Realtime Compilation for Continuous Angle Quantum Error Correction Architectures

Sayam Sethi
The University of Texas at Austin
sayams@utexas.edu

Abstract

Quantum error correction (QEC) is necessary to run large scale quantum programs. Regardless of error correcting code, hardware platform, or systems architecture, QEC systems are limited by the types of gates which they can perform efficiently. In order to make the base code's gate set universal, they typically rely on the production of a single type of resource state, commonly T, in a different code which is then distilled and injected into the base code. This process is neither space nor time efficient and can account for a large portion of the total execution time and physical qubit cost of any program. In order to circumvent this problem, alternatives have been proposed, such as the production of continuous angle rotation states [1, 5]. These proposals are powerful because they not only enable localized resource generation but also can potentially reduce total space requirements.

However, the production of these states is non-deterministic and can require many repetitions in order to obtain the desired resource. The original proposals suggest architectures which do not actively account for realtime management of its resources to minimize total execution time. Without this, static compilation of programs to these systems will be unnecessarily expensive. In this work, we propose a realtime compilation of programs to these continuous angle systems and a generalized resource sharing architecture which actively minimizes total execution time based on expected production rates. To do so, we repeatedly redistribute resources on-demand which depending on the underlying hardware can cause excessive classical control overhead. We further address this by dynamically selecting the frequency of recompilation. Our compiler and architecture improves over the baseline proposals by an average of 4.5×.

1 Introduction

Current quantum computers very noisy and only able to reliably execute small programs. In order to combat high error rates, quantum error correction is necessary in order to execute large scale applications. Even still, as we scale, quantum resources will be relatively scarce and it is critical to develop real time management of these resources in order to maximize the value of available hardware. These resources include classical bandwidth for decoding [16], ancilla management for communication and decompositions [6], and the creation and use of special resource states called *magic states* [5, 14]. So far, the study of these resources has been concentrated around surface codes as a promising candidate error correction code for small to intermediate scale quantum hardware [7, 13] due to its limited hardware connectivity, high threshold, and well studied decoding procedures. While surface codes are not definitively the best choice of error correction code, for example qLDPC codes [3, 19] are a promising choice for high rate codes, they are fairly easy to work since they have well-defined logical operators via lattice surgery [8] and can serve as as testbed for different architectural designs.

Recently, several approaches apart from Magic State Distillation have been proposed [1, 4] which rather than requiring magic state factories to create one fixed type of Rz rotation instead propose methods to create analog rotation states $|m_\theta\rangle$, adopting the syntax from [1]. When injecting this state we can perform an arbitrary $Rz(\theta)$ rotation. This is especially powerful because it ideally reduces the total space requirement for producing non-Clifford gates since we can use ancilla local to the injection site to prepare the necessary $|m_\theta\rangle$ and also does not require us to distill hundreds or thousands of T gates per Rz rotation. So far, however, there is limited architectural support for this strategy specifically for the realtime management of the nondeterministic behavior of the $|m_\theta\rangle$ production. In the original proposal from [1] their STAR architecture provides a basic structure to demonstrate the technique's efficacy, but 1. limits state production to atomic STAR patches which limits parallel production when other space is unused and 2. does not directly adapt program execution as a function of the highly non-deterministic state production.

The primary contributions of this work are

1. An efficient compilation scheme for quantum error correction systems which support native continuous angle resource states. We improve over baseline proposals by an average of 4.5× in total program execution time, both quantum and classical delay costs accounted for.
2. An improved architecture for local resource state production which reduces total space (ancilla) requirements while simultaneously reducing the total runtime of programs on these systems.
3. Compilation which directly accounts for the inherent non-deterministic behavior of continuous angle resource state production; we introduce the notion of

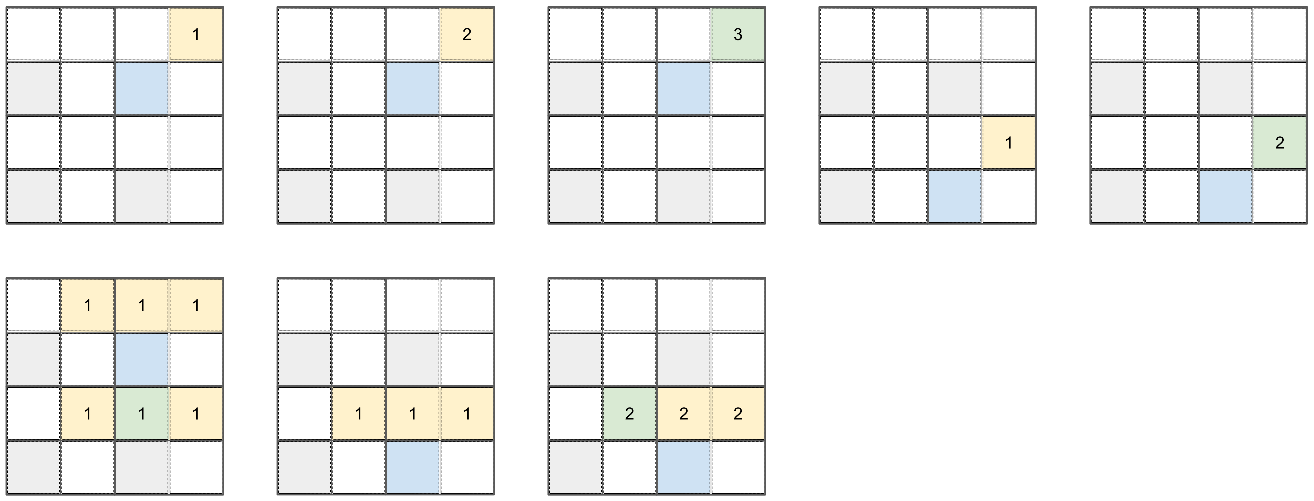


Figure 1. Sample execution patterns of STAR [1] (top) compared to our more dynamic and efficient resource management compiler (bottom). In STAR, atomic units of 2x2 grids are allocated containing 1 data qubit and 3 ancilla. Continuous angle rotation states $|m_\theta\rangle$ can be prepared locally and consumed within each STAR unit. However, ancilla use is locally restricted as well which means non-deterministic state production can take longer than necessary without utilizing parallel preparations with neighboring ancilla. Our compiler effectively manages ancilla for both state preparation and communication based on dynamically and in realtime computed expected free times for each resource and allow for multiple parallel preparations to minimize total expected execution time, here 6 preparations are attempted in parallel (bottom) as opposed to 1 (top), this essentially guarantees a single cycle to produce the necessary state. This leads to on average 4.5× improvement in total runtime.

real-time recompilation depending on the prior success of production and consumption of these states.

4. Real system measurement latencies restrict the amount and frequency of classical recompilation that can occur without incurring excessive idling on the quantum system; our compilation scheme easily adapts to any hardware platform and dynamically selects the frequency of recompilation the first of its kind to do so. This real-time control consideration is extensible to other QEC systems.

2 Background

2.1 Surface Codes

We summarize the necessary information for this work:

- Program qubits are assigned to a single $d \times d$ tile in the larger fabric
- The logical operation CNOT (Figure 2) occurs in **two** steps: a. measure the ZZ operator between the control and the ancilla followed by b. measure the XX operator between the ancilla and the target.
- Interactions between logical qubits use a contiguous path of ancilla qubits which must touch every interacting qubit
- A special type of multi-qubit interaction, or *Pauli-product measurement*, can in **one** step by interacting the P-edge of each interacting logical qubit to an intermediate ancilla channel contiguous between each, where P is the specific Pauli. For example, a ZZ Pauli product measurement can be done by interacting the Z edge of qubit 1 and the Z edge of qubit 2 to one contiguous block of ancilla. This can be done in 1 step regardless of distance between them, so long as the ancilla channel is contiguous.

2.2 Compilation: Physical vs. Logical

In this work, we want to clearly distinguish the compilation of programs at the physical level from the compilation of programs at the logical level. Specifically, for the surface code, we consider *physical* compilation to be the process by which the data and ancilla qubits and physical gates of the logical qubit are mapped, routed, and scheduled on the specific hardware [11, 17, 18]. For example, if the target hardware is trapped ions the exact execution of specific syndrome measurements is determined by physical compilation and is otherwise unimportant at the higher program level. In this work, we focus on *logical compilation* which refers to the mapping, routing, and scheduling of program qubits onto logical qubits [9, 13, 14]. In order to be platform agnostic, we consider code cycles during which an entire round of low-level physical operations are performed. Therefore, our

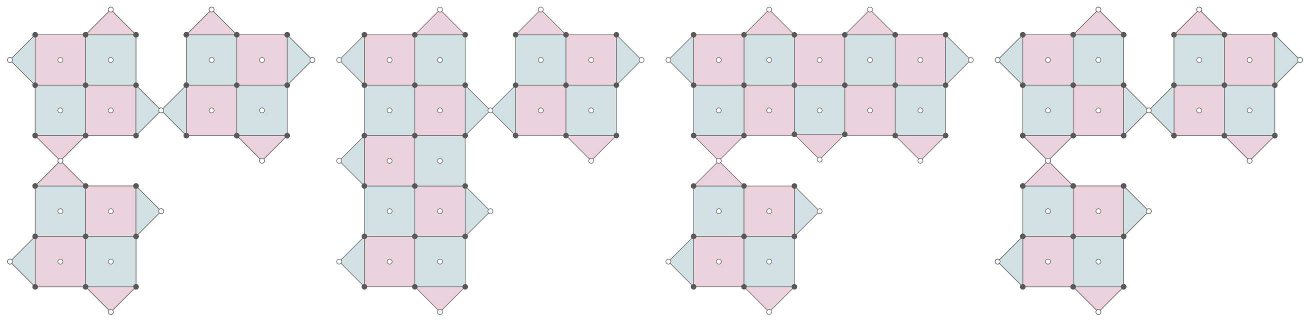

(a) Initial state **(b)** Merge step (1 cycle) **(c)** Split + Merge step (1 cycle) **(d)** Split step (0 cycles)

Figure 2. Execution of a CNOT gate in lattice surgery using exactly 2 steps. Here the CNOT is performed with a single ancilla patch in between. In general, this patch can be arbitrarily long and shaped so long as the correct boundaries are adjacent. We assume ancilla prepartion is done ahead of time.

basic units are logical qubits and code cycles as opposed to physical devices and execution time in seconds.

2.3 Clifford + T, Synthesis

All quantum error correction codes do *not* support a set of universal set of logical operators. Codes admit a partially complete set of gates; for example, surface codes can perform the Clifford gates easily which includes $\langle H, S, X, Z, CNOT \rangle$, the set generated by these gates. However, Cliffords are not universal and need to be augmented by some additional non-Clifford gate. The two most typical are T or Toffoli (CCX) gates and all gates in the input program must be synthesized into the gate sets Clifford+T or Clifford+CCX. For example, if an input program contains $Rz(\theta)$ rotations they must be decomposed into a finite sequence of $\{H, S, T\}$ as a function of approximation precision. These sequences can often take up the bulk of large-scale programs. For example, precision on the order of 10^{-8} or smaller is necessary and each require hundreds if not thousands of gates. Because T is not native, it must be prepared remotely which requires a large amount of space and time to prepare.

3 Related Work

Recent works [1, 5] have proposed fault-tolerant architectures to prepare arbitrary small angle rotation gates with low overheads and small logical error rates. The typical architecture for QEC systems is to have the primary "computational" code which maintains all data qubits and external "factory" regions which produce special resource states of a single variety, usually T for surface codes. It's been acknowledged that T count and depth dominate resource costs both space and time [14] and alternative proposals have appeared including code switching [2], higher dimensional codes [10] and most recently these "small-angle" synthesis procedures [1, 5] which propose repeat-until-success (RUS) procedures for the production of *arbitrary* magic states $|m_\theta\rangle$ which can be injected to perform $Rz(\theta)$ directly without the need for expensive decompositions or distillation with varying success. While perhaps not guaranteed to be the way forward, the total space-time cost of these procedures is appealing for near and intermediate term demonstrations of error correction by severely cutting down on total physical qubit requirements that would be necessary for distillation factories. These works propose simple versions of architectures which support their RUS strategies, but they are limited in scale and lack a full compiler to optimize for the non-deterministic behavior of their techniques, which even in other literature has gone largely ignored.

In this work, we focus primarily on the technique and architecture proposed in [1] which uses a [[4, 1, 1, 2]] error *detection* code to produce the $|m_\theta\rangle$ which can be embedded into the larger surface code architecture multiple times as in Figure 4. They give three examples of simple architectures which localize the production of these states: 1. STAR, a 2x2 grid of surface code tiles 1 of which is data and 1 of which produces the resource state and 2 ancilla used for communication, 2. Compact STAR, a 3x1 grid with 1 data and 2 ancilla and 3. Compressed STAR, a 2x1 grid with 1 data and 1 ancilla. Their atomic abstraction is wholly unnecessary and a more complete compiler (this work) can better manage ancilla to both create local and remote magic states and allocate ancilla for communication. We can blend and overlap these tiles in our larger fabric. Each of these can be found in Figure 3.

4 Realtime Compilation for Continuous Angle Rotation Architectures

In this section, we discuss the compilation of high level programs to surface code architectures which support Clifford+Rz gates as opposed to the traditional Clifford+T. We assume all programs have already been synthesized into the appropriate gate set. In typical distillation architectures, factories are bulky and hard to intersperse amongst the data

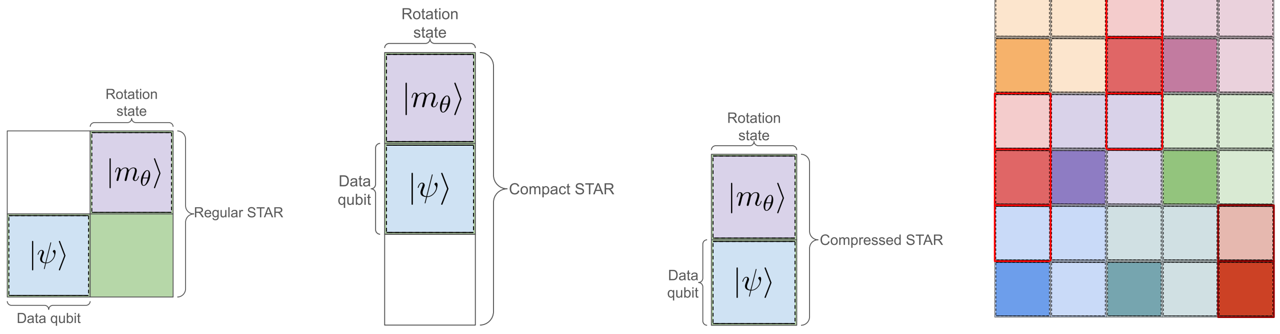


Figure 3. Different patches proposed in the STAR architecture and the grid on the right shows how we 'insert' the compact and compressed patches into a 2D grid of regular STAR patches. The different colours show the regular STAR patch with the darker colour indicating the data qubit and the lighter colours indicating ancilla qubits. The patches with red outline are the compact and compressed patches.

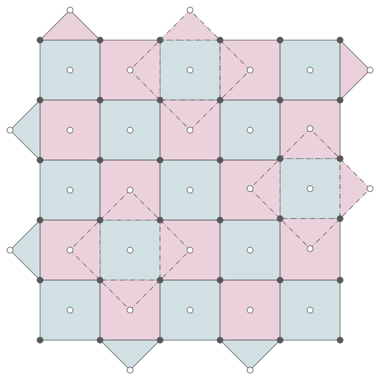


Figure 4. Parallel preparation of the $|m_\theta\rangle$ state within a surface code patch of distance 5

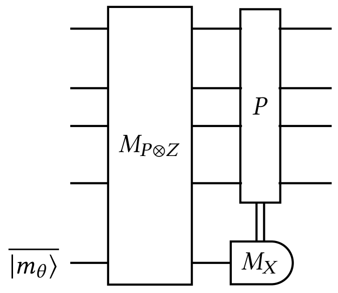


Figure 5. A Pauli-product measurement circuit

qubits. Therefore, resource states are prepared and stored remotely on the boundaries, only to be routed in for consumption as necessary. In contrast, in small angle rotation architectures, only small seeds of $|m_\theta\rangle$ states are prepared and expanded locally into surface code qubits. It takes at most a single logical patch for preparation at the cost of additional uncertainty in its preparation time. In prior work, atomic units of data qubits and ancilla for the preparation of single qubit gates are prepared. We, however, propose a much more flexible architecture which allows ancilla to be reused and allocated for various rotations dynamically.

4.1 Execution of $Rz(\theta)$ Rotation Gate

As discussed in Section 3, the $Rz(\theta)$ rotation gate is executed in two steps, a. preparation of an ancilla qubit in the state $|m_\theta\rangle$, and b. injection of the $|m_\theta\rangle$ state into the data qubit and measurement. If the measurement output signifies a failure (with probability 1/2), a correction $Rz(2\theta)$ would be required. If this correction fails, another correction gate $Rz(4\theta)$ would be required and so on, in a Repeat-Until-Success (RUS) manner. Specifically, failed injection means a $Rz(-\theta)$ was prepared so a $Rz(2\theta)$ correction would yield the proper rotation. However, since $Rz(2\theta)$ is likely a non-Clifford we must repeat. In general, if an $Rz(2^k\theta)$ injection fails we require $Rz(2^{k+1}\theta)$ correction. Every injection fails with probability 1/2, hence

$$\mathbb{E}[\text{Num. Injections}] = \sum_{k=1}^{\infty} k \cdot \Pr[\text{k-1 Failures, 1 Success}] = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$$

In the case where $Rz(2^k\theta)$ is a Clifford for some k (for example consider T or $\sqrt{T}$), this expectation will be < 2 since it will no longer require an injection step. There are two potential strategies for injection, summarized in Table 1, one using a CNOT (Figure 6) and one using a ZZ pauli-product measurement (Figure 5 with $P = Z$). In either case, the compiler must allocate a contiguous block of ancilla. Our compiler reserves ancilla and generates a schedule for each ancilla (either dynamically or statically) determining when its free; we simply require that some path between $|m_\theta\rangle$ and target all be free at some time.

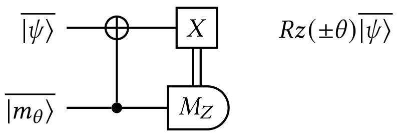


Figure 6. A Rotation gate injection circuit

Parameter	CNOT	ZZ
Exposed edge	X	Z
Number of ancillas required	2	1
Lattice surgery cycles needed for injection	2	1

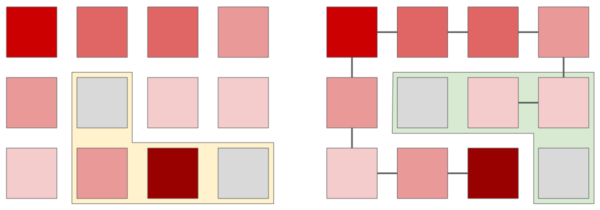
Table 1. Difference between the two injection strategies

The preparation of the rotation states $|m_\theta\rangle$ is itself non-deterministic and is prepared in an ancilla patch. Consequently, assigned ancilla are claimed for an indeterminate number of cycles until the state is prepared correctly. Different states $|m_\alpha\rangle$ and $|m_\beta\rangle$ require disjoint ancilla for preparation. Multiple ancilla can be assigned for the preparation of any individual state and any additional successful preparations can be discarded if necessary. The number of ancilla dedicated to the production of a particular state $|m_\theta\rangle$ can dynamically change; for example if n ancilla are assigned in cycle 1 and each fails and some m of these ancilla are needed for other operations, we can reclaim them and in the next cycle try to prepare the state using $n - m > 0$ ancilla. Non-deterministic preparation implies that the exact cycle when consumption can occur is unknown a head of time and motivates eager preparation.

4.2 Execution of CNOT Gate

Lattice surgery on surface codes gives us a convenient way to perform CNOT gates between two data qubits that can be arbitrarily far apart in constant time (taking 2 lattice surgery cycles). The only requirements for the execution is that there should exist a path of ancilla qubits connecting the control to the target (via at least one ancilla qubit), and the path should be from the Z edge of the control qubit to the X edge of the target qubit. It might be possible that such a path does not exist, either due to the ancillas being occupied for another gate operation, or the required edges (X or Z edges) of the data qubit not having any neighboring ancillas (see Section 4.3 for an example of this situation). In such cases, the compiler would need to insert an edge-rotation gate to orient the correct edge of the qubit onto the chosen path. This gate operation requires one free ancilla qubit and takes 3 lattice surgery cycles.

The quality and speed of the algorithm used for path computation is thus an essential consideration for the compilation strategy. For example, Figure 7 shows two different selection algorithms. Both algorithms choose the *shortest*



(a) Shortest path **(b)** Path with earliest start time

Figure 7. Different CNOT routing strategies between the two gray data qubits. The darkness of the ancilla qubit indicates the activity of the ancilla.

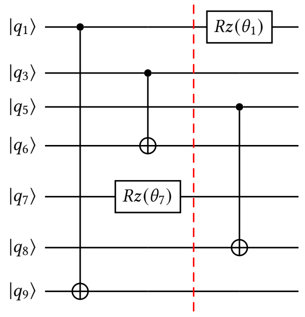


Figure 8. Example circuit for execution of the static compiler shown in Figure 9

path, however the naive selection of shortest may potentially lead to selecting an ancilla that is being used by a lot of other gate operations and this might lead to delayed start of the CNOT execution since the ancilla might not be free until much later.

4.3 Variable Ancilla Availability

A flexible architecture is considered with a fraction of the data qubits residing in a *compact patch* and the other fraction of the data qubits make up the *regular patch*. Such an architecture also motivates the need to share the ancillas between the qubit patches due to limited resources available to each patch by itself. This allows for better consideration of an actual system with finite resources and non-uniform resource allocation. The compiler needs to take the locally available and shared resources into account when determining which ancillas to use for different operations and how to maximize parallel computation.

4.4 Static Compilation for Continuous Angle Rotation Architectures

We consider a static compilation strategy that divides the circuit into layers of gates that can be executed in parallel. The input circuit is divided into layers by iterating on the gates in the order of execution and reserving the ancillas for exactly one gate operation per layer. If a particular gate

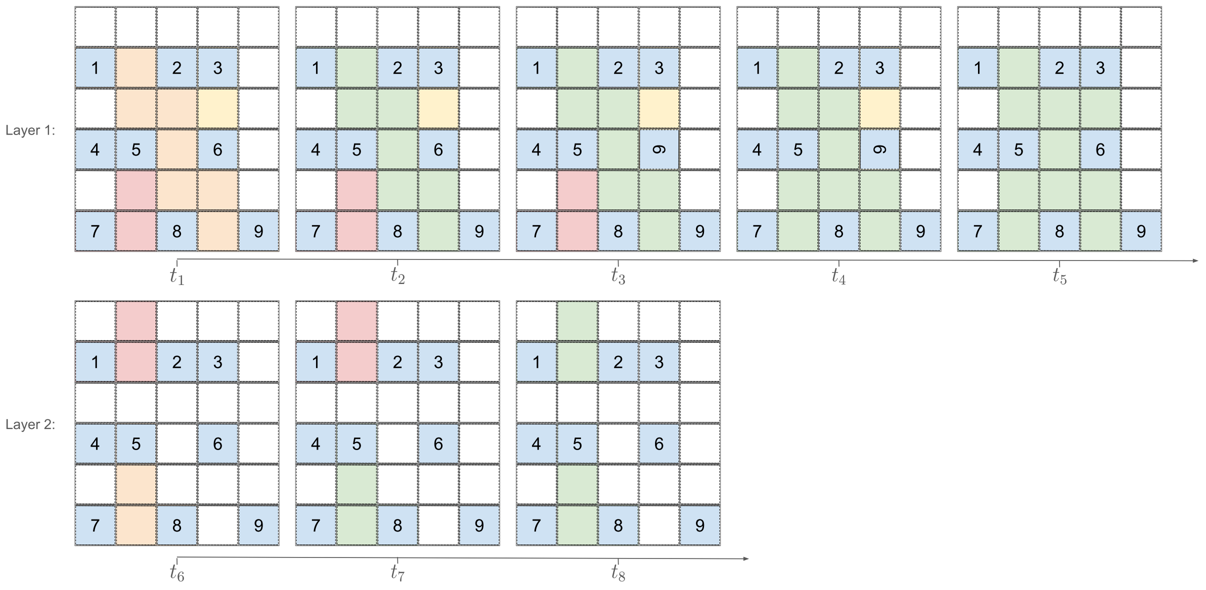


Figure 9. Execution begins at time t_1. For layer 1, the ancillas being used for CNOT between q_1 and q_9 are colored in orange, CNOT between q_3 and q_6 are colored in yellow, and Rz on q_7 are colored in red. Note that the CNOT between q_3 and q_6 requires to edge-rotate q_6 to orient the edges correctly. Completion of the gate execution is indicated by the ancillas turning green. For layer 2, the ancillas being used for the CNOT between q_5 and q_8 are colored in orange and the Rz on q_1 are colored in red.

cannot be executed in the current layer because of lack of ancilla resources, the compiler attempts to execute the gate in the next layer.

For the $Rz(\theta)$ gates, we only consider a single ancilla qubit for the preparation, depending on the patch configuration and the exposed edges. For the execution of CNOT gates, the shortest path between the control and target is chosen. While computing the shortest path, only the ancillas that have not been reserved for the current layer are considered. If the need for an edge-rotation is found for the chosen path, edge-rotation gates are inserted before and after the CNOT gate. The rotation gates are inserted after the CNOT gate to ensure that the correct edges continue to remain exposed for the $Rz(\theta)$ gate injection. Choosing the shortest path helps to maximize the parallelism in each layer since the next gate has more ancillas available to choose from. If the control and target cannot be connected, then gate is skipped in the current layer. Figure 9 shows an example execution for the circuit shown in Figure 8.

5 Dynamic Compilation

In this section, we discuss our dynamic compilation strategy that aims to minimize the total program execution time by executing the gate operations as soon as possible. The performance improvements of this strategy are based on two major observations:

1. Multiple ancillas can be prepared in parallel and the preparation can begin before the actual $Rz(\theta)$ gate operation can begin.
2. The expected 'free' time of the ancilla qubits, that is, the time when the ancilla qubits are going to be available for connecting the control and the target, can be used to make instantaneous decisions when determining the best path for the CNOT gate execution

The first observation motivates the need to efficiently manage and allocate the ancillas available to the different gates and data qubits. Since the preparation and injection for the $Rz(\theta)$ rotation gates is done locally, it makes the management feasible with a negligible classical overhead and thus it can be done during runtime without any performance penalty. The exact techniques used are discussed in Section 5.1.

Finding the best path in a graph (the 2D grid of the data qubits and ancillas) such that it can begin execution at the earliest is equivalent to solving the minimax path in a graph. This problem has a polynomial time solution which is obtained by computing the Minimum Spanning Tree and connecting the control and target on this tree. However, the complexity of this algorithm is $O((N+E)\log E)$ where N is the number of logical qubits in the grid and E is the number of edges. Since we are considering a 2D grid, $E = O(N)$ and therefore the time complexity can simply be written as $O(N \log N)$. Thus this approach would scale with the size of the grid and would be inefficient.

5.1 Intelligent Ancilla Resource Management

The execution of any gate can be broadly broken down into two steps, preparation and execution. The preparation step only requires the associated ancilla qubits to be free while the execution step requires the data qubit(s), prepared ancilla qubit(s) and any ancilla qubit connecting these qubits to be free. Thus, if we consider the timeline of execution of the gate, the preparation can be done asynchronously. The execution step may or may not require more ancillas apart from the ancilla that was used in the preparation step. Also some gates might not have any preparation step at all.

This is especially important for the dynamic compilation of the $Rz(\theta)$ rotation gates. Consider a single ancilla qubit being used for the preparation of the $Rz(\theta)$ gate. For the ancilla and data qubits, we define the parameters t_θ, T_d, T_a, T_b, T_i to understand the dynamic compilation step. t_θ is the expected time it takes to prepare the ancilla qubit in the state $|m_\theta\rangle$. The time when the data qubit is free and can participate in the execution of the gate operation is denoted by T_d. Similarly, the time when the ancilla qubit is free and it can participate in the preparation of the gate operation is denoted by T_a. In an ideal scenario, we would want to begin the preparation of the ancilla qubit at time $T_b = \max(T_d - t_\theta, T_a)$. T_i is the time during the execution of the program when it can be deterministically asserted that the data qubit is going to be free at time T_d. Figure 10 shows an example execution and relevant timestamps.

5.1.1 Queue for Ancillas. Another way to reduce the expected execution time of the $Rz(\theta)$ gates is by preparing multiple ancilla qubits in parallel. To make it easy to keep a track of the gate operations that the ancilla qubits will assist, either during the preparation or during the execution, each ancilla qubit maintains a queue of the gate operations that the ancilla has prepare/execute for. The compiler then looks at the head of the queue and begins the required steps necessary for the gate operation. Similarly, when the compiler chooses the relevant ancilla qubits for a particular gate's operations, it pushes the gate to the queue of all the associated ancillas. Each queue entry needs to store some relevant information about the ancilla's role in the preparation and/or execution. In the case of a Rz gate, it might be possible that the ancilla is only being involved for execution, however, it might be associated with two different preparing ancillas. In such scenarios, it might be possible that the ancilla which is not the head of the queue gets prepared first. The compiler then removes all entries in queue that appear before this entry from the queue for this gate even if they appear in the middle of the queue. Most of the queue operations can be performed independently irrespective of the state of the queue of the other ancillas. Therefore, the queues for all ancillas can be maintained in parallel using small computation units for each ancilla with minimal support for coherence.

5.2 Routing Strategy

Now that we have a convenient way to claim ancillas for the gate operations, all that remains is to efficiently determine the path of ancillas that will be used to execute any CNOT gate. To get an estimate of the activity of each ancilla during runtime, the MST algorithm is run on the sub-grid comprising of the ancilla qubits every k cycles, where k is a modifiable parameter. Assuming that the computation of the MST takes about p cycles, the results of the MST computation are *delayed* by p cycles. If we use the weight of each ancilla during the MST computation as the expected time when the ancilla finishes execution, the information would be stale by the time the MST is computed. Thus, instead we use the expected activity of the ancilla in the last p cycles to capture the *heat* (activity) of each ancilla. Now when a CNOT gate requires to be executed, it just looks at the latest available MST and computes up to 4 different paths. These paths computed are the paths from the $Z \to X$, $X \to Z$, $X \to X$, $Z \to Z$ edges of the control $\to$ target respectively.

Note that all paths might not always exist. For the existing paths, the earliest expected start time is computed after accounting for the edge-rotation of the data qubits if the required edges do not align. Once the best path is chosen, the CNOT gate is added to the queue of all the ancilla qubits on the path. Once this CNOT gate reaches the head of all the queues, the execution of the CNOT gate begins. However this approach can lead to increasing load on the same set of ancillas since the same MST is used for multiple CNOT gate executions. This load can be controlled by tuning the parameter k. However, reducing the parameter k to a very small number would increase the space required for storing all partially/full computed MSTs. At any point of time, there would be p/k MST computations going on and thus the space required to keep a track of all MSTs would be $O(p/k \cdot N)$ since storing a single tree requires $O(N)$ space. Figure 11 shows an example execution of the delayed MST computation.

6 Evaluation

We randomly generated a 2D grid with 50% probability of the data qubit being a compact or compressed patch. For the static compiler, the compilation was done before the execution. For the dynamic compiler, compilation was done on-the-fly and the compiler was run for different parameters. We used the QASM Benchmarks from [12] by choosing circuits with significant number of Rz gates. We transpiled these benchmarks on Qiskit[15] to the basis gate set of $X, H, Rz, CNOT$ at an optimization level of 2 and then simulated the execution of the assembled circuits. We tested the circuit with different values of k. For the case of $k = 0$, i.e., where we compute the MST for each CNOT operation, we introduced stalls during the execution of the circuit. For all the executions, we assume p (number of cycles it takes to

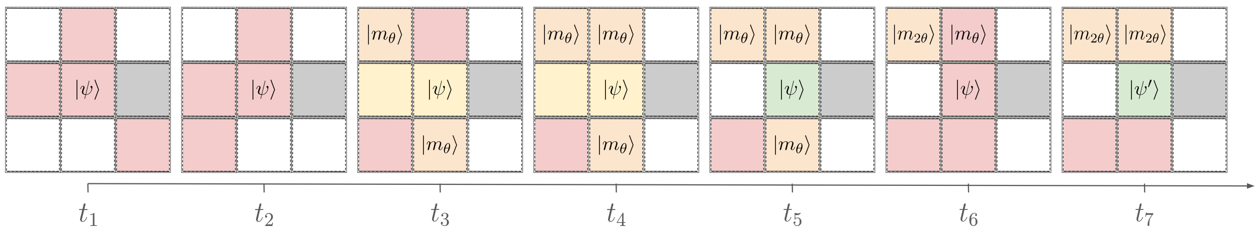


Figure 10. An example execution for an $Rz(\theta)$ gate to be executed next on the middle data qubit. The qubit in grey is another data qubit. Time steps t_1 and t_2 show execution times when different sets of ancilla qubits are busy. $T_i = t_3$ is when it is determined that the data qubit would be free at time $T_d = t_5$ and thus we start preparing all possible ancilla qubits in the neighborhood. Another ancilla qubit becomes free at time $T_a = t_4$ and we start preparing it. At t_6, one ancilla qubit succeeds in preparation and thus the injection begins, simultaneously we start preparing ancillas in the state $|m_{2\theta}\rangle$. The injection completes at time t_7 and $|\psi'\rangle = Rz(-\theta)\,|\psi\rangle$, thus we need to perform the correction gate.

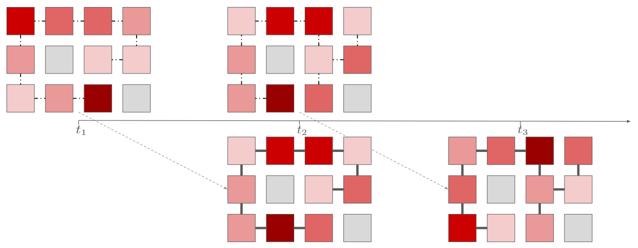


Figure 11. Timeline showing the MST computation procedure. The MST computation begins at t_1 and the result is available by t_2. Another MST computation begins at t_2 and its result is available by t_3. For simplicity, here $k = p = t_2 - t_1 = t_3 - t_2$

compute the MST) to be 100. Figure 12 contains the plot of execution times for physical qubit error rate 10^{-3} and a code distance of 7.

We observe a significant improvement in the execution times for the case of dynamic compiler, even after accounting for the stalls due to MST computation. The improvement is due to maximum utilization of the ancillas during the preparation phase and minimal waiting time for each gate operation. This can also be observed from the heatmap of the ancillas for the total activity in 1000 cycles as shown in Figure 13. Unlike the static case, the ancillas are active for almost half of the epoch duration. It was also observed that the the performance improvements are maintained as the physical error rates are reduced.

7 Conclusion

Fault-tolerant quantum architectures are still far from being realised on physical systems. Various error correction techniques have been proposed, surface codes being one of them. The surface code architecture does not natively support non-Clifford gates and thus require specialized support for such operations. Continuous rotation angle architectures have been proposed that allow for localized, low-latency ancilla preparation in the required states. However, this leads to the problem of supporting dynamic ancilla allocation to minimize runtime. Our work tackles this problem and we propose a dynamic compilation technique that utilizes the variable ancilla availability and activity to perform efficient allocation for different gate operations in parallel, thus minimising the total program execution time. We improve significantly over the basseline proposals while simultaneously minimizing classical overhead for realtime recompilation.

References

[1] Yutaro Akahoshi, Kazunori Maruyama, Hirotaka Oshima, Shintaro Sato, and Keisuke Fujii. Partially fault-tolerant quantum computing architecture with error-corrected clifford gates and space-time efficient analog rotations. *arXiv preprint arXiv:2303.13181*, 2023.

[2] Jonas T Anderson, Guillaume Duclos-Cianci, and David Poulin. Fault-tolerant conversion between the steane and reed-muller quantum codes. *Physical review letters*, 113(8):080501, 2014.

[3] Sergey Bravyi, Andrew W Cross, Jay M Gambetta, Dmitri Maslov, Patrick Rall, and Theodore J Yoder. High-threshold and low-overhead fault-tolerant quantum memory. *arXiv preprint arXiv:2308.07915*, 2023.

[4] Hyeongrak Choi, Frederic T Chong, Dirk Englund, and Yongshan Ding. Fault tolerant non-clifford state preparation for arbitrary rotations. *arXiv preprint arXiv:2303.17380*, 2023.

[5] Yongshan Ding, Adam Holmes, Ali Javadi-Abhari, Diana Franklin, Margaret Martonosi, and Frederic Chong. 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)*, pages 828–840. IEEE, 2018.

[6] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T Chong. Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 570–583. IEEE, 2020.

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

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

[9] Fei Hua, Yanhao Chen, Yuwei Jin, Chi Zhang, Ari Hayes, Youtao Zhang, and Eddy Z Zhang. Autobraid: A framework for enabling efficient

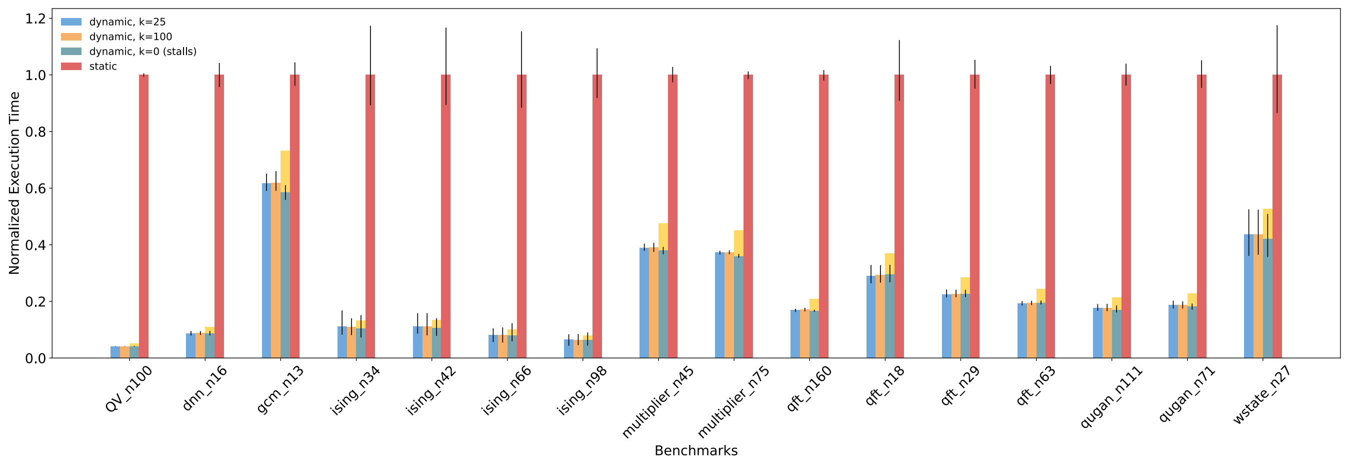
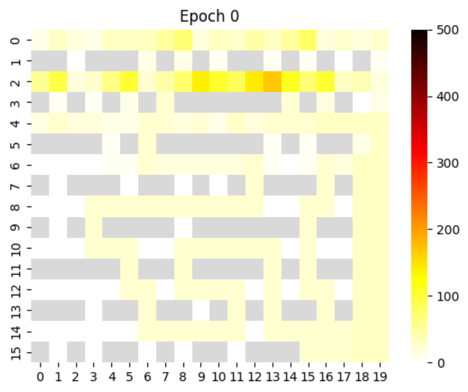
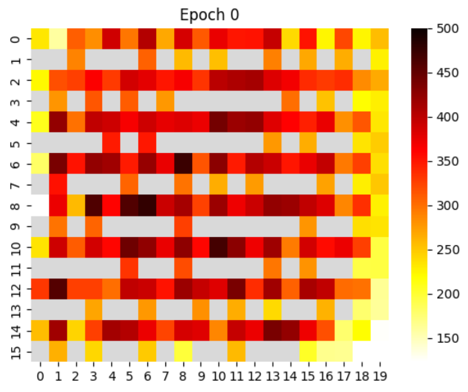


Figure 12. Normalized average execution time of different compilers for a variety of benchmarks (the bar in yellow shows the classical stalls)



(a) Static compiler



(b) Dynamic compiler ($k = 25$)

Figure 13. Heatmap of the ancilla qubits for different compilers for the first 1000 cycles for the QV_n100 benchmark (the greyed blocks are the data qubits)

surface code communication in quantum computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 925–936, 2021.

[10] Aleksander Marek Kubica. *The ABCs of the color code: A study of topological quantum codes as toy models for fault-tolerant quantum computation and quantum phases of matter.* PhD thesis, California Institute of Technology, 2018.

[11] Tyler LeBlond, Ryan S Bennink, Justin G Lietz, and Christopher M Seck. Tiscc: A surface code compiler and resource estimator for trapped-ion processors. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1426–1435, 2023.

[12] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation, 2022.

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

[14] Daniel Litinski. Magic state distillation: Not as costly as you think. *Quantum*, 3:205, 2019.

[15] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[16] Gokul Subramanian Ravi, Jonathan M Baker, Arash Fayyazi, Sophia Fuhui Lin, Ali Javadi-Abhari, Massoud Pedram, and Frederic T Chong. 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*, pages 88–102, 2023.

[17] Joshua Viszlai, Sophia Fuhui Lin, Siddharth Dangwal, Jonathan M Baker, and Frederic T Chong. An architecture for improved surface code connectivity in neutral atoms. *arXiv preprint arXiv:2309.13507*, 2023.

[18] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, and Yuan Xie. Mapping surface code to superconducting quantum processors. *arXiv preprint arXiv:2111.13729*, 2021.

[19] Qian Xu, J Ataides, Christopher A Pattison, Nithin Raveendran, Dolev Bluvstein, Jonathan Wurtz, Bane Vasic, Mikhail D Lukin, Liang Jiang, and Hengyun Zhou. Constant-overhead fault-tolerant quantum computation with reconfigurable atom arrays. *arXiv preprint arXiv:2308.08648*, 2023.