Orchestrating Multi-Zone Shuttling in Trapped-Ion Quantum Computers

Daniel Schoenberger[1] Robert Wille[1,2,3]

[1] Chair for Design Automation, Technical University of Munich, Germany
[2] Munich Quantum Software Company GmbH, Germany
[3] Software Competence Center Hagenberg GmbH, Austria
daniel.schoenberger@tum.de, robert.wille@tum.de

***Abstract*—Trapped-ion quantum computers are a promising platform, offering high-quality qubits with long coherence times and high-fidelity gate operations. The Quantum Charge Coupled Device (QCCD) architecture provides a scalable blueprint by leveraging the ability to shuttle ions between distinct zones. However, realizing such architectures in practice requires software support to manage ion movement across multi-zone layouts. In this work, we propose a compilation strategy for QCCD architectures with multiple processing zones located outside a grid-type memory zone. Unlike previous approaches that treat processing zones as black-boxes, our method explicitly models their structural constraints, enabling optimized ion movement to and through them. It combines qubit partitioning with dependency-aware gate selection to reduce inter-zone shuttling while enabling simultaneous gate execution. We implemented the method in an open-source tool and empirically demonstrated its effectiveness across several QCCD layouts, laying a foundation for the compilation of multi-zone trapped-ion systems.**

I. Introduction

In recent years, quantum computing has made remarkable progress in scaling and improving available hardware. Large-scale quantum computing promises transformative potential for cryptography [1], optimization [2], and quantum simulations [3]. As these research efforts shift from prototypical implementations to more scalable architectures, transitioning beyond the Noisy Intermediate-Scale Quantum (NISQ) era becomes the central goal. Achieving reliable large-scale quantum computation will require further substantial progress in not only hardware but also in developing the necessary software to operate, design, and build new devices.

Alongside superconducting [4], neutral atom [5], [6], and recent candidates such as optical quantum computers [7], trapped-ions stand out due to long coherence times, high-fidelity gates, and the ability to shuttle ions, which enables flexible qubit connectivity while minimizing additional wiring. In particular, the Quantum Charge Coupled Device architecture has shown promise by leveraging ion movement to employ multiple optimized zones in scalable designs.

QCCD-based devices have already been experimentally demonstrated [8]–[10] and experiments on QCCD processors have shown promise for fault-tolerant quantum computing. For instance, [11] has demonstrated how logical error rates can be brought below physical error rates through efficient encoding and error correction on a QCCD device.

Despite these advances, significant challenges remain in orchestrating quantum operations across increasingly complex QCCD systems. Designing scalable QCCD architectures requires efficient and automated ion-shuttling schedules that minimize decoherence and operational overhead, especially when multiple processing zones are incorporated. While prior work has addressed shuttling between single zones and focused on shuttling within the memory zone, the integration and coordination of multiple processing zones introduce additional constraints and orchestration complexity.

In this paper, we address these challenges by presenting a comprehensive compilation strategy tailored to QCCD devices with multiple external processing zones. Unlike approaches that treat these zones as black-boxes, we explicitly model their internal structure. Our compilation framework integrates qubit partitioning with dependency-aware gate selection, reducing the need for shuttling between processing zones and promoting parallel gate execution. We implement our strategy in an open-source compilation tool as part of the *Munich Quantum Toolkit* (MQT) [12] at https://github.com/cda-tum/mqt-ion-shuttler and demonstrate its efficacy through empirical evaluations across several representative QCCD layouts.

The remainder of this paper is structured as follows: Section II provides background information; Section III details the general idea of the compilation framework; Section IV elaborates on the shuttling methods; Section V presents the orchestration strategy; Section VI describes empirical evaluations; and Section VII concludes the paper.

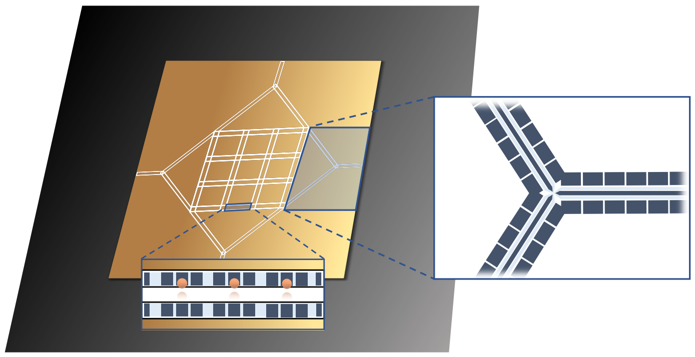
II. Background

To provide the background for this work, this section reviews the principles of trapped-ion quantum computing and the Quantum Charge Coupled Device architecture, highlighting the aspects most relevant to shuttling-based architectures.

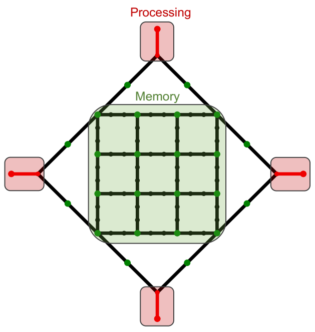
A. Trapped-Ion Quantum Computing

Trapped-ion quantum computers utilize individual ions as qubits, confining them via electromagnetic fields [13], [14]. A common setup used in industrial settings is known as the Paul trap, in which ions are held in a potential generated by a combination of radio-frequency and quasi-static electric fields. By integrating these fields into surface electrode traps, it becomes possible to realize increasingly complex layouts.

A key advantage of trapped-ion technology lies in the ability to physically *shuttle* ions between different locations within the trap. Unlike many other quantum computing platforms that rely on fixed wiring or geometric layouts for qubit connectivity, trapped-ion devices can move qubits as needed, effectively providing all-to-all connectivity. This greatly simplifies connectivity requirements and can reduce the hardware overhead necessary for large-scale quantum processors. However, the act of moving ions must be carefully orchestrated to minimize decoherence and avoid scheduling overhead.



(a) QCCD device



(b) Graph abstraction

Fig. 1: Illustration of a QCCD device and its corresponding graph abstraction

B. Quantum Charge Coupled Device (QCCD) Architecture

The QCCD architecture leverages ion shuttling to build scalable, modular trapped-ion processors [15]. Since the ions are able to move around in the system, the QCCD architecture proposes to partition the trap into specialized zones optimized for different stages of quantum computation: Processing Zones (PZs) for high-fidelity gate operations, Memory Zones (MZs) where ions are stored to protect them from decoherence while idle, Measurement Zones for efficient state readout, and Loading Zones for introducing new ions into the system. Under this model, ions shuttle among these zones as required by a quantum algorithm, e.g., starting in the memory-zone for storage, moving to a processing-zone for gate operations, and then returning once they have completed their gate. To make full use of this potential, future devices may use junctions to connect linear regions and form two-dimensional (2D) architectures. Due to its concept, a dedicated MZ would not only be shielded from noise but also from control elements such as laser pulses, which limits its operational ability. Consequently, ions within a 2D MZ usually cannot simply swap positions and must rather be rearranged.

Example 1. *As a concept of a future 2D QCCD device, see the architecture illustrated in Figure 1a. Below the chip, a linear region is highlighted, capable of holding up to three ions (orange). The electrical control elements confining the ions are shown in blue. On the right, the corresponding control elements forming a processing zone (PZ) are depicted, connected via a Y-junction.*

Effectively managing these constraints to minimize ion motion is crucial for reducing overall runtime and mitigating the impact of decoherence. Recent device prototypes demonstrate the use of multi-zone capabilities [9], [10], yet scaling up to large qubit numbers increases the complexity of moving between zones and efficiently exploiting their potential.

The compilation strategy proposed in this paper addresses exactly these challenges by orchestrating ion shuttling across multi-zone QCCD systems. In particular, we focus on QCCD architectures comprising an MZ structured as a two-dimensional grid and multiple external PZs modeled as linear trap regions. In the following, we consider the shuttling between the MZ and PZs since these are the zones we expect to interact the most while executing a quantum circuit.

III. General Idea

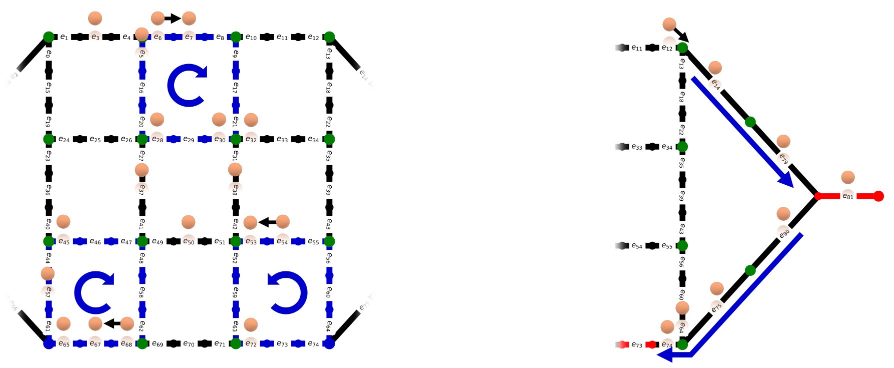
The promise of scalable quantum computing using QCCD architectures relies on the ability to efficiently manage ion movement across increasingly complex device layouts. As architectures incorporate multiple specialized zones, the compilation challenge shifts from merely scheduling gates to orchestrating between computation, shuttling, and resource allocation across distributed areas of the device. Efficiently executing a quantum circuit on a QCCD device requires not only efficient shuttling within zones but also coordinated movement between multiple zones, all while maximizing parallel gate execution whenever circuit dependencies permit.

Previous work has mainly focused on optimized movement within smaller systems and specific trap geometries [16]–[22]. Addressing larger architectures, [23] introduced efficient shuttling compilation within a grid-type MZ. However, that work was limited by its connection to only a single PZ for processing quantum gates. Moreover, shuttling within the PZ was treated as a black-box operation, neglecting the internal structure and specific constraints of moving ions into and out of it. Managing the simultaneous operation of multiple distinct PZs, each with its own internal shuttling and operational constraints, introduces significant orchestration complexity. Our approach addresses these challenges with a compilation framework tailored to QCCD architectures featuring a grid-type MZ and multiple PZs. Instead of treating PZs as black-boxes, we explicitly model them as linear trap regions connected at the memory grid boundaries

This framework integrates three core components:

1) *Memory Zone Shuttling:* We retain efficient cycle-based shuttling schemes for ion movement within the MZ [23].
2) *Shuttling through Processing Zones:* For PZ-related movement through linear regions, we introduce and model path-based shuttling, as detailed in Section IV.
3) *Orchestration Strategy:* Crucially, we implement a novel compilation strategy (Section V) to coordinate all inter-zone shuttling using qubit partitioning and gate commutativity to select favorable gates for each PZ.

By combining explicit modeling of multiple PZs with an orchestration layer that manages dependencies and enables parallelism, our approach offers a holistic solution for compiling quantum computations onto realistic QCCD devices and lays the groundwork for efficiently utilizing next-generation trapped-ion processors.



(a) Shuttling in the MZ grid

(b) Shuttling through linear PZs

Fig. 2: Comparison of cycle-based shuttling within the MZ and path-based shuttling through a linear PZ.

IV. Shuttling

Efficient shuttling remains the key task in compiling QCCD devices. This section introduces the core shuttling techniques used in this work, on which the next section builds to orchestrate multi-zone shuttling.

A. *Shuttling in the Memory Zone Grid*

Movement within the main MZ, structured as a grid connected by X-junctions, utilizes the cycle-based shuttling approach previously developed in [23]. We briefly summarize the key concepts here.

The MZ architecture is represented as an undirected graph $G = (V, E)$, where the set of edges $E = \{e_0, \ldots, e_k\}$ correspond to linear trap segments capable of holding ions, and nodes V represent junctions (major nodes) or intermediate points within linear regions (minor nodes).

Example 2. *A corresponding graph representation of a two-dimensional QCCD device is given in Figure 1b. Since the device can hold three ions in each linear region, the graph consists of three edges in between junctions.*

Moving multiple ions simultaneously along their shortest paths toward a PZ often leads to conflicts, as ions cannot directly swap positions within the MZ. To address this, the cycle-based approach leverages the grid topology. Conflicts are avoided by constructing closed loops (cycles) from graph edges that include both the shortest paths of target ions and any blocking ions. In one time step, all ions on a cycle are rotated forward by one position along the cycle's direction. This enables target ions to advance while simultaneously moving blockers aside without complex backtracking. Grid architectures naturally offer many rectangular loops, supporting simple cycle construction. Multiple cycles can be executed in parallel per time step, provided they do not overlap, i.e., share a junction or edge. The selection of cycles to execute in case of conflicts is determined by the priority queue generated by the orchestration layer. The concept of the priority queue is explained in Section V. For further details on the cycle-based algorithm, the interested reader is referred to [23].

B. *Shuttling through Linear Processing Zones*

While cycles work well within the MZ grid, movement into, within, and out of PZs requires a different approach. Our architecture retains the concept of a shielded MZ, connected to PZs via dedicated paths outside the grid. Crucially, instead of treating PZs as black-boxes, we explicitly model them as linear trap regions. To realize the connection and enable directed movement, we utilize Y-junctions as the interface between the MZ and the linear PZs. While more complex PZ interface designs are conceivable, this topology represents a minimal yet functional model incorporating one-way entry and exit paths and an explicit linear processing region. By introducing additional junction nodes into the entry and exit paths, their length is now also considered in the graph representation.

To schedule the shuttling through these linear zones, we introduce a path-based approach, that can be seamlessly incorporated into the scheduling of the cycle-based approach.

Movement Towards and Into the Processing Zone: When the orchestration strategy (Section V) targets an ion for a specific PZ, and its calculated next move within the MZ would place it onto the linear entry path leading to that PZ, the shuttling mechanism switches from cycle to path generation.

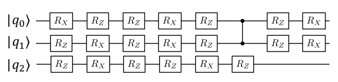
1) A directed path is selected from the ion's current edge along the linear entry path to the target PZ.
2) In one time step, all ions on this path (including the target ion) are shifted one edge forward.
3) Ions can be pushed directly into the PZ edge, provided the PZ is not at its capacity limit.

Movement Out of the PZ: Unlike in [23], multiple PZs may now compete for access to the MZ grid. These exit paths must therefore be integrated into the orchestration scheme (Section V), which then decides which of the paths are scheduled.

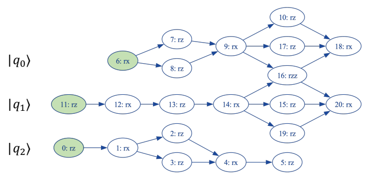
1) To exit a PZ, a suitable unoccupied edge within the MZ is first identified as a target.
2) A Breadth-First Search (BFS) is initiated from the MZ junction connected to the PZ's linear exit path to locate the nearest free edge.
3) Once found, a directed path is constructed from the ion's current edge through the MZ to the target.
4) As with entry paths, all ions along this exit path are shifted one edge forward.

This ensures ions exiting a PZ reach a clear destination in the MZ. Depending on MZ congestion, locating a free edge and completing the move may take multiple time steps.

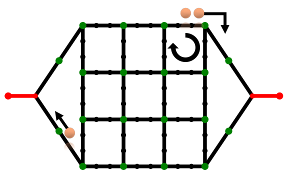
Example 3. *Consider the configuration in Figure 2. In Figure 2a, three ions attempt to move but are blocked in the MZ,*


$|q_0\rangle$ R_X R_Z R_Z R_X R_Z R_Z R_X
$|q_1\rangle$ R_Z R_X R_Z R_X R_Z R_Z R_X
$|q_2\rangle$ R_Z R_X R_Z R_Z R_X R_Z

(a) Circuit



(b) Dependency graph



(c) Shuttling schedule

Fig. 3: Compilation steps

prompting the construction of three cycles to clear the way. In Figure 2b, an ion is trying to enter the entry to the PZ. To make room for the ion, a path is constructed to the PZ edge. To move ions out of the PZ edge, the path leading from the PZ edge to the free edge e_{73} in the MZ is used.

Constraints: A key constraint for both entry and exit paths of a PZ is that movement may be temporarily blocked. During quantum gate execution, access to the corresponding PZ is restricted for the duration of the gate. The orchestration layer must account for these timings when scheduling shuttling. Conflicts can also arise if multiple PZs attempt to use overlapping paths, e.g., exiting into the same MZ region. As with cycles, such conflicts are resolved by the orchestration strategy based on priority (see Section V). Scheduling must also respect PZ capacity limits. If a PZ is at maximum capacity, an ion attempting to enter via the entry path will be blocked unless an ion simultaneously exits the PZ.

V. Orchestrating Multi-Zone Shuttling

Efficient use of a QCCD architecture with multiple PZs requires a compilation strategy that goes beyond simple gate scheduling. It must orchestrate ion movement between the shared MZ and multiple PZs, respecting circuit dependencies and architectural constraints while maximizing parallel gate execution. The proposed approach achieves this by integrating two key components: **(A)** strategic qubit partitioning to minimize shuttling between PZs, and **(B)** dependency-aware gate selection using a Directed Acyclic Graph (DAG) for each PZ. This combination allows the compiler to make informed decisions that balance costly inter-zone shuttling with the goal of maximizing parallel processing.

A. Qubit Partitioning

The first step aims to reduce the overhead from moving ions between PZs. Since PZ connections lie at the boundaries of the memory grid, shuttling between them is typically more time-consuming than waiting for access to an occupied PZ. To minimize such movement, we begin by partitioning the circuit's qubits across available PZs. As only one gate per ion can occur per timestep and single-qubit gates do not introduce dependencies on other qubits, we focus on efficiently scheduling two-qubit gates. Specifically, we aim to group frequently interacting qubits into the same PZ. This is achieved by constructing an interaction graph, where nodes represent qubits and weighted edges capture the number of two-qubit interactions. A repeating bisection strategy based on the Kernighan–Lin (KL) algorithm [24] is then applied: starting with all qubits in one set, we iteratively bisect the largest partition until reaching the desired number of PZs or until further splitting is not possible. The KL algorithm heuristically minimizes the cut size of the interaction graph, reducing the number of inter-zone edges and thereby shuttling. Simultaneously, repeating bisection ensures balanced partition sizes across PZs.

Example 4. *Consider the quantum circuit in Figure 3a. In the case of two available PZs, the proposed strategy focuses on the single two-qubit gate acting on qubits q_0 and q_1. The two ions representing these qubits are mapped to the first PZ, while q_2 will be scheduled to the second PZ.*

This partitioning assigns each qubit a PZ, which guides all subsequent scheduling. However, the KL partitioning might still result in a two-qubit gate acting on ions whose PZs differ. In such cases, the gate itself is dynamically assigned to the PZ that minimizes the combined estimated shuttling cost for bringing *both* required ions to that specific PZ. This ensures that even cross-partition gates are handled efficiently.

B. DAG-based Gate Selection

With partitioning providing a high-level qubit-to-PZ assignment, we next determine the gate execution order, respecting dependencies while promoting parallelism. To do this, we convert the quantum circuit into a *Directed Acyclic Graph* (DAG) using Qiskit [25], where nodes represent gates and edges indicate dependencies. The DAG exposes the *front layer*—gates without preceding dependencies. These gates are mutually commutative and can, in principle, execute in parallel, assuming ion and PZ availability.

Building upon the gate selection strategy from the single-zone setting [23, Sec. IV], which selects the single gate with the closest ions, we extend this concept to multiple PZs. For each PZ, we consider all gates in the current DAG front layer that are assigned to it, either because their single qubit was partitioned to that PZ, or because it was determined to be the optimal PZ for a two-qubit gate as described in Section V-A. From this candidate set, we select the *best* gates: the ones whose required ions are currently closest to their PZ.

Example 5. *Consider the DAG in 3b. Given the scenario in Example 4, we established the partitioned mapping: $\{$PZ1: $\{q_0, q_1\}$, PZ2: $\{q_2\}\}$. The initial front layer containing nodes $\{$0: (rz on q_0), 6: (rx on q_1), 11: (rz on q_2)$\}$ is highlighted in Figure 3b. Accordingly, PZ1 considers the gates of nodes 0 and 6. Let the ion representing q_0 (for Node 0) be at distance 5 and the ion representing q_1 (for Node 6) be at distance 10 from PZ1, then the rz gate of node 0 is selected for PZ1. Since PZ2 considers only the gate of node 11, the rz gate on q_2 is selected for PZ2.*

Based on the DAG, a *priority queue* is constructed for each PZ to resolve conflicts and determine which ions to shuttle forward. For a detailed explanation of the priority queue, see [23, Sec. VI]. This concurrent selection enables the parallel use of all PZs whenever dependencies allow.

C. Orchestration Algorithm

The combination of partitioning and DAG-based gate selection yields the following algorithm:

Algorithm 1: Orchestration Algorithm

```
Input: Quantum circuit and initial ion locations
Output: A schedule $\mathcal{S}$ (shuttling operations, gate
        executions, and required time steps)
1  Initialize:
2  - Partition the ions (mapped to qubits) into sets.
3  - Create the DAG $D = (V, E)$.
4  - Let $\mathcal{S} \leftarrow \emptyset$.
5  - Let $t \leftarrow 0$ (current time step).
6  while $V(D) \neq \emptyset$ do
7      Identify front layer $F$:
         $F = \{ v \in V(D) \mid \nexists u \in V(D)$ with $(u, v) \in E\}$;
8      foreach PZ $p_i$ do
9          Identify candidates $C_{p_i}$ from $F$;
10         Select the best gate $g_{p_i}$ from $C_{p_i}$;
11     Let $G_{\text{done}} \leftarrow \emptyset$ (set of completed gates);
12     while $G_{done} = \emptyset$ do
13         Perform Shuttling Step:
14         Create shuttling operations (cycles and paths);
15         Update ion positions;
16         Add shuttling operations to $\mathcal{S}$;
17         foreach PZ $p_i$ do
18             if ions of $g_{p_i}$ are present in $p_i$ then
19                 Start gate execution of $g_{p_i}$;
20             if $g_{p_i}$ finished its gate time then
21                 Add $g_{p_i}$ to $G_{\text{done}}$;
22         $t \leftarrow t + 1$;
23         Update DAG: Remove nodes of gates in $G_{\text{done}}$;
24 return $\mathcal{S}$
```

This iterative process dynamically adapts the schedule to the evolving ion state and the gates exposed by the DAG. It systematically balances the trade-off between minimizing shuttling through partitioning and maximizing parallelism via concurrent, DAG-aware gate selection, providing a robust compilation strategy for multi-zone QCCD architectures.

VI. Empirical Evaluation

In this section, we evaluatete the performance of our proposed compilation strategy on multi-zone QCCD architectures. We first evaluate the effectiveness of our DAG-based gate selection technique, demonstrate broad applicability through a comprehensive study of various architectures and different quantum circuits, and finally investigate how multiple PZs impact performance relative to architectures with a single PZ.

A. Experimental Setup

To evaluate our approach, we use a suite of quantum benchmarks varying in size and structure:

- "GHZ"; prepares the Greenberger–Horne–Zeilinger state,
- "QFT"; scheduling the quantum Fourier transform, and
- "Random"; using a random circuit of up to four-qubit gates which is as deep as wide.

MQT Bench [26] circuits are translated via pytket [27] to the native RZZ, RZ, RY, and RX gates used in Quantinuum QCCD devices [10]. Each benchmark is compiled and scheduled using a range of QCCD architectures. The MZ is modeled as a grid-type array connected to one or more linear PZs via Y-junctions. Following the approach of [23], the grid is described by four values m, n, v, h: an $m \times n$ grid-graph with v (h) ions between vertical (horizontal) junctions. For example, the graph in Figure 1b corresponds to $4, 4, 3, 3$.

In our evaluations, up to four PZs are connected to the memory grid. Each PZ can hold up to two ions to allow two-qubit gates, but note that this constraint can be relaxed if future hardware supports larger linear regions. In some research setups, two-qubit gates approach single-qubit speed [28], while commercial systems often show a 10–100$\times$ slowdown. We assume junction traversal takes one time step while shuttling along linear paths is instantaneous. To reflect that gate times are typically faster than junction traversal—without overly penalizing two-qubit operations—we model single-qubit gates as one time step and two-qubit gates as three [29]. These values are configurable in the open-source tool.

All experiments were run on an Intel(R) Xeon(R) W-1370P CPU (@ 3.22 GHz) with 32 GiB RAM using Python 3.8.10. Each benchmark was repeated five times with different seeds to reduce bias from initial ion placement. For all experiments, we initially filled each MZ completely with ions.

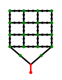
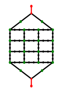
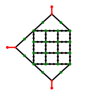
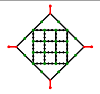
B. Impact of DAG-Based Gate Selection

Figure 4 compares the average execution time (in time steps) achieved with and without the DAG-based gate selection strategy introduced in Section V-B, evaluated across four representative architectures of varying sizes. For this comparison, we executed the QFT circuit on each architecture on all available qubits. While both evaluations used the qubit partitioning step described in Section V-A, the version without DAG-based selection simply used the input circuit as a fixed sequence of gates and moved through it gate-by-gate. The scatter plot illustrates the average total time steps required (left vertical axis), while the corresponding bars highlight the percentage improvement (right vertical axis) provided by the DAG-based method. Across all tested benchmarks, the DAG-based scheme consistently reduces the required time steps by between 52% and 88%. This demonstrates how dependency-aware selection of upcoming gates, rather than naively moving through the input circuit, can efficiently mitigate unnecessary shuttling.

C. Overall Performance Across QCCD Layouts

Next, we evaluated the broader applicability of our method by compiling our full set of benchmarks (QFT, random, and GHZ) for multiple QCCD layouts, as summarized in Table I. Each cell in the table reports the average number of time steps T to complete all gate executions and ion movements for a given circuit with G gates and a specified architecture, along with the CPU time required by our compiler. The architectures are varied in terms of grid size (m, n, v, h) and number of PZs, with each architecture holding N ions. The results show that the proposed approach reliably produces valid shuttling schedules across a wide range of QCCD configurations, including both compact and larger grid-type MZs. The implementation is also able to produce efficient shuttling schedules for a single PZ, as well as for multiple PZs. Furthermore, our implementation is able to efficiently schedule both single- and multi-PZ setups, up to four external PZs. We note that while the tool supports configurations with more than four PZs, we limited this evaluation to a maximum of four to maintain consistency across our comparisons.

TABLE I: Results of the Empirical Evaluation

Architecture			**GHZ**			**QFT**			**Random**		
Number of PZs	m n v h	N	G	$\hat{T}$	t_{CPU} [s]	G	$\hat{T}$	t_{CPU} [s]	G	$\hat{T}$	t_{CPU} [s]
	3 3 1 1	12	220	277.4	1.5	1039	1337.2	10.3	1340	3553.2	48.1
	4 4 1 1	24	460	603.2	6.3	4217	5476.6	139.3	6252	17 601.4	1029.4
	3 3 3 3	36	700	890.6	16.1	10033	12 639.8	741.6	14914	36 413.6	4466.3
	5 5 1 1	40	780	1023.2	18.0	12315	15 566.2	1112.1	17770	45 354.8	6689.8
	3 3 5 5	60	1180	1580.2	50.8	23759	30 353.2	4222.1	40435	52 463.7	11 003.5
Single PZ	4 4 3 3	72	1420	1854.2	74.5	30623	38 622.2	7149.4	59899	90 722.9	15 669.0
	3 3 1 1	12	220	216.4	1.2	1039	849.4	8.3	1340	2477.6	37.2
	4 4 1 1	24	460	492.2	5.0	4217	3383.5	98.4	6252	11 056.6	735.6
	3 3 3 3	36	700	768.8	13.0	10033	8419.8	692.7	14914	23 465.6	3248.0
	5 5 1 1	40	780	819.4	13.7	12315	9641.2	894.8	17770	26 733.4	4547.1
	3 3 5 5	60	1180	1393.0	45.5	23759	24 496.5	3510.9	40435	45 179.1	10 956.7
Two PZs	4 4 3 3	72	1420	1585.8	64.4	30623	25 096.1	5362.7	59899	61 621.1	13 290.9
	3 3 1 1	12	220	186.8	0.8	1039	626.8	7.1	1340	2130.0	35.5
	4 4 1 1	24	460	461.4	5.4	4217	2538.6	82.3	6252	9579.0	697.4
	3 3 3 3	36	700	722.6	13.1	10033	7052.8	507.1	14914	21 296.6	3334.1
	5 5 1 1	40	780	810.8	13.6	12315	7059.4	662.5	17770	23 341.0	4341.3
	3 3 5 5	60	1180	1361.6	47.1	23759	22 292.6	3167.7	40435	42 501.2	10 672.2
Three PZs	4 4 3 3	72	1420	1538.8	68.8	30623	21 033.2	4694.2	59899	48 511.8	12 043.2
	3 3 1 1	12	220	162.2	1.4	1039	520.6	5.1	1340	1712.5	33.9
	4 4 1 1	24	460	424.8	5.2	4217	2028.2	73.4	6252	7088.0	691.9
	3 3 3 3	36	700	708.6	13.1	10033	6514.8	487.3	14914	19 355.0	3355.3
	5 5 1 1	40	780	738.4	14.2	12315	5842.3	596.7	17770	15 803.0	4358.1
	3 3 5 5	60	1180	1360.8	52.1	23759	21 705.7	3828.6	40435	40 630.0	10 023.9
Four PZs	4 4 3 3	72	1420	1520.8	73.4	30623	18 591.1	4369.5	59899	40 822.3	10 871.0

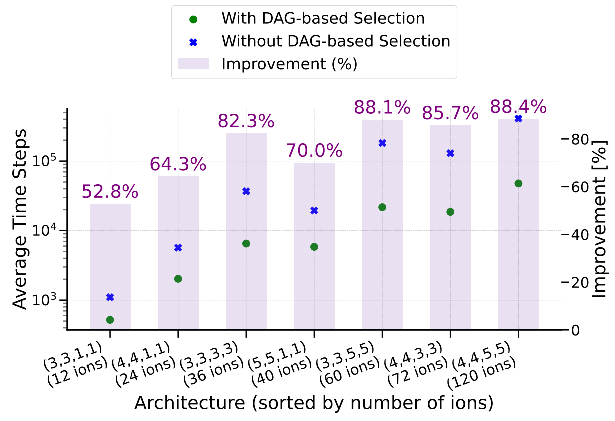


Fig. 4: Improvement in time steps executing "QFT" of using the DAG-based Gate Selection step described in Section V-B.

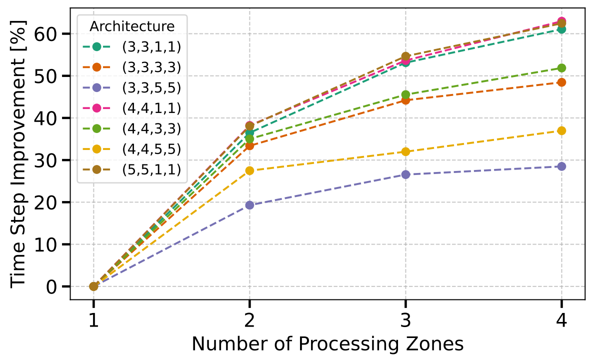


Fig. 5: Improvement in time steps executing "QFT" for an increasing number of PZs.

D. Impact of Multiple Processing Zones

Finally, Figure 5 illustrates how increasing the number of PZs (from one to four) affects the overall schedule length when executing a QFT circuit across several architectures. The y-axis shows the percentage improvement in total execution time relative to the single-zone baseline. As anticipated, adding PZs yields execution time improvements—up to 50–60%—depending on device geometry and circuit structure. These gains result from parallel gate execution, which helps eliminate wait times and reduce memory grid congestion. However, the benefit of each additional PZ diminishes, indicating reduced returns beyond the initial increase in parallelism.

Collectively, these results demonstrate the applicability and effectiveness of our compilation approach for multi-zone QCCD systems while providing first insights into the benefits of multiple available PZs to reduce overall execution runtimes.

VII. Conclusions

In this work, we presented a comprehensive compilation strategy for QCCD systems with a grid-based MZ and multiple connected PZs. Our approach models PZs as linear regions connected via Y-junctions, moving beyond previous black-box abstractions. We integrated path-based shuttling through PZs into an open-source, cycle-based method for conflict-free transport within the MZ. Central to our method is an orchestration layer that combines strategic qubit partitioning with dependency-aware, concurrent gate selection, enabling efficient operation assignment, reduced inter-zone shuttling, and increased parallelism. The corresponding tool is available open-source at https://github.com/cda-tum/mqt-ion-shuttler. Empirical evaluations across three circuits and representative QCCD layouts demonstrated the effectiveness of the proposed approach. The results confirm significant reductions in total execution through the dependency-aware gate selection compared to naively scheduling the input quantum circuit. This allowed us to also assess the impact of scheduling to multiple processing zones. This work provides a robust and extensible framework for compiling quantum circuits onto multi-zone QCCD architectures, laying a foundation for leveraging future hardware advancements. Future research includes more realistic noise models, dynamic qubit partitioning, and scheduling error-correcting codes.

Acknowledgments

This work was supported by the EU's Horizon 2020 programme (DA QC, Grant No. 101001318; MILLENION, Grant No. 101114305), the State of Upper Austria via the COMET programme, the QuantumReady project under Quantum Austria (FFG), and the Munich Quantum Valley, funded by the Bavarian government through Hightech Agenda Bayern Plus.

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Symp. on Foundations of Computer Science*, 1994, pp. 124–134.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Symp. on Theory of Computing*, 1996, pp. 212–219.

[3] R. Babbush *et al.*, "Low-depth quantum simulation of materials," *Phys. Rev. X*, vol. 8, p. 011 044, 1 2018.

[4] M. Kjaergaard *et al.*, "Superconducting qubits: Current state of play," *Annual Review of Condensed Matter Physics*, vol. 11, no. 1, pp. 369–395, 2020.

[5] L. Henriet *et al.*, "Quantum computing with neutral atoms," *Quantum*, vol. 4, p. 327, 2020.

[6] D. Bluvstein *et al.*, "Logical quantum processor based on reconfigurable atom arrays," *Nature*, vol. 626, no. 7997, pp. 58–65, 2023.

[7] S. Slussarenko and G. J. Pryde, "Photonic quantum information processing: A concise review," *Applied Physics Reviews*, vol. 6, no. 4, 2019.

[8] S. Debnath *et al.*, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, no. 7614, pp. 63–66, 2016.

[9] J. M. Pino *et al.*, "Demonstration of the trapped-ion quantum ccd computer architecture," *Nature*, vol. 592, no. 7853, pp. 209–213, 2021.

[10] S. A. Moses *et al.*, *A race track trapped-ion quantum processor*, 2023. arXiv: 2305.03828 [quant-ph].

[11] M. P. da Silva *et al.*, *Demonstration of logical qubits and repeated error correction with better-than-physical error rates*, 2024. arXiv: 2404.02280 [quant-ph].

[12] R. Wille *et al.*, "The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing," in *IEEE International Conference on Quantum Software*, 2024. arXiv: 2405.17543, A live version of this document is available at https://mqt.readthedocs.io.

[13] T. P. Harty *et al.*, "High-fidelity preparation, gates, memory, and readout of a trapped-ion quantum bit," *Phys. Rev. Lett.*, vol. 113, p. 220 501, 22 2014.

[14] J. I. Cirac and P. Zoller, "Quantum computations with cold trapped ions," *Phys. Rev. Lett.*, vol. 74, pp. 4091–4094, 20 1995.

[15] D. Kielpinski, C. Monroe, and D. J. Wineland, "Architecture for a large-scale ion-trap quantum computer," *Nature*, vol. 417, no. 6890, pp. 709–711, 2002.

[16] P. Murali, D. M. Debroy, K. R. Brown, and M. Martonosi, "Architecting noisy intermediate-scale trapped ion quantum computers," in *International Symposium on Computer Architecture*, 2020, pp. 529–542.

[17] O. Keszöcze, N. Mohammadzadeh, and R. Wille, "Exact physical design of quantum circuits for ion-trap-based quantum architectures," *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 344–349, 2021.

[18] A. A. Saki, R. O. Topaloglu, and S. Ghosh, "Muzzle the shuttle: Efficient compilation for multi-trap trapped-ion quantum computers," in *2022 Design, Automation &; Test in Europe Conference &; Exhibition (DATE)*, 2022, pp. 322–327.

[19] T. Schmale *et al.*, "Backend compiler phases for trapped-ion quantum computers," in *2022 IEEE International Conference on Quantum Software (QSW)*, 2022.

[20] F. Kreppel *et al.*, "Quantum circuit compiler for a shuttling-based trapped-ion quantum computer," *Quantum*, vol. 7, p. 1176, 2023.

[21] C.-M. Chang, J.-H. R. Jiang, D.-W. Chiou, T. Hsu, and G.-D. Lin, "Quantum circuit compilation for trapped-ion processors with the drive-through architecture," *IEEE Transactions on Quantum Engineering*, pp. 1–14, 2025.

[22] D. Schoenberger, S. Hillmich, M. Brandl, and R. Wille, "Using Boolean satisfiability for exact shuttling in trapped-ion quantum computers," in *Asia and South-Pacific Design Automation Conf.*, 2024.

[23] D. Schoenberger, S. Hillmich, M. Brandl, and R. Wille, *Shuttling for scalable trapped-ion quantum computers*, 2024. arXiv: 2402.14065 [quant-ph].

[24] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[25] A. Javadi-Abhari *et al.*, *Quantum computing with qiskit*, 2024. arXiv: 2405.08810 [quant-ph].

[26] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, vol. 7, p. 1062, 2023.

[27] S. Sivarajah *et al.*, "T—ket⟩: A retargetable compiler for nisq devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014 003, 2020.

[28] V. M. Schäfer *et al.*, "Fast quantum logic gates with trapped-ion qubits," *Nature*, vol. 555, no. 7694, pp. 75–78, 2018.

[29] K. R. Brown, J. Kim, and C. Monroe, *Co-designing a scalable quantum computer with trapped atomic ions*, 2016. arXiv: 1602.02840 [quant-ph].