thus, a decoder that can correct up to Hamming weight 10 is enough to reliably decode surface codes up-to distance 7.*

4.3 Insight #2: Low Hamming weight syndrome vectors only have limited perfect matchings

A syndrome vector with Hamming weight w has

$$\frac{w!}{2^{w/2} \times (w/2)!} \tag{2}$$

perfect matchings. For example, a syndrome vector with $w = 4$ has 3 perfect matchings, and a syndrome vector with $w = 10$ has 945 perfect matchings. As there are only a few hundred matchings, an exhaustive search can quickly find the MWPM amongst all perfect matchings, so they are *practical to decode.* We propose *Astrea* which leverages this insight.

5 ASTREA: DESIGN

Astrea is designed to target real-time decoding for surface codes in the near-term, specifically between distances 3 to 7. We assume a physical error rate 10^{-4}. We first explain the organization of weights and then our hardware implementation.

5.1 Global Weight Table

To compute the MWPM for a decoding graph, Astrea requires the collection of weights between every possible pair of syndrome bits. Each weight is an 8-bit value corresponding to $-\log_{10}$(*probability of the pair matching*). Thus, a match occurring with a 1 in a million probability will have a weight of 6. These weights are stored in an on-chip memory and we refer to this data structure as the *Global Weight Table (GWT).* For a syndrome vector of length ℓ, the GWT maintains a $\ell \times \ell$ matrix of weights corresponding to all possible pairs. We use the diagonal weight to represent the probability that the given node matches with the boundary. When Astrea receives a non-zero syndrome vector for decoding, it retrieves weights corresponding to all possible pairings of the non-zero syndrome bits in the vector and places them into an *Active Weight Array.* For example, let W_{ij} denote the 8-bit quantized weight for the pair corresponding to non-zero syndrome bits i and j in a syndrome vector. If a syndrome vector has three non-zero bits denoted by a, b, and c, the weights W_{ab}, W_{ac}, and W_{bc} are retrieved from the GWT and placed in the Weight Array.

5.2 Evaluating Common Case Syndrome Vectors

We discuss the difficulty of decoding the lowest weight syndrome vectors with Hamming weights between 0 to 6. We divide our analysis into three parts based on the difficulty.

5.2.1 Hamming Weights Until 2 are Trivial. We observe that MWPM for Hamming weights 0, 1, and 2 is trivial. Syndrome vectors of Hamming weight 0 correspond to a case of no-errors. On the other hand, syndrome vectors of Hamming weights 1 and 2 only have a single possible option for matching.

5.2.2 Decoding Hamming Weights 3 and 4. Syndrome vectors of Hamming weights 3 and 4 both have only 3 possible perfect matchings.[2] Consider a syndrome vector with nonzero bits $a, b, c,$ and d. As shown in Figure 8, there are six possible pairings (= ${}^4C_2$) of these four non-zero syndrome bits, resulting in six weights. As a perfect

[2]The Hamming weight 3 case is the same as the Hamming weight 4 case because one syndrome bit will be matched to the boundary.

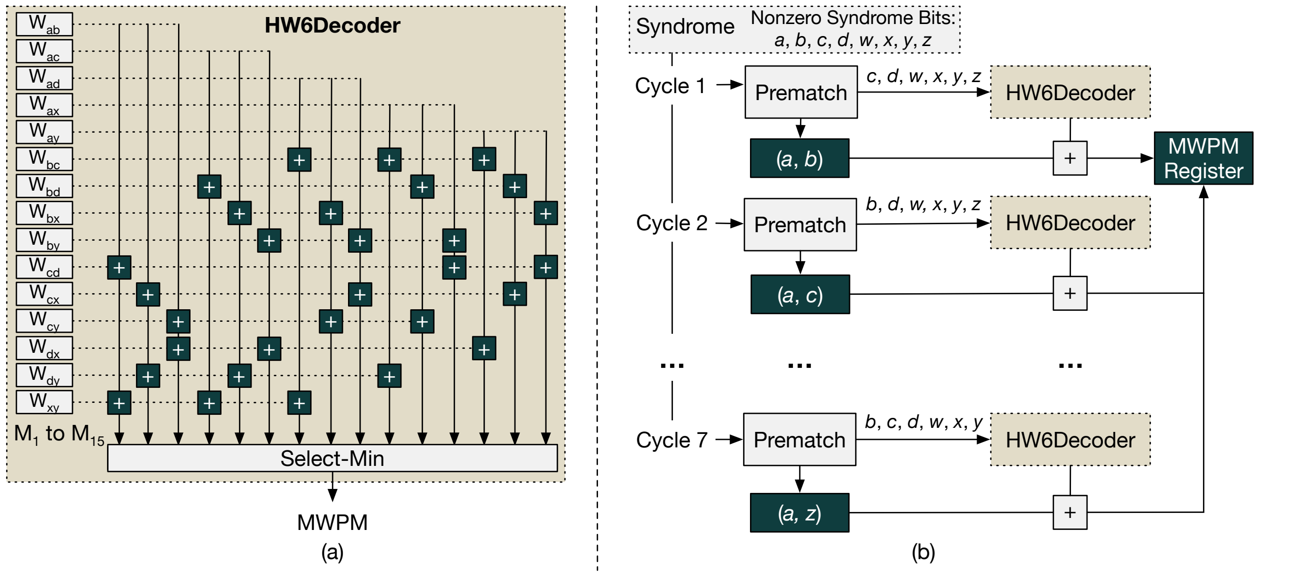

Figure 7: (a) Astrea's `HW6Decoder` that decodes syndromes with Hamming weight 6. For 15 possible perfect matchings, each option requires two 8-bit adders (a total of thirty 8-bit adders) (b) Using `HW6Decoder` for syndrome vectors of Hamming weight 8.

matching for a Hamming weight 4 syndrome vector has two pairs of syndrome bits, three perfect matchings are created by combining two pairs of syndrome bits. Finally, the perfect matching with the lowest aggregated weight is selected as the MWPM.

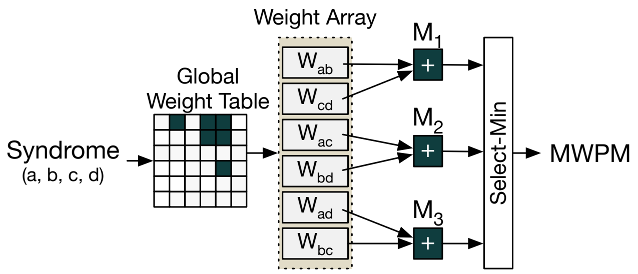

Figure 8: Logic to decode syndromes of Hamming weights 3 or 4. There are three possible matchings, each of which contains two pairs of syndrome bits.

5.2.3 Decoding Hamming Weights 5 and 6. Decoding syndrome vectors of Hamming weights 5 or 6 uses a similar approach to the one used for syndrome vectors with Hamming weights 3 or 4. As shown in Figure 7(a), 15 weights (=$^6C_2$) are loaded into the Weight Array. Then, using a network of 30 adders, these 15 weights are combined into 15 perfect matchings such that each perfect matching contains three pairs of syndrome bits, similar to how pairs are combined to create perfect matchings in Figure 8. The logic unit described in Figure 7(a), the `HW6Decoder`, is Astrea's *fundamental building block* and is used to decode higher Hamming weight syndrome vectors.

5.3 Decoding Less Common Cases

Astrea can decode higher Hamming weight syndrome vectors by *pre-matching* a few syndrome bits and then using the `HW6Decoder` to match the remaining syndrome bits. Figure 7(b) shows how to build a decoder for syndrome vectors up to Hamming weight 8 using `HW6Decoder` as a building block. Astrea splits the eight nonzero syndrome bits into two parts: (1) two syndrome bits that are *pre-matched*, and (2) six syndrome bits that are matched using `HW6Decoder`. Syndrome bit a is paired seven times, once with each of b, c, d, w, x, y, z. For example, in cycle 1, a is paired with b and the MWPM for bits c through z is combined with (a, b) to form a new perfect matching M_1, and M_1 is placed in the *MWPM Register*. Every cycle, a new perfect matching M_i (in cycle i) is created, and the the MWPM Register is updated if the M_i has a lower weight than the matching currently stored in the MWPM register. Thus, at the end of the seventh cycle, the MWPM Register contains the MWPM for the Hamming weight 8 syndrome vector because all possible perfect matchings have been exhausted.

Astrea can decode syndrome vectors up to Hamming weight 10 by scaling up the same strategy. Instead of pre-matching only a pair of bits, Astrea must pre-match two pairs of bits each cycle. As there are 9 options for the first pair and 7 options for the second pair, decoding a Hamming weight 10 syndrome vector takes 63 cycles.

5.4 Astrea Overheads and Latency

We synthesized and implemented Astrea on Xilinx's Zynq UltraScale+ FPGA. Table 3 shows the utilization and maximum clock frequency of Astrea on the FPGA. Astrea requires less than 10% of the LUTs and BRAM on the FPGA while running at 250MHz clock frequency. Importantly, Astrea does not require connecting an additional FPGA to existing quantum control infrastructure, because its low overheads ensure it can be co-located with the control processor, which is also typically implemented on an FPGA in commercial systems [3, 13, 23, 35, 45, 55]. Thus, Astrea does not require significant changes to the quantum control infrastructure.

Table 3: FPGA Synthesis and Implementation of Astrea

LUT%	FF%	BRAM%	Max Freq. (MHz)
5.57	0.86	9.60	250

We discuss the latency of our design. Before decoding begins, Astrea transfers the weights from the Global Weight Table (GWT) to the weight array. Our design requires HW + 1 cycles to retrieve all weights from the GWT and transfer them to the weight array, incurring up-to 11 cycles for a syndrome vector with Hamming weight 10. Moreover, our design requires 1, 11, 103 cycles to decode Hamming weights 3-6, 7-8, and 9-10 respectively; note that Hamming weights 0-2 are trivial and need not be decoded. Thus, Astrea's worst case latency is 103 + 11 = 114 cycles for a syndrome vector with Hamming weight 10. Figure 9 shows the mean and maximum latencies of Astrea for $d = 3, 5, 7$ at $p = 10^{-4}$. The average latency remains within 1ns, whereas the worst-case latency ranges from 32ns ($d = 3$) to 80ns ($d = 5$) to 456ns ($d = 7$). Thus, Astrea operates well within the 1μs constraint for real-time decoding.

5.5 Logical Error Rate for Astrea

We evaluate Astrea for code distances of 3, 5, and 7 considering a physical error rate of 10^{-4}. As shown in Table 4, Astrea achieves the same logical error rate as MWPM, demonstrating that it does not compromise accuracy. Also note that there is an exponential suppression of errors with increasing redundancy or code distance.

Table 4: Logical Error Rate (LER) for different decoders at $p = 10^{-4}$ for a distance d decoder using d syndrome rounds

d	MWPM	Astrea	LILLIPUT	Clique	AFS
3	8.1×10^{-6}	8.1×10^{-6}	8.1×10^{-6}	8.3×10^{-6}	9.4×10^{-5}
5	1.3×10^{-7}	1.3×10^{-7}	N/A	1.4×10^{-6}	2.3×10^{-5}
7	6.0×10^{-9}	6.0×10^{-9}	N/A	2.3×10^{-8}	5.7×10^{-7}

Astrea Enables Real-Time MWPM Decoding up-to d=7

To the best of our knowledge, Astrea is the first low-cost, practical hardware-based solution that can *decode surface codes up to distance 7 in real-time and with the same accuracy as an idealized MWPM decoding.* Thus, Astrea is ideal for adoption in near-term fault-tolerant systems.

5.6 Comparison With Prior Works

LILLIPUT: It can only decode distance 3, and distance 5 with only two rounds of syndromes due to exponentially increasing memory requirements. For example, LILLIPUT requires 2×2^{60} bytes to decode distance 5 codes using 5 syndrome extraction rounds. For distance 7, it requires 2×2^{168} bytes! In contrast, Astrea is practical up to $d = 7$.

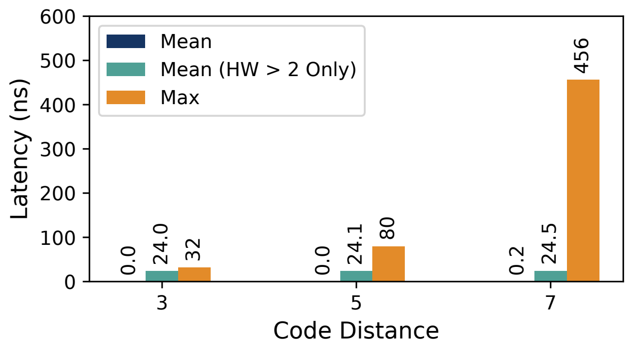

Figure 9: Latency of Astrea when decoding syndromes for distance 3, 5, and 7. Astrea meets real-time decoding requirements as it operates within 1μs. Astrea takes 0ns to decode Hamming weight ≤ 2 as these are trivial syndromes.

AFS: Table 4 shows that AFS has 105× higher logical error rate compared to MWPM and Astrea.

Clique: Astrea achieves up to 3.8× lower error rate than this decoder in real-time, as shown in Table 4. The Clique decoder is unable to decode all error events in real-time due to its reliance on the software MWPM decoder for "hard to decode events", which dominates the design's critical path. Moreover, it relies on emerging superconducting device technologies, limiting its applicability to near-term FTQCs.

5.7 Limitations of Astrea

Although Astrea can correct syndrome vectors with Hamming weights up to 10, it cannot tolerate higher Hamming weights. For example, a Hamming weight 20 syndrome has 6.5×10^8 possible perfect matchings, which is impractical for an exhaustive search. This limits the scalability of Astrea beyond Hamming weights of 10. Note that (a) higher physical error rates and/or (b) larger distances cause higher Hamming weight syndromes to appear frequently.

A higher physical error rate causes errors on data qubits and measurement operations to occur more frequently, resulting in *more independent error chains* and thus, more non-zero bits in the syndrome vector. Table 5 shows that the probability of obtaining *high Hamming weight* (> 10) syndrome vectors at physical error rates of $p = 10^{-3}$ and $p = 10^{-4}$ for distance $d = 7$. At $p = 10^{-3}$, Hamming weights larger than 10 occur with almost 1000× higher probability than the logical error rate.

Table 5: Syndrome vector probability by Hamming weight

Hamming Weight	Prob. ($p = 10^{-3}$)	Prob. ($p = 10^{-4}$)
0	0.22	0.859
1 to 10	0.777	0.141
> 10	0.003	4×10^{-9}
Logical Error Rate	2.4×10^{-7}	6×10^{-9}

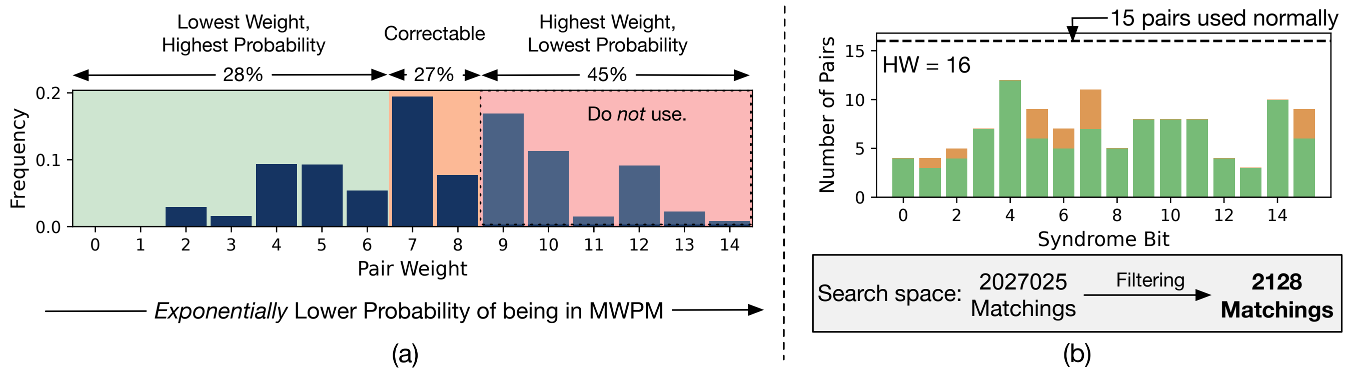

Figure 10: (a) Distribution of weights for $d = 7, p = 10^{-3}$. The weights can be divided into three regions depending on the probability they will appear in the MWPM. (b) Number of pairs for each syndrome bit for a Hamming weight 16 syndrome vector after eliminating all weights above 8. Removing high-weight pairs results in a 58% reduction in the number of syndrome bit pairs, resulting in a 953× for the MWPM search space.

6 ASTREA-G: KEY INSIGHTS

To extend Astrea to decode larger code distances and higher physical error rates, we propose *Greedy-Astrea* or *Astrea-G*. We discuss the key insights of Astrea-G which enables it to find the MWPM for high Hamming weight syndromes.

6.1 Insight #1: Filter Unlikely Weights

For a syndrome vector of length ℓ, there are $\ell \times \ell$ weights in the GWT. Figure 10(a) shows the distribution of the weights in the GWT (recall that weights are $-\log_{10}(P)$, so 14 denotes a likelihood of 10^{-14}). Weights that correspond to events which occur at a significantly lower probability than the logical error rate are highly unlikely to be part of the MWPM pairings as they represent improbable events. Therefore, we can simply skip the pairings that use weights greater than a certain threshold (W_{th}). For $d = 7$ ($p = 10^{-3}$), the logical error-rate is approximately 10^{-5}, so weight values of 5 represent events occurring at similar probability. We pick a W_{th} that is 2 greater (so occurring with 100× lower probability) than this weight for filtering. The histogram in Figure 10(a) is colored into green (weights ≤ 7), orange (weights between 7 and 9) and red (weights ≥ 9). We only use pairings with green weights.

6.2 Insight #2: Search from Low to High Weights

For the green region, low-value weights are orders of magnitude more likely than high-value weights. Therefore, when performing a greedy search, it is useful to first exmaine the low value weights and only then try the higher value weights. With this strategy, we are very likely to converge to the MWPM early in the search.

Insights: Not all Matches are Created Equal

Instead of trying all possible pairings, we can make the search much more efficient by (a) filtering out the weights that represent improbable events as they are unlikely to result in MWPM (b) order the greedy search of the remaining candidates from low weights to high weights.

7 ASTREA-G: DESIGN

Figure 11 shows an overview of Astrea-G. Similar to Astrea, Astrea-G starts by retrieving weights for each pair of non-zero bits in the syndrome vector from the GWT and puts them into the Local Weight Table (LWT). However, Astrea-G also filters out any weights larger than a pre-configured cutoff or *Weight Threshold*, W_{th} to constrain the MWPM search space. For a target logical error-rate of P_L, we use W_{th} equal to $-\log_{10}(0.01 \times P_L)$, to suppress events that occur with 100× lower probability than the logical error rate.

7.1 Organization of the Matching Pipeline

Astrea-G attempts to pre-match syndrome bits until six syndrome bits remain, which are matched exhaustively using the `HW6Decoder` logic unit. Astrea-G orders pre-matchings of syndrome bits greedily by prioritizing low-weight pairs first because there is a very high probability that the MWPM will contain most of these pairs.

Astrea-G implements the greedy search using a *Matching Pipeline* that receives pre-matchings from priority queues. This pipeline has three stages: *Fetch*, *Sort*, and *Commit*. Our default design fetches one pre-matching from each of the F priority queues. A priority queue contains up to a maximum of E entries. At the initialization step, the priority queue entries are empty and as the pipeline processes pre-matchings, the priority queue entries are updated.

In the *Fetch* stage, Astrea-G fetches syndrome bit pairs for an unmatched syndrome bit and pops a pre-matching off a priority queue. In the *Sort* stage, these syndrome bit pairs are sorted based on their weight. Finally, in the *Commit* stage, Astrea-G chooses the top F bit pairs with the lowest weight and combines each of them with the prior pre-matching from the *Fetch* stage to create F new pre-matchings. *If only six syndrome bits are unmatched*, then Astrea-G uses the `HW6Decoder` unit to exhaustively match the remaining six syndrome bits and combines this matching with the pre-matching to create a perfect matching. This perfect matching is used to update the MWPM Register. *Otherwise*, the F new pre-matchings are pushed onto the priority queues such that each priority queue receives a pre-matching.

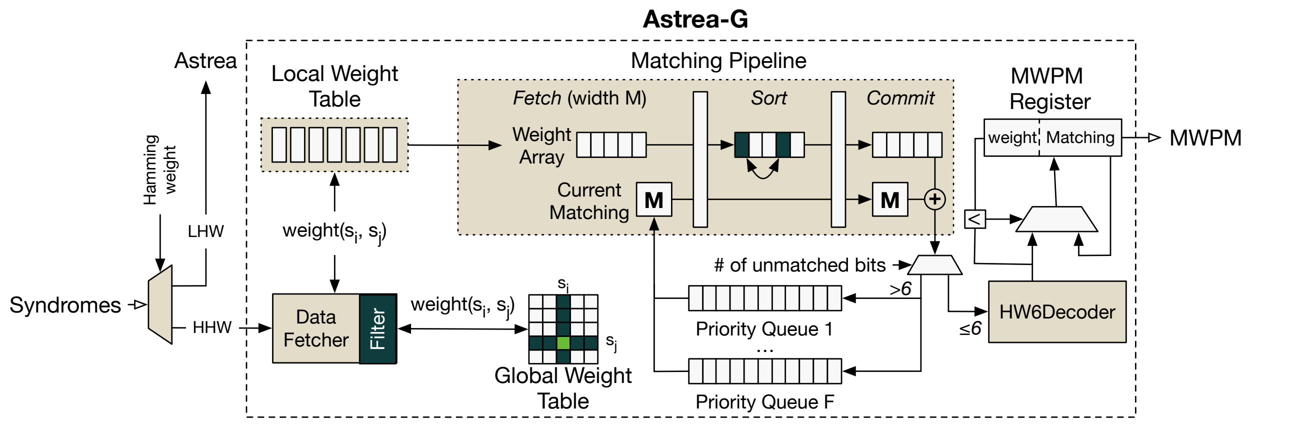

Figure 11: Micro-architecture of Astrea-G for decoding High Hamming-Weight (HHW) syndromes that Astrea cannot decode. Unlike Astrea, Astrea-G computes perfect matchings iteratively, using priority queues and sorting to prioritize lower weight pairs. Furthermore, Astrea-G filters out higher-weight pairs from the GWT to constrain the search space of a HHW syndrome.

The priority queues re-orders the pre-matchings based on the score s/b, where s is the cumulative weight of a pre-matching and b is the number of matched bits in the same pre-matching. Astrea-G iterates using the pipeline and finishes execution when either the priority queues are empty or $1\mu s$ has elapsed. Once the priority queues are empty, the MWPM register is guaranteed to contain the MWPM. On the other hand, if the priority queues are not empty at the end of $1\mu s$, the MWPM Register is not guaranteed to contain the MWPM. However, because Astrea-G has prioritized low-weight syndrome bit pairs, the MWPM is most likely already in the MWPM Register. Our evaluations show that a fetch width of $F = 2$ and priority queue sizes of $E = 8$ are sufficient and we use these parameters for our default design. Larger fetch widths and priority queues improve accuracy but require more logic to implement and may have longer access times.

Astrea-G greedily prioritizes low-weight pairs in two places. *First*, by sorting syndrome bit pairs and only committing the lowest-weight pairs into the priority queues in each *Commit* Stage, Astrea-G filters out higher weight pairs. *Second*, as priority queues score and sort pre-matchings based on their weights and progress represented by the number of matched syndrome bits, higher weight pre-matchings with few matched bits are pushed to the end of the priority queues from where they seldom leave. Furthermore, because each priority queue has a fixed size, these higher weight pre-matchings are evicted as a greater number of lower weight pre-matchings takes precedence and are introduced into the queue.

7.2 Astrea-G's Performance and Latency

Figure 12 shows the logical error-rate of Astrea-G and idealized implementation of MWPM for distance 7 surface codes when the physical error rates is varied from $p = 10^{-3}$ to $p = 10^{-4}$. Astrea-G remains as accurate as MWPM.

Astrea-G incurs an average decoding latency of about 131ns for $p = 10^{-3}$ and the worst-case latency of $1\mu s$. Astrea-G reduces the mean latency relative to Astrea by eliminating many high-weight syndrome bit pairs. Furthermore, Astrea-G may use the entire $1\mu s$ period for decoding because it continues iterating until the entire search space is exhausted. However, by prioritizing lower weight pairs first, Astrea-G converges on the MWPM early into the search.

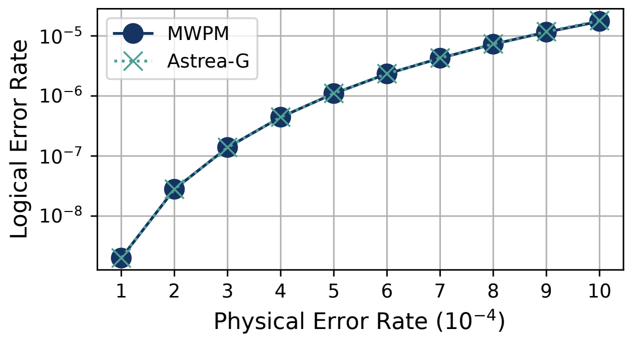

Figure 12: Logical error rates for MWPM and Astrea-G for physical error rates between 10^{-4} and 10^{-3} for $d = 7$. Astrea-G is as accurate as MWPM.

7.3 Analyzing Impact of the Weight Threshold

Astrea-G's latency depends on the Weight Threshold (W_{th}), which is responsible for constraining the search space. We use a default W_{th} of 7 (representing 100x lower probability than the logical error rate). Figure 13 shows the logical error-rate of Astrea-G normalized to an idealized MWPM implementation as W_{th} is varied from 4 to 8. At $W_{th} = 4$, Astrea-G's logical error rate is 3.0×10^{-5} compared to MWPM's 1.8×10^{-5}. As W_{th} is increased beyond 4, the performance of Astrea-G approaches the accuracy of idealized MWPM.

7.4 Extending Astrea-G to Distance 9 Codes

We examine Astrea-G's performance for larger distances, say $d = 9$. For this evaluation, we use 100 billion trials for each configuration (as the logical error rate is quite small). Figure 14 shows the logical error rate of Astrea-G and idealized MWPM as the physical error

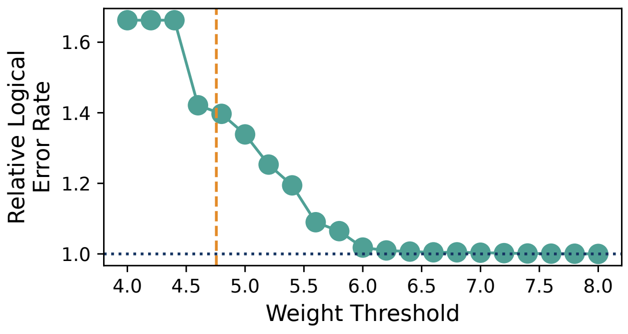

Figure 13: The impact of different Weight Thresholds for $d = 7, p = 10^{-3}$ on the logical error rate of Astrea-G relative to idealized MWPM decoding.

rate is varied from 10^{-4} to 10^{-3}. We observe that Astrea remains within 2.7× of MWPM's logical error rate, *which significantly outperforms prior work which are* 100× *or* 1000× *worse.* As the logical error rate decreases, Astrea-G moves closer to the performance of MWPM. The average decoding latency is 450ns for $p = 10^{-3}$ with the worst case being 1μs, thus meeting the real-time constraints.

Astrea-G Enables MWPM up-to Distance 9 in Real-Time

To the best of our knowledge, Astrea-G is the first practical decoder that can *decode surface codes up to distance* 9 *at a physical error rate of* 10^{-3} *in real time, while maintaining comparable accuracy to idealized MWPM decoding.*

7.5 Storage Overheads of Astrea-G

We analyze the SRAM overheads of Astrea-G for decoding X or Z stabilizers of a distance 7 and distance 9 codes in Table 6. Astrea-G's overheads are dominated by the GWT, while other data structures such as the priority queues only require a few KB. Data structures such as the LWT, priority queues, and MWPM Register have overheads that increase for larger Hamming weight. Consequently, for lower physical error rates and distance, Astrea-G's hardware overheads decrease.

Table 6: SRAM Overheads for Astrea-G

Component	$d = 7$	$d = 9$
Global Weight Table (GWT)	36KB	156KB
Local Weight Table (LWT)	512B	512B
Priority Queues	3.4KB	4.1KB
Pipeline Latches	2.3KB	2.9KB
MWPM Register	24B	30B
Total	42 KB	164KB

7.6 Syndrome Bandwidth Requirements

We briefly discuss the syndrome bandwidth requirements for Astrea-G for a $d = 9$ code at $p = 10^{-3}$. Every round, $9^2 - 1 = 80$ parity qubits are measured, resulting in 80 syndrome bits that must be transmitted to Astrea-G. However, Astrea-G must receive the syndrome and decode it within 1μs. If the syndrome is transmitted over the entire 1μs, then Astrea-G has no time to decode. Thus, the system must be provisioned with sufficient bandwidth. Table 7 shows the time to transmit at a particular bandwidth (MBps) and the impact on error-rate for Astrea-G. At lower bandwidth, when almost half the time is allocated for transmission, the relative error rate increases by 33%. However, having a bandwidth of 50 MBps provides a similar error rate as unlimited bandwidth. As syndromes are typically compressible, we can further employ Syndrome Compression to reduce bandwidth requirement [14].

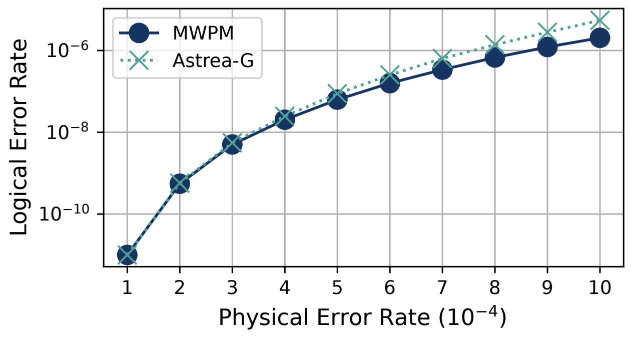

Figure 14: Logical error rates for MWPM and Astrea-G for $10^{-4} \leq p \leq 10^{-3}$ for $d = 9$. Astrea-G remains within 2.7x of MWPM's LER.

Table 7: Bandwidth Requirements for Astrea-G

Transmission Time (ns)	Bandwidth (MBps)	Relative LER
0	Unlimited	1.0×
50	200	1.0×
100	100	1.0×
200	50	1.0×
300	33	1.01×
400	25	1.08×
500	20	1.33×

7.7 FPGA Utilization and Latency

We synthesized and implemented Astrea-G with $F = 2$ and $E = 8$ on Xilinx's Zynq UltraScale+ FPGA. Table 8 shows the utilization and maximum clock frequency of Astrea-G on the FPGA. Astrea-G requires less than half of the resources on the FPGA, making a practical design for near-term QEC.

Table 8: FPGA Synthesis and Implementation, Astrea-G

LUT%	FF%	BRAM%	Max Freq. (MHz)
20.2	3.92	35.7	250

8 RELATED WORK

In this section, we discuss the related work and compare and contrast with Astrea as appropriate.

8.1 Decoders

Although error decoding has been studied for several years, only recently there has been a paradigm shift towards leveraging architectural optimizations to improve the latency of decoders for enabling real-time decoding [9, 13, 14, 33, 44, 46, 48, 58–60]. Ideally, the **Minimum Weight Perfect Matching (MWPM)** decoder is deemed as the gold standard for decoding surface codes and all decoders strive to achieve its accuracy [17–19, 21, 22]. Recently, there have been several fast implementations of MWPM such as **PyMatching** [32]. However, we observe that these implementations are only fast in the *average case* with worst-case latencies that make them untenable for real-time decoding. **LILLIPUT** implements MWPM decoding using Lookup Tables (LUTs) on FPGAs [13]. Although it uses commodity hardware, the size of the LUTs limit the scalability of this design to distance 5 with only two syndrome rounds. Other prior real-time decoders trade-off accuracy by relying on approximate techniques or specialized hardware to improve the decoding latency. The **NISQ+** decoder is a superconducting based design that leverages speedup from superconducting devices but has lower accuracy owing to its inability to tackle errors during syndrome extraction (only one syndrome round used for decoding) [33]. **QECOOL** and **QULATIS** makes similar trade-offs but only handle a limited number of measurement errors during syndrome extraction (uses fewer than d rounds for decoding distance d codes) [58, 60]. Furthermore, these three decoders rely on emerging technologies that limit their adoption in the near-term. The **AFS** decoder uses custom ASICs to decode using the approximate Union-Find decoding algorithm. Although this decoder can decode any arbitrary distances d using d syndrome rounds, its accuracy is much lower than the MWPM decoder [14]. The **Clique** decoder is an orthogonal approach that involves *hierarchical decoding*, which uses combination of decoders [48]. However, its reliance on software MWPM for "hard-to-decode" syndromes limits its applicability for real-time decoding and its inaccuracy negatively impacts the accuracy of the overall hierarchical decoder. Unlike Clique, **NEO-QEC** is a hierarchical decoder that relies on QULATIS instead of MWPM [59]. While its neural network pre-decoder improves the performance of QULATIS, NEO-QEC's accuracy still pales in comparison to MWPM. In contrast to prior work, Astrea enables real-time decoding with comparable accuracy to MWPM while using commodity hardware, making it ideal for practical adoption. Astrea scales up to larger distances than LILLIPUT, and is significantly more accurate than NISQ+, QECOOL, QULATIS, AFS, and Clique decoders.

8.2 Demonstrations of QEC

Quantum error correction is reaching a regime where there are demonstrations of small QEC codes on both trapped-ions and superconducting based systems [1, 11, 39, 50, 56]. The largest demonstrated surface codes so far corresponds to distance 5 on the latest edition of the Google Sycamore device. However, these experiment studies focus on demonstrating error suppression and have relied on offline decoding. To date, there has been no demonstration of surface codes with real-time decoding. FTQCs require real-time decoding to perform fault-tolerant gate operations on logical qubits. Thus, a successful demonstration of real-time decoding would mark a monumental milestone in the field of quantum computing.

As device error-rates exhibit spatio-temporal variation, a decoder must be capable of handling non-uniform error rates and error drift to remain accurate. To the best of our knowledge, prior real-time decoders like NISQ+, QECOOL/QULATIS, and AFS lack the flexibility for handling non-uniform error rates or reprogramming physical error rates [14, 33, 58]. Astrea and Astrea-G natively handles non-uniform error rates and error drift by virtue of its GWT because weights can be adjusted to account for non-uniform error rates and can further be re-programmed if drift occurs. Thus, Astrea is not only more accurate, but is also more flexible than prior works.

9 DISCUSSION

If an FTQC's decoder can decode errors accurately, then the code distance is determined by the gap between the desired application error rate and the physical error rate [22, 25]. However, if the decoder is *inaccurate*, then the FTQC must use a *larger code distance*, and thus more hardware resources, to compensate for the decoder's inaccuracy. Current trends in real-time decoding have favored approximate decoders which sacrifice accuracy to meet real-time constraints because MWPM is perceived to be too difficult to implement in real-time [9, 14, 33, 48, 58, 60]. However, approximate decoders incur logical error rates 100× or 1000× worse than MWPM and thus require significantly larger code distances compared to idealized MWPM decoding. Instead of optimizing solely for decoding speed at the expense of accuracy, decoders should optimize achieving the *best possible accuracy* because this can enable practical applications at lower code distances.

Astrea and Astrea-G challenge the widespread belief that *achieving real-time MWPM decoding is practically impossible*, and they are the *first* real-time decoders with comparable accuracy to MWPM, marking a significant advancement over prior work. Their orders of magnitude improvement over the state-of-the-art are an important step towards accurate real-time decoding, which is necessary for implementing promising applications with minimal resources.

10 CONCLUSION

We propose *Astrea*, which leverages the insight that the Hamming weight of most syndromes corresponding to small surface codes remains low enough for us to be able to exhaustively search for the MWPM in real-time. For distance 7 codes, Astrea achieves a logical error-rate virtually identical to the idealized implementation of MWPM while meeting the real time constraints. However, the Hamming weight of the syndromes increases with the code distance and physical error-rates which makes an exhaustive search infeasible. To tackle this challenge, we propose Greedy Astrea or *Astrea-G* that leverages the insight that not all possible matchings have the same probabilities of appearing in the MWPM. Astrea-G performs an efficient search that filters out high weight syndrome bit pairings and prioritizes low weight pairings during matching. This enables Astrea-G to decode up to distance 9 and an order of magnitude higher physical error-rate of 10^{-3} while meeting the real-time requirements.

ACKNOWLEDGMENTS

We thank the reviewers of ISCA-2023 for their suggestions and feedback. This work was funded in part by EPiQC, an NSF Expedition in Computing, under grant CCF-1730449. This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA.

A APPENDIX

A.1 Evaluating Astrea-G for Larger Distance

We present additional results for Astrea-G evaluated for the $d = 11$ surface code at $p = 1 \times 10^{-4}$ to understand how Astrea-G will scale to surface codes beyond the near-term. Because the logical error rate is quite small (less than one in a trillion) and is thus practically impossible to estimate through Monte Carlo sampling, we estimate the logical error rate as follows. For $1 \leq k \leq 20$, we generate syndromes corresponding to error events with k errors using several millions of trials. Then, we estimate the probability of failure for k errors ($P_f(k)$) using the millions of trials, and we estimate the probability of occurrence for k errors ($P_o(k)$) using our error model. Then, the logical error rate is approximated as in Equation (3).

$$LER = \sum_k P_f(k) \times P_o(k) \tag{3}$$

Using this methodology, we evaluate both MWPM and Astrea-G for $d = 7, 9, 11$ at $p = 1 \times 10^{-4}$ as shown in Table 9. Although Astrea-G is able to suppress errors at $d = 11$, it is about 17× worse than idealized MWPM in terms of logical error rate. While this is better than results observed in prior work, we note that having an order of magnitude worse LER is undesirable. Thus, accurate real-time decoding still remains an open problem for $d = 11$ and beyond, and closing the accuracy gap in this regime is necessary for implementing many promising applications and achieving quantum fault-tolerance at scale.

Table 9: Logical error rates at $p = 1 \times 10^{-4}$

d	MWPM LER	Astrea-G LER
7	4.6×10^{-10}	4.6×10^{-10}
9	1.2×10^{-11}	1.2×10^{-11}
11	1.7×10^{-14}	2.9×10^{-13}

B ARTIFACT APPENDIX

B.1 Abstract

The artifact contains the source code used to simulate and evaluate the designs proposed in this paper. We note that because Astrea is a limited version of MWPM, there is no explicit implementation of it provided. We have provided a cycle-level simulator for Astrea-G. The artifact provides information how to reproduce key results from the paper, namely the data presented in Section IV, Table 2 (Astrea's critical insight); Section VII, Figures 12 and 14 (MWPM and Astrea-G logical error rates); and Section VII, Table 7 (analysis of bandwidth requirements for Astrea-G).

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

- **Program:** astrea
- **Compilation:** cmake, g++-17, gcc-8
- **Hardware:** Multiple Intel Xeon Gold 6226 CPUs (2.7GHz)
- **Execution:** via Terminal
- **Metrics:** Logical Error Rate, Probability/Frequency
- **Output:** Text Files
- **Experiments:** Memory experiments with architectural simulation.
- **How much disk space required (approximately)?:** 16 MB
- **How much time is needed to prepare workflow (approximately)?:** A minute or so for compilation.
- **How much time is needed to complete experiments (approximately)?:** 48 hours with 1024+ cores for running experiments. About 5 minutes for plotting the data.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** 10.5281/zenodo.7755625

B.3 Description

B.3.1 How to access. The code is available on Zenodo at https://doi.org/10.5281/zenodo.7755625.

B.3.2 Hardware dependencies. Any computing cluster should be sufficient for to execute the relevant experiments. In our evaluations, we used 1024 cores (in total) to run the experiments within a reasonable amount of time. We recommend using 1GB of memory per core.

B.3.3 Software dependencies. The code is compiled using CMake v3.20.3, though slightly older versions should be fine and can be enabled by modifying `CMakeLists.txt`. The target compiler is g++-17, and the program requires OpenMPI v4.x.x. All other dependencies have been packaged with the code and are referenced through CMake. The code provided works on MacOS and Linux (Red Hat Enterprise v7.9).

The provided plotting script has been tested using Python v3.10.6, and the following packages are dependencies: matplotlib v3.6.1, numpy v1.23.4, scipy v1.9.2.

B.4 Installation

The following commands should create the `astrea` executable for running experiments.

```
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make -j8
```

B.5 Experiment workflow

An experiment can be run using the following syntax:

```
$ mpirun -np <X> ./astrea <output-file> <experiment-no> <arg1>
<arg2>
```

Where X is the number of processors available, and the arguments after `experiment-no` depend on the experiment. More details can be found in the following section. The data will be appended to the provided output file (this file will be made if it does not exist), which is saved in the directory `astrea/data`.

Plots can be generated using the `astrea_plot.py` script in the `astrea/python` folder. Explanation on how to use the script is discussed in the next section.

B.6 Evaluation and expected results

We explain how to reproduce the key results of our paper. For **Section IV, Table 2** which shows the probabilities of different Hamming weights, run the following for $d = 3, 5, 7$:

```
$ mpirun -np <X> ./astrea <output-file> 6 <d> 1e-4
```

This should generate a text file with entries HW, k where HW is the Hamming weight and k is the number of occurrences (out of 1B). We expect that the results should match the results in Table 2 within 2x (some deviation may occur due to randomness). We note that this command does not require multiprocessing because syndrome generation is typically fast, so running with *one* processor should return results within 10 minutes.

For **Section VII, Figure 12 and Figure 14** which show results for the logical error rate of Astrea-G, run the following command

```
$ mpirun -np <X> ./astrea <output-file> 1 <d>
```

For $d = 7$, 1B trials is run for each configuration from $p = 10^{-4}$ to $p = 10^{-3}$ at a step of 10^{-4}; 64 or 128 cores should be sufficient to finish the experiment within 12 hours. For $d = 9$, 100B trials is run for each configuration; 1024 cores should finish the experiment within 48 hours. We note that because the logical error rate increases for larger physical error rate, less trials can be run for the higher physical error rates. For example, $d = 9, p = 10^{-3}$ has a logical error rate in the regime of 10^{-6}, so 10^7 trials (10M) would suffice. To make this change, modify the code within the `experiment == 1` block in `astrea/src/main.cpp`. The output file should contain 10 lines (one per physical error rate). The first entry in the line is the code distance, the second is the physical error rate, the sixth entry is the logical error rate of idealized MWPM, and the seventh entry is the logical error rate for Astrea-G. The other entries are supplementary information for which more information can be found in `src/experiments.cpp`.

To plot the data, run the `plot_ler` function in `astrea_plot.py`: first argument is the experiment output file (should only have 10 entries), and the second argument is an output file for the plot. The results should be similar to Figure 12 and Figure 14 (with possible varation due to randomness).

For **Section VII, Table 7** which shows results for Astrea-G's error rate with respect to different bandwidth requirements, run the following command

```
$ mpirun -np <X> ./astrea <output-file> 12 9 500 1000 100
```

which runs Astrea-G with decoding time constraints of 500ns to 1000ns at a step size of 100ns for $d = 9, p = 10^{-3}$. The file should return six lines. For each line, the first entry is the code distance, the second entry is the physical error rate, the seventh entry is the logical error rate of Astrea-G, and the thirteenth entry is the time alloted for decoding (t). The transmission time for each entry is $1000 - t$, and the bandwidth (in MBps) is $80/(8 \times (1000 - t))$. The relative error rates reported in Table 7 are relative to the scenario with $t = 1000$ (ideal scenario with unlimited bandwidth/instant transmission). This experiment should be able to finish within 12 hours with 256 cores.

The data used in this paper has been provided for all experiments in the folder `astrea/data/examples`.

B.7 Experiment customization

The experiments provided have a reasonable amount of customizability, which can be explored by providing different inputs (i.e. different code distances). If further customization is needed to the experiments, see the code provided in `astrea/src/experiments.cpp`. If one seeks to modify Astrea-G, see `quarch/src/astrea/simulator.cpp`.

B.8 Notes

The version of Stim [24] provided in the artifact has been heavily modified for this paper. The official version of Stim is: https://github.com/quantumlib/Stim.

B.9 Methodology

Submission, reviewing and badging methodology:

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

REFERENCES

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

[2] Rajeev Acharya, Igor Aleiner, Richard Allen, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Juan Atalaya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Joao Basso, Andreas Bengtsson, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Yu Chen, Zijun Chen, Ben Chiaro, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben Curtin, Dripto M. Debroy, Alexander Del Toro Barba, Sean Demura, Andrew Dunsworth, Daniel Eppens, Catherine Erickson, Lara Faoro, Edward Farhi, Reza Fatemi, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, William Giang, Craig Gidney, Dar Gilboa, Marissa Giustina, Alejandro Grajales Dau, Jonathan A. Gross, Steve Habegger, Michael C. Hamilton, Matthew P. Harrigan, Sean D. Harrington, Oscar Higgott, Jeremy Hilton, Markus Hoffmann, Sabrina Hong, Trent Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Pavol Juhas, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Tanuj Khattar, Mostafa Khezri, Mária Kieferová, Seon Kim, Alexei Kitaev, Paul V. Klimov, Andrey R. Klots, Alexander N. Korotkov, Fedor Kostritsa, John Mark Kreikebaum, David Landhuis, Pavel Laptev, Kim-Ming Lau, Lily Laws, Joonho Lee, Kenny Lee, Brian J. Lester, Alexander Lill, Wayne Liu, Aditya Locharla, Erik Lucero, Fionn D. Malone, Jeffrey Marshall, Orion Martin, Jarrod R. McClean, Trevor Mccourt, Matt McEwen, Anthony Megrant, Bernardo Meurer Costa, Xiao Mi, Kevin C. Miao,

Masoud Mohseni, Shirin Montazeri, Alexis Morvan, Emily Mount, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Murphy Yuezhen Niu, Thomas E. O'Brien, Alex Opremcak, John Platt, Andre Petukhov, Rebecca Potter, Leonid P. Pryadko, Chris Quintana, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus, Christopher Schuster, Michael J. Shearn, Aaron Shorter, Vladimir Shvarts, Jindra Skruzny, Vadim Smelyanskiy, W. Clarke Smith, George Sterling, Doug Strain, Marco Szalay, Alfredo Torres, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Theodore White, Cheng Xing, Z. Jamie Yao, Ping Yeh, Juhwan Yoo, Grayson Young, Adam Zalcman, Yaxing Zhang, and Ningfeng Zhu. 2022. Suppressing quantum errors by scaling a surface code logical qubit. https://doi.org/10.48550/ARXIV.2207.06431

[3] Google Quantum AI. Accessed: June 19, 2021. Quantum Computer Datasheet. https://quantumai.google/hardware/datasheet/weber.pdf.

[4] Lucas Berent, Lukas Burgholzer, and Robert Wille. 2022. Software Tools for Decoding Quantum Low-Density Parity Check Codes. https://doi.org/10.48550/ARXIV.2209.01180

[5] Nikolas P Breuckmann, Christophe Vuillot, Earl Campbell, Anirudh Krishna, and Barbara M Terhal. 2017. Hyperbolic and semi-hyperbolic surface codes for quantum storage. *Quantum Science and Technology* 2, 3 (aug 2017), 035007. https://doi.org/10.1088/2058-9565/aa7d3b

[6] 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 366–382. https://doi.org/10.1145/3470496.3527417

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

[8] Earl Campbell, Ankur Khurana, and Ashley Montanaro. 2019. Applying quantum algorithms to constraint satisfaction problems. *Quantum* 3 (jul 2019), 167. https://doi.org/10.22331/q-2019-07-18-167

[9] Christopher Chamberland, Luis Goncalves, Prasahnt Sivarajah, Eric Peterson, and Sebastian Grimberg. 2022. Techniques for combining fast local decoders with global decoders under circuit-level noise. https://doi.org/10.48550/ARXIV.2208.01178

[10] Christopher Chamberland, Guanyu Zhu, Theodore J. Yoder, Jared B. Hertzberg, and Andrew W. Cross. 2020. Topological and Subsystem Codes on Low-Degree Graphs with Flag Qubits. *Phys. Rev. X* 10 (Jan 2020), 011022. Issue 1. https://doi.org/10.1103/PhysRevX.10.011022

[11] Zijun Chen, Kevin J Satzinger, Juan Atalaya, Alexander N Korotkov, Andrew Dunsworth, Daniel Sank, Chris Quintana, Matt McEwen, Rami Barends, Paul V Klimov, et al. 2021. Exponential suppression of bit or phase flip errors with repetitive error correction. *arXiv preprint arXiv:2102.06132* (2021).

[12] Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (sep 2018), 9456–9461. https://doi.org/10.1073/pnas.1801723115

[13] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 541–553. https://doi.org/10.1145/3503222.3507707

[14] 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)*. 259–273. https://doi.org/10.1109/HPCA53966.2022.00027

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

[16] Nicolas Delfosse, Vivien Londe, and Michael E. Beverland. 2022. Toward a Union-Find decoder for quantum LDPC codes. *IEEE Transactions on Information Theory* (2022), 1–1. https://doi.org/10.1109/TIT.2022.3143452

[17] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (Sep 2002), 4452–4505. https://doi.org/10.1063/1.1499754

[18] Jack Edmonds. 1965. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics* (1965), 125.

[19] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17 (1965), 449–467.

[20] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

[21] Austin G. Fowler. 2014. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time. arXiv:1307.1740 [quant-ph]

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

[23] Jay Gambetta. 2022. Quantum-centric supercomputing: The Next Wave of computing. https://research.ibm.com/blog/next-wave-quantum-centric-supercomputing

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

[25] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (apr 2021), 433. https://doi.org/10.22331/q-2021-04-15-433

[26] Craig Gidney, Michael Newman, Austin Fowler, and Michael Broughton. 2021. A Fault-Tolerant Honeycomb Memory. *Quantum* 5 (Dec. 2021), 605. https://doi.org/10.22331/q-2021-12-20-605

[27] Craig Gidney, Michael Newman, and Matt McEwen. 2022. Benchmarking the Planar Honeycomb Code. *Quantum* 6 (Sept. 2022), 813. https://doi.org/10.22331/q-2022-09-21-813

[28] Daniel Gottesman. 1997. Stabilizer codes and quantum error correction. *arXiv preprint quant-ph/9705052* (1997).

[29] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043* (1996).

[30] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters* 103, 15 (oct 2009). https://doi.org/10.1103/physrevlett.103.150502

[31] Oscar Higgott. 2021. PyMatching: A Python package for decoding quantum codes with minimum-weight perfect matching. arXiv:2105.13082 [quant-ph]

[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] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshan Ding, Massoud Pedram, and Frederic T. Chong. 2020. NISQ+: Boosting quantum computing power by approximating quantum error correction. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 556–569. https://doi.org/10.1109/ISCA45697.2020.00053

[34] Shilin Huang, Michael Newman, and Kenneth R. Brown. 2020. Fault-tolerant weighted union-find decoding on the toric code. *Phys. Rev. A* 102 (Jul 2020), 012419. Issue 1. https://doi.org/10.1103/PhysRevA.102.012419

[35] IBM. 2021. IBM Quantum breaks the 100-qubit processor barrier. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle.

[36] A Yu Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (dec 1997), 1191. https://doi.org/10.1070/RM1997v052n06ABEH002155

[37] Ian D. Kivlichan, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Wei Sun, Zhang Jiang, Nicholas Rubin, Austin Fowler, Alá n Aspuru-Guzik, Hartmut Neven, and Ryan Babbush. 2020. Improved Fault-Tolerant Quantum Simulation of Condensed-Phase Correlated Electrons via Trotterization. *Quantum* 4 (jul 2020), 296. https://doi.org/10.22331/q-2020-07-16-296

[38] Vladimir Kolmogorov. 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1, 1 (2009), 43–67.

[39] Sebastian Krinner, Nathan Lacroix, Ants Remm, Agustin Di Paolo, Elie Genois, Catherine Leroux, Christoph Hellings, Stefania Lazar, Francois Swiadek, Johannes Herrmann, et al. 2022. Realizing repeated quantum error correction in a distance-three surface code. *Nature* 605, 7911 (2022), 669–674.

[40] Argonne National Laboratory. 2018. INTRODUCTION TO QUANTUM ERROR CORRECTION. https://cpb-us-w2.wpmucdn.com/voices.uchicago.edu/dist/0/2327/files/2019/11/QECIntro.pdf.

[41] Andrew J. Landahl, Jonas T. Anderson, and Patrick R. Rice. 2011. Fault-tolerant quantum computing with color codes. https://doi.org/10.48550/ARXIV.1108.5738

[42] Joonho Lee, Dominic W. Berry, Craig Gidney, William J. Huggins, Jarrod R. McClean, Nathan Wiebe, and Ryan Babbush. 2021. Even More Efficient Quantum Computations of Chemistry Through Tensor Hypercontraction. *PRX Quantum* 2, 3 (jul 2021). https://doi.org/10.1103/prxquantum.2.030305

[43] Jessica Lemieux, Guillaume Duclos-Cianci, David Sé néchal, and David Poulin. 2021. Resource estimate for quantum many-body ground-state preparation on a quantum computer. *Physical Review A* 103, 5 (may 2021). https://doi.org/10.1103/physreva.103.052408

[44] Wang Liao, Yasunari Suzuki, Teruo Tanimoto, Yosuke Ueno, and Yuuki Tokunaga. 2023. WIT-Greedy: Hardware System Design of Weighted ITerative Greedy Decoder for Surface Code. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '23)*. Association for Computing Machinery, New York, NY, USA, 209–215. https://doi.org/10.1145/3566097.3567933

[45] Satvik Maurya, Chaithanya Naik Mude, William D. Oliver, Benjamin Lienhard, and Swamit Tannu. 2022. Hardware Efficient Neural Network Assisted Qubit Readout. arXiv:2212.03895 [quant-ph]

[46] Ramon 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 Trade-Offs. arXiv:2202.05741 [quant-ph]

[47] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5 (2014), 4213.

[48] Gokul Subramanian Ravi, Jonathan M. Baker, Arash Fayyazi, Sophia Fuhui Lin, Ali Javadi-Abhari, Massoud Pedram, and Frederic T. Chong. 2022. Have your QEC and Bandwidth too!: A lightweight cryogenic decoder for common / trivial errors, and efficient bandwidth + execution management otherwise. https://doi.org/10.48550/ARXIV.2208.08547

[49] Markus Reiher, Nathan Wiebe, Krysta M. Svore, Dave Wecker, and Matthias Troyer. 2017. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences* 114, 29 (2017), 7555–7560. https://doi.org/10.1073/pnas.1619152114 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1619152114

[50] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes, and R. P. Stutz. 2021. Realization of Real-Time Fault-Tolerant Quantum Error Correction. *Phys. Rev. X* 11 (Dec 2021), 041058. Issue 4. https://doi.org/10.1103/PhysRevX.11.041058

[51] T. R. Scruby, D. E. Browne, P. Webster, and M. Vasmer. 2022. Numerical Implementation of Just-In-Time Decoding in Novel Lattice Slices Through the Three-Dimensional Surface Code. *Quantum* 6 (May 2022), 721. https://doi.org/10.22331/q-2022-05-24-721

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

[53] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* (1999).

[54] Samuel C. Smith, Benjamin J. Brown, and Stephen D. Bartlett. 2022. A local pre-decoder to reduce the bandwidth and latency of quantum error correction. https://doi.org/10.48550/ARXIV.2208.04660

[55] Matthias Steffen, Jerry Chow, Sarah Sheldon, and Doug McClure. 2022. IBM Quantum's highest performant system, yet. https://research.ibm.com/blog/eagle-quantum-error-mitigation

[56] Neereja Sundaresan, Theodore J. Yoder, Youngseok Kim, Muyuan Li, Edward H. Chen, Grace Harper, Ted Thorbeck, Andrew W. Cross, Antonio D. Córcoles, and Maika Takita. 2022. Matching and maximum likelihood decoding of a multi-round subsystem quantum error correction experiment. https://doi.org/10.48550/ARXIV.2203.07205

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

[58] 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)*. 451–456. https://doi.org/10.1109/DAC18074.2021.9586326

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

[60] Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki, and Yutaka Tabuchi. 2022. QULATIS: A Quantum Error Correction Methodology toward Lattice Surgery. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 274–287. https://doi.org/10.1109/HPCA53966.2022.00028

[61] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, and Yuan Xie. 2022. A Synthesis Framework for Stitching Surface Code with Superconducting Quantum Devices. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/3470496.3527381