Muzzle the Shuttle: Efficient Compilation for Multi-Trap Trapped-Ion Quantum Computers

Abdullah Ash Saki
Dept. of Electrical Engineering
Pennsylvania State University
University Park, PA
ash.saki@live.com

Rasit Onur Topaloglu
IBM
Poughkeepsie, NY
rasit@us.ibm.com

Swaroop Ghosh
Dept. of Electrical Engineering
Pennsylvania State University
University Park, PA
szg212@psu.edu

***Abstract*—Trapped-ion systems can have a limited number of ions (qubits) in a single trap. Increasing the qubit count to run meaningful quantum algorithms would require multiple traps where ions need to shuttle between traps to communicate. The existing compiler has several limitations, which result in a high number of shuttle operations and degraded fidelity. In this paper, we target this gap and propose compiler optimizations to reduce the number of shuttles. Our technique achieves a maximum reduction of 51.17% in shuttles (average $\approx 33\%$) tested over 125 circuits. Furthermore, the improved compilation enhances the program fidelity up to 22.68X with a modest increase in the compilation time.**

***Index Terms*—Quantum Computing, Qubit, Trapped-Ion, Shuttle, Compiler**

I. Introduction

The trapped-ion (TI) quantum bit (qubit) is one of the front-runner technologies to build practical quantum computers. They offer several advantages such as identical qubits, long coherence times, and all-to-all connectivity among qubits [1]. Several companies such as IonQ and Honeywell are pursuing this technology. Recently, Honeywell reported a trapped-ion system with a high quantum volume (QV) of 1024 [2]. Several TI hardware systems [1], [2] are already commercially available through quantum cloud services such as Honeywell, IBM Quantum Experience, AWS Braket, and Microsoft Azure. Moreover, TI systems are being used for many practical cases and demonstrations (e.g., [3]).

The existing TI systems have a smaller number of qubits compared to their superconducting counterparts (IBM's Manhattan quantum processor has 65 qubits whereas the largest known TI system has 11 qubits [1]). However, roadmaps for larger systems with 50-100 qubits are in place [4], [5]. Confining many ions in a single trap is a major roadblock for scalable TI systems. With many ions in a single trap, the spacing between ions reduces and makes individual ion addressing difficult. Moreover, the gate time becomes slow, leading to a longer program execution time. To resolve these issues and to build TI systems with more ions, quantum charge-coupled device (QCCD) based on multiple interconnected traps are proposed [6]. Fig. 1 shows a schematic of a 2-trap system interconnected by a *shuttle path*.

This work is supported by National Science Foundation (NSF) (OIA-2040667 and DGE-2113839) and seed grants from Penn State Institute for Computational and Data Sciences (ICDS) and Penn State Huck Institute of the Life Sciences.

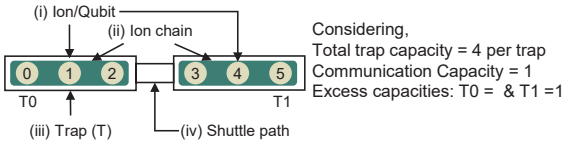


Fig. 1. Schematic of a multi-trap TI system. Ions (qubits) (i) are confined inside traps (iii) using DC and oscillatory potentials. Inside a trap, ions form a chain (ii). Traps are interconnected via the shuttle path (iv) which allows movement of ions between traps.

In a multi-trap system, sometimes computation is required on data from ions situated in different traps. In such cases, one ion needs to be shuttled (moved) from one trap to the other so that the ions are co-located, and the gate operation can be performed. A compiler adds shuttle operations to a quantum program to satisfy the inter-trap communication. The shuttle operation is expensive as it degrades quantum gate fidelity. Therefore, minimizing the number of shuttle operations is beneficial and desired. Murali et al. [7] performed extensive architectural studies for multi-trap trapped ion systems. They developed a QCCD compiler to generate hardware executable programs from high-level versions and a simulator [1] with experimentally calibrated values. Although the first attempt to build a compiler for multi-trap TI systems, the compiler in [7] suffers from several inefficiencies (described in detail in Section III), which lead to a higher number of shuttle operations. This elevated shuttle operations, in turn, inevitably increase ion-chain energy and degrades program fidelity.

In this paper, we optimize the compiler to reduce the number of shuttles and improve the program fidelity in the process. We make the following contributions in this paper:

- We introduce three optimization heuristics, i.e., future ops-based shuttle direction policy, opportunistic gate re-ordering, and nearest-neighbor-first re-balancing (with better ion selection).
- We evaluate our proposals across 5 NISQ and 120 random quantum benchmarks and compare with [7] to showcase the efficiency of our optimizations.
- We report the improvement in fidelity with minor compilation

[1]Ref. [8] is the accompanying code-base for [7].

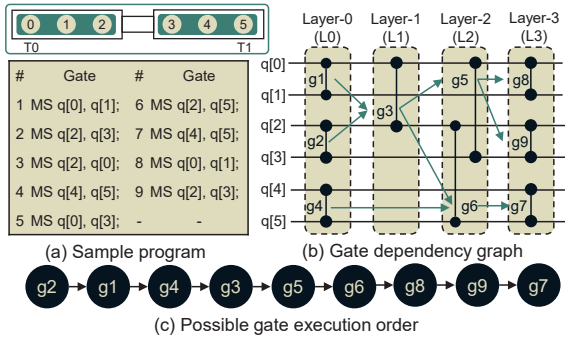


Fig. 2. (a) A sample quantum program consisting of 2-qubit gates. (b) Gate dependency graph of the sample program. (c) A possible gate order that satisfies the dependency graph. (Inset on top left: Possible allocation of 6 ions in the 2-trap system).

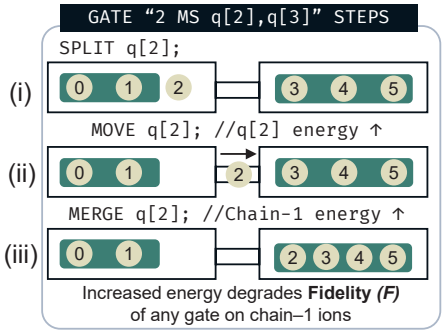


Fig. 3. Shuttle steps to bring ions 2 and 3 in the same trap.

time overhead.

The rest of the paper is organized as follows: Section II describes the basics of trapped-ion systems. Section III explains the limitations of the existing compiler and presents our algorithms for compiler optimizations. Section IV reports numerical values of the number of shuttles, fidelity, and compilation time. Finally, the conclusion is drawn in Section V.

II. Basics

This section discusses the basics of trapped-ion quantum computers and the terminologies used in the paper.

A. Gate dependency graph

A quantum program is a sequence of quantum gates. Fig. 2 shows a sample quantum program consisting of 2-qubit MS gates. A quantum program can be converted to a gate dependency graph (a directed acyclic graph, DAG) which consists of *layers*. Gates in a layer are independent of each other but depend on one or more gates from previous layers. Fig. 2b shows the dependency graph for the sample program in Fig. 2a. Gates $g5$ and $g6$ are independent of each other (both in Layer–2 or L2) as they work on different sets of qubits. However, both $g5$ and $g6$ depends on $g3$ which means $g5$ or $g6$ cannot be executed before $g3$. Fig. 2c shows a possible gate order that satisfies the DAG in Fig. 2b.

B. Trapped-ion Quantum Computer

1) Trap details: A trapped-ion system consists of micro-fabricated surface electrode traps which confine ions like Yb or Ca using electromagnetic fields [1]. We schematically show different components of a trapped-ion system in Fig. 1. A single trap can accommodate a fixed number of ions. We name this *total trap capacity*. During the initial allocation of ions, a part of the total trap capacity is loaded with ions, and the remaining capacity (termed as *communication capacity*) is kept unoccupied to allow for shuttled ions from other traps. During program execution, ions may need to move (get shuttled) between traps which will free up one trap and fill up another trap. The number of free spaces is termed as *excess capacity* (EC) and defined as "total trap capacity − number of ions in a trap". Inside a trap, gates are executed serially (technology constraint) while different traps can have parallel gates.

2) Need for a shuttle operation: Qubits (ions) are accessible directly inside a trap in trapped-ion systems.

For example, the 1^{st} gate `MS q[0], q[1]` in the sample program (Fig. 2a) can be directly executed as both ions 1 and 2 are located in the same trap T0. However, the 2^{nd} gate in the sample program, `MS q[2], q[3]`, cannot be directly executed as ion 2 is in T0 and ion 3 is in T1. One of the ions needs to be shuttled to bring both ions into the same trap.

3) Gate fidelity model: Quantum gates in existing quantum computers including TI systems are erroneous. Ref [7] presents an analytical gate fidelity model for TI systems: Fidelity $F = 1-\Gamma\tau-A(2\bar{n}+1)$. Here, Γ = trap heating rate, τ = gate time, $\bar{n}$ = motional mode or vibrational energy of an ion-chain, A is a scaling factor that varies as $\#qubits/log(\#qubits)$. The simulator in [7] uses experimental values of Γ, τ, A, and $\bar{n}$ [9], [10]. The values are reported in the paper [7] and are embedded in the GitHub code-base [8]. We omit those values in this paper for brevity.

4) Shuttle steps and impact on fidelity: The shuttle operation involves several steps as depicted in Fig. 3. First, ion 2 is split from the ion-chain, and then, moved from T0 to T1. The movement adds energy to the ion. Then, ion 2 is merged with the other chain, and gate `MS q[2],q[3]` can be executed. This merge operation increases the vibrational energy ($\bar{n}$) of the chain in T1. As motional mode $\bar{n}$ is now higher, the subsequent gate operations in this chain will experience a lower gate fidelity (F).

III. Compiler Optimizations

In this section, we describe our compiler optimization algorithms. We also discuss the key policies of the QCCD compiler [7] along with its limitations and show how our approaches can address them.

A. Shuttle direction policy

Shuttle direction policy dictates which ion will be moved to execute a 2-qubit gate. The shuttle direction policy in [7] uses excess capacity as the deciding factor. The policy is illustrated in Listing 1.

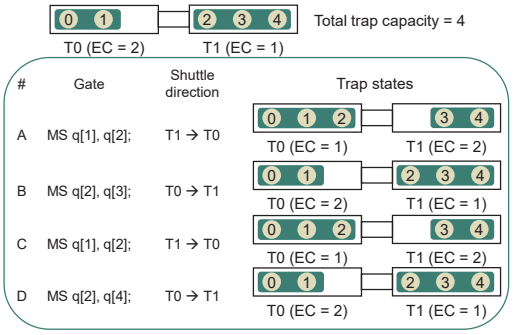


Fig. 4. Issues with the shuttle direction policy used in [7], [8]. Excess capacity-based logic can lead to repeated shuttles.

```
if excess_cap0 < excess_cap1:
    Move Trap0 --> Trap1
elif excess_cap0 == excess_cap1:
    Move 1st ion of the gate
else:
    Move Trap1 --> Trap0
```

Listing 1. Shuttle direction policy [7], [8].

*1) **Issue with excess capacity-based policy**:* The excess capacity-based shuttle direction policy can result in repeated shuttle between traps. We illustrate the issue with an example in Fig. 4. Consider a 2-trap system with a total trap capacity of 4 ions per trap. Consider, there are 2 ions in T0 and 3 ions in T1. Therefore the excess capacities (ECs) of the traps are EC(T0) = $4-2=2$ and EC(T1) = $4-3=1$.

Next, consider 4 gates to be executed starting with `MS q[1],q[2]`. As EC(T0) $>$ EC(T1), according to the shuttle policy in [7] (Listing 1), ion 2 will be moved from T1 to T0. The trap states are updated after this shuttle, and new excess capacities are EC(T0) $=$ 1 and EC(T1) $=$ 2. For the next gate `MS q[2],q[3]`, ion 2 will again be moved back to T1 according to the shuttle policy. Likewise, the next 2 gates will also require shuttles. Therefore, 4 shuttles are required, and ion 2 is shuttled back-and-forth between T0 and T1.

*2) **Proposed future ops-based shuttle direction**:* We propose a future operations-based policy. Suppose, ion_A belongs to $trap_A$, and ion_B belongs to $trap_B$. To implement gate(ion_A, ion_B), one of the ions needs to be shuttled. To make the decision, our heuristics algorithm computes *move score* for each ion as defined below:

- $ion_{A(A\rightarrow B)}$ move score $=$ # ion_A gates in $trap_B$ $+$ # ion_B gates in $trap_B$
- $ion_{B(B\rightarrow A)}$ move score $=$ # ion_A gates in $trap_A$ $+$ # ion_B gates in $trap_A$

An $ion_{A(A\rightarrow B)}$ move score greater than $ion_{B(B\rightarrow A)}$ move score means that keeping both ions in $trap_B$ will satisfy more future gates than if both ions are kept in $trap_A$. Therefore, moving ion_A to $trap_B$ will be more beneficial and vice-versa.

Consider the 4 gate-program in Fig. 4. Suppose, $ion_A = 1$, $ion_B = 2$, $trap_A =$ T0, and $trap_B =$ T1. To perform Gate-A, our logic will look-up the remaining 3 gates and compute the move score as tabulated in Table I. As $ion_{A(A\rightarrow B)}$ move score $> ion_{B(B\rightarrow A)}$ move score, $ion_A = 1$ will move from $trap_A$ (T0) to $trap_B$ (T1) for Gate-A according to our optimized logic. Moving ion 1 will satisfy requirement for the remaining 3 gates in this case, and thus, we will require only 1 shuttle compared to 4 as in the previous case.

TABLE I
MOVE SCORE COMPUTATION EXAMPLE

# ion_A gates in $trap_B$	1	(Gate-C)
# ion_B gates in $trap_B$	2	(Gate-B and D)
$ion_{A(A\rightarrow B)}$ move score	3	-
# ion_A gates in $trap_A$	0	-
# ion_B gates in $trap_A$	1	Gate-C
$ion_{B(B\rightarrow A)}$ move score	1	-

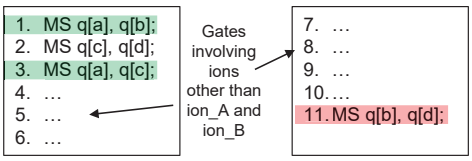


Fig. 5. Gate proximity consideration. Distant gates are excluded.

*3) **Gate proximity consideration**:* While computing the number of future gates, we adopt a proximity-based approach. We argue that not all gates involving ion_A and ion_B have the same priorities in decision making. If there are many other gates between two gates involving ion_A and/or ion_B, then we flag the later gates as distant (low proximity) and exclude them from *move score* computation. The proximity between two gates involving ion_A and ion_B is a design parameter in our compiler optimization. The distance should not be too low as it should exclude most future gates from consideration and should not be too high as distant future gates may not represent ion locations correctly. From our analysis, setting the proximity parameter to 6 provided good results.

We illustrate the proximity-based approach in Fig. 5. Gate 1 and 3 involves either 1 or both of ion_A and ion_B. There is one other gate between these two gates, and thus, distance $= 1$. As distance $1 < 6$, gate 3 is considered. The next gate involving ion_A and/or ion_B is gate 11. The distance between gate 3 and gate 11 is 7 which is greater than a set threshold of 6. Thus, gate 11 (and any related gate after gate 11) is marked as *low-proximity* and excluded from score computation.

*4) **Comment on the complexity**:* The future ops-based shuttle policy checks remaining gates for each gate requiring shuttle. It can lead to $O(n^2)$ time complexity, where n is the number of 2-qubit gates in the program. With gate proximity consideration, the complexity becomes $O(n.k)$ where $k \leq n$. Moreover, not all gates require shuttles and do not invoke the shuttle direction-related computations. Combined with gate proximity consideration, this keeps the value of n and the compilation time in check even for very large circuits.

B. Gate execution order

For gate execution order, we keep the baseline earliest-ready-gate-first heuristics as in [7]. It finds the order by topologically sorting the gate dependency graph. In some cases, the favorable shuttle direction computed by the future ops-based shuttle

Algorithm 1: Re-order gate execution

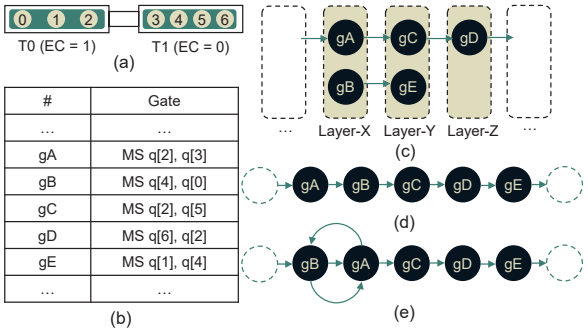
Input: active gate, old_destination, gate dependency graph, gate order, remaining gates
Output: new gate order

active layer ← the *layer* the *active gate* belongs to;
check_gates ← empty;
for *layer* ∈ 1^{st} *layer to active layer* **do**
  for *gate* ∈ *layer* **do**
    if *gate* ∈ *remaining gates and gate* ≠ *active gate* **then**
      check_gates.append(gate);
    end
  end
end
for *gate* ∈ *check_gates* **do**
  Find source trap for the *gate* using future-ops shuttle policy;
  if *source_trap* == *old_destination* **then**
    new_gate_order ← Remove *gate* from gate order and insert before active gate; break;
  end
end

direction policy may not be achievable if the destination trap is full. In such cases, we reorder the gate execution sequence to free up the trap maintaining the gate dependency graph. We name the gate to be executed as *active gate*. First, we identify the *layer number* of the active gate (e.g., if $g8$ from Fig. 2b is the active gate, then layer number is 3). Any pending gate in this layer and preceding layers are candidates as they can be executed without breaking the gate dependency graph. The algorithm checks each candidate gate. If a candidate gate can free up space in the destination trap, then it is moved up the gate execution order before the active gate, and it becomes the new active gate. The algorithm is detailed in Algorithm 1.

Example 3.1: Fig. 6 illustrates an example of gate re-ordering. Gate-A (gA) requires a shuttle, and the favorable direction is moving ion 2 from T0 to T1 (as ion 2 has more operations in T1). However, with the present trap state (Fig. 6a) T1 is full and cannot accept an incoming ion. For such a case, the gate re-ordering logic will be invoked. The logic checks candidate gates (pending gates in the active-layer, Layer-X in this example, and preceding layers). Gate-B in Layer-X is a candidate gate. This gate (gB) also requires a shuttle, and the direction is moving ion 4 from T1 to T0 (as ion 4 has more operations in T0). Thus, the re-ordering the baseline order in Fig. 6d to the order in Fig. 6e (i.e., executing gB before gA) will free up a space in T1. Next, gA can execute with its favorable direction (ion 2, T0 → T1). Fig. 6f illustrates how re-ordering gates to support favorable shuttle direction can save shuttles in the example partial program.

*1) **Comment on the complexity**:* The algorithm checks every *pending gate* in the active layer and the preceding layer(s), and for each pending gate computes a shuttle direction. Again, it has a time complexity of $O(n^2)$. However, the number of pending



#	Gate
…	…
gA	MS q[2], q[3]
gB	MS q[4], q[0]
gC	MS q[2], q[5]
gD	MS q[6], q[2]
gE	MS q[1], q[4]
…	…

Without gate re-ordering (5 shuttles)

Gate	Ion & Direction	Trap state after the gate
MS q[2], q[3]	Ion 3, T1 → T0	T0: [0, 1, 2, 3] T1: [4, 5, 6]
MS q[4], q[0]	Ion 0, T0 → T1	T0: [1, 2, 3] T1: [0, 4, 5, 6]
MS q[2], q[5]	Ion 5, T1 → T0	T0: [1, 2, 3, 5] T1: [0, 4, 6]
MS q[6], q[2]	Ion 2, T0 → T1	T0: [1, 3, 5] T1: [2, 0, 5, 6]
MS q[1], q[4]	Ion 4, T1 → T0	T0: [1, 3, 5, 4] T1: [2, 0, 6]

With gate re-ordering (2 shuttles)

Gate	Ion & Direction	Trap state after the gate
MS q[4], q[0]	Ion 4, T1 → T0	T0: [0, 1, 2, 4] T1: [3, 5, 6]
MS q[2], q[3]	Ion 2, T0 → T1	T0: [0, 1, 4] T1: [2, 3, 5, 6]
MS q[2], q[5]	No shuttle	T0: [0, 1, 4] T1: [2, 3, 5, 6]
MS q[6], q[2]	No shuttle	T0: [0, 1, 4] T1: [2, 3, 5, 6]
MS q[1], q[4]	No shuttle	T0: [0, 1, 4] T1: [2, 3, 5, 6]

(f)

Fig. 6. An example of opportunistic gate re-ordering. (a) An example trap state. (b) Partial quantum program. (c) Gate dependency graph of the partial program. (d) Baseline gate execution order. (e) Re-ordered gate sequence to free-up T1. (f) Illustration of shuttle reduction with gate re-ordering.

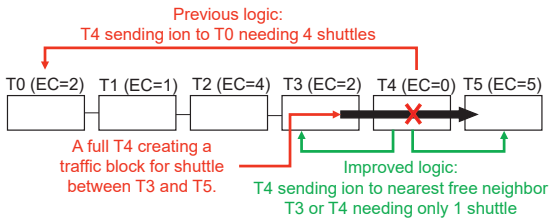


Fig. 7. Trap re-balancing logic for traffic block resolution: the problem with the previous logic and a fix with our improved logic.

gates is typically small even for large circuits. This keeps compilation time in check (supported by numerical results).

C. Resolving traffic blocks

The compiler [7] incorporates logic to resolve traffic blocks in a shuttle path. If a trap is full, it cannot receive any ion which potentially creates a traffic block. The compiler uses a minimum-cost-maximum-flow (MCMF) algorithm to move an ion from a full trap and resolve the traffic block. In [7], the re-balancing logic always starts searching for a destination trap from $trap - 0$. This may result in an inefficient re-balancing which is illustrated with the following example.

*1) **Issue with re-balancing logic**:* Consider, an ion needs to shuttle between T3 and T5 (Fig. 7). However, T4 is full creating a traffic block. To remove the traffic block, an ion from T4 needs to be moved to another trap. Therefore, we need another trap (destination) with excess capacity (> 0). With the present logic in QCCD-simulator [7], the search for a destination trap always starts with T0. In the example in Fig. 7,

T0 has excess capacity, therefore T4 will send an ion to T0 to re-balance the trap (i.e., free-up traffic block). This will require 4 shuttles. However, the neighboring traps of T4 (T3 and T5) also have excess capacities. Thus, moving to either T3 or T5 would require only 1 shuttle.

Algorithm 2: Nearest-neighbor-first re-balancing

```
Input: blocking_trap, all_traps, trap_topology
Output: destination_trap
source_trap ← blocking_trap;
candidate_dest_dist ← empty hash table;
for candidate_dest_trap ∈ all_traps do
    if candidate_dest_trap ≠ source_trap and
      trap.excess_capacity > 0 then
        distance ← shortest distance between
          source_trap and candidate_dest_trap on
          trap_topology;
        candidate_dest_dist[candidate_dest_trap] = dist;
    end
end
destination_trap ← key (candidate_dest_trap) with the
  smallest distance in candidate_dest_dist;
```

*2) **Nearest-neighbor-first re-balancing logic**:* We improve the re-balancing logic by searching from nearest-neighbor traps first. The algorithm (Algorithm 2) first filters out traps with 0 excess capacities and creates a list of candidate (destination) traps that can accept an ion. Finally, the nearest candidate trap is selected as the destination.

Max-score shuttle ion selection: Besides selecting the destination trap intelligently, we add another optimization in the logic to select a better ion to move. We apply the following heuristics: the ion should have a high number of gates in the destination trap and a low number of gates in the source trap. We compute a *score* = ($wd \times$ # gates in destination $- ws \times$ # gates in source) for each ion in the source trap. Usually, we set $wd = ws = 0.5$. If # gates in destination and # gates in source are equal we make $wd = 0.49$ and $ws = 0.51$ to avoid score being 0. Finally, the ion with the highest score is moved.

*3) **Comment on the complexity**:* Finding a neighboring trap with free spaces has linear time complexity. As the number of traps is small, searching for a candidate trap is fast. For the *max-score shuttle ion* algorithm, the number of ions to consider becomes a constant as we have a fixed source trap. Thus, it also has a complexity of $O(constant \times n)$. In [7], the authors show that $15 - 25$ ions per trap is suitable NISQ applications which bounds the $constant$ to 25.

IV. Evaluation and Discussions

A. Evaluation Setup

Experimental Platform: All simulations are run on an Ubuntu 20.04 virtual machine with 8 GB RAM on a Windows 10 host with Intel i7-9700k 3.60 GHz.

Benchmarks: To showcase the efficacy of the compiler optimizations, we choose several NISQ benchmarks from [7] and the Qiskit circuit library. The benchmark suite includes circuits from Google's supremacy experiment, quantum approximate optimization algorithm (QAOA), quantum Fourier transform (QFT), Square Root, and QuadraticForm [11] (quadratic form finds its application in constrained polynomial binary optimization problems). Besides the NISQ benchmarks, we test our compiler with 120 random circuits. The random circuits are of sizes 60, 65, 70, and 75 qubits. For each size, we take 30 randomly generated circuits with average 1438 2-qubits gates ($\sigma \approx 413$). For random circuits, we tabulate the mean value with standard deviation in parentheses for performance metrics.

Hardware model: For a fair comparison, we use the same hardware model as in [7]. We consider the "L6" trap topology as in [7] where 6 traps are connected in a linear fashion (Fig. 7). Each trap has a total capacity of 17 with a communication capacity of 2 per trap. To get the program fidelity estimates, we leverage the QCCD simulator [7] which includes experimental operation time and gate fidelity models.

B. Number of shuttles

Table II shows the reduction in number of shuttle operations for this work compared to [7] for benchmarks listed in Section IV-A. We observe a $\approx 19\%$ to 51% reduction in shuttles. Our compiler outperforms (results in significantly less number of shuttles) the QCCD compiler in [7] for every circuit in the test suite of 125 circuits (supports algorithms' stability).

The 5 NISQ benchmarks have structured but different 2-qubit gate pattern [7]. For example, Supremacy and QAOA circuits have *nearest neighbor gate* pattern, and for both of them, we observe $\approx 38\%$ reduction. The QFT and the QuadraticForm circuits have *all-to-all* connectivities. For such circuits, moving one ion satisfies many future gates, and thus, they tend to have a smaller number of shuttles. The SquareRoot circuit has *short and long-range* gates, and results indicate that we may get the best reductions for such patterns. The random circuits have a more unstructured pattern. Our compiler works for random gate patterns as well achieving 26% reduction on average.

C. Program fidelity improvement

Shuttle operation increases vibrational energy or motional mode($\bar{n}$) of an ion-chain i.e., heats ion-chain and degrades gate fidelity. As our compiler optimizations reduce shuttles, it curbs motional mode resulting in the improved gate and overall program fidelity. Fig. 8 shows improvement in program fidelity across different benchmarks. QAOA exhibits a very high gain

TABLE II
Reduction in the number of shuttles

Benchmark	Qubits	2Q gates	[7]	This Work	$\Delta(\downarrow)$	$\%\Delta$
Supremacy	64	560	365	223	142	38.90%
QAOA	64	1260	1552	957	595	38.34%
SquareRoot	78	1028	717	355	372	51.17%
QFT	64	4032	241	196	45	18.67%
QuadraticForm	64	3400	228	164	64	28.07%
Random	60-75	1438 (413)	1048	775 (270)	273 (109)	26% (6)

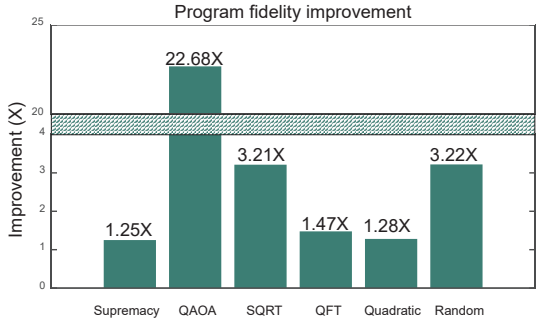


Fig. 8. Improvement in program fidelity compared to [7].

TABLE III
COMPILATION TIME OVERHEAD

Benchmark	Compile time (sec) [This work]	Compile time (sec) [7]	$\Delta(\uparrow)$ (sec)
Supremacy	2.6	1.1	1.5
QAOA	12.99	3.88	9.11
SquareRoot	6.29	1.83	4.46
QFT	18.42	4.22	14.2
QuadraticForm	24.55	3.74	20.81
Random	19.15 (12.59)	3.53	15.62 (11.28)

as it requires the highest number of shuttles across benchmarks. In general, applications with a high shuttle-to-gate ratio will experience more improvement in program fidelity. This is because the fidelity of such applications is dominated by shuttle-induced aggravated motional mode (last term of the fidelity equation, Section II) than the background heating. However, reducing shuttle operations still significantly improves program fidelity for benchmarks with a low shuttle-to-gate ratio, e.g, QFT.

D. Compilation time overhead

Our compiler optimizations for shuttle direction, gate reordering, and efficient re-balancing increase the compilation time compared to [7]. Our algorithms have worst case time complexity of $O(n^2)$. However, as we discuss in Section III-A4, III-B1, and III-C3, the value of n is contained. Therefore, the compilation time remains tractable even for very large circuits like QFT and QuadraticForm (3000-4000 gates) as evident from the Table. III. For all the circuits the increase in compilation times is in a few tens of seconds, and it remains under a minute for very large circuits. Therefore, we are trading-off compilation time in a scalable manner to reduce # operations.

E. Discussions

1) Heuristic vs. exact methods: Applying exact methods like integer linear programming (ILP) and satisfiability modulo theorem (SMT) solvers can lead to the best results. However, these methods do not scale well with circuit size. For the NISQ-era benchmarks considered in this paper or the previous paper [7] the exact approaches will become intractable. Therefore, majority proposals on quantum compilers (e.g., [12], [13]) resort to heuristics methods and trade off some performance in the favor of better scalability. We also follow the same approach.

2) Test benchmark sizes: We test benchmarks with $60-75$ qubits and thousands of 2-qubit gates. One can argue that these benchmarks are not of practical sizes for present-day noisy devices. However, the domain of quantum computing is in a trajectory where benchmarks of this size will become and must become practical. Our compiler generates more compact circuits with substantially less number of operations (shuttles) which will definitely be beneficial.

3) Initial mapping policy: In this paper, we used popular greedy initial mapping policy [14]. In the future, different initial mapping policies can be explored. Even without modifying the initial mapping policy, our compiler shows significant gains.

V. CONCLUSION

In this paper, we present compiler optimizations for multi-trap TI quantum computers. Our methods drastically reduce the number of shuttles compared to previous state-of-the-art and improve program fidelity.

REFERENCES

[1] K. Wright, K. Beck, S. Debnath, J. Amini, Y. Nam, N. Grzesiak, J.-S. Chen, N. Pisenti, M. Chmielewski, C. Collins *et al.*, "Benchmarking an 11-qubit quantum computer," *Nature communications*, vol. 10, no. 1.

[2] Honeywell, "Honeywell Sets Another Record For Quantum Computing Performance," Jul. 2021. [Online]. Available: https://www.honeywell.com/us/en/news/2021/07/honeywell-sets-another-record-for-quantum-computing-performance

[3] ——, "How BMW Can Maximize Its Supply Chain Efficiency with Quantum," Jan. 2021. [Online]. Available: https://www.honeywell.com/us/en/news/2021/01/exploring-supply-chain-solutions-with-quantum-computing

[4] ——, "Get to Know Honeywell's Latest Quantum Computer System Model H1," Oct. 2020. [Online]. Available: https://www.honeywell.com/us/en/news/2020/10/get-to-know-honeywell-s-latest-quantum-computer-system-model-h1

[5] STAQ, "Software-Tailored Architectures for Quantum Co-design - Hardware," May 2021. [Online]. Available: https://staq.pratt.duke.edu/hardware

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

[7] P. Murali, D. M. Debroy, K. R. Brown, and M. Martonosi, "Architecting Noisy Intermediate-Scale Trapped Ion Quantum Computers," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 529–542.

[8] P. Murali, "QCCDSim," Jan. 2021. [Online]. Available: https://github.com/prakashmurali/QCCDSim

[9] P. H. Leung, K. A. Landsman, C. Figgatt, N. M. Linke, C. Monroe, and K. R. Brown, "Robust 2-qubit gates in a linear ion crystal using a frequency-modulated driving force," *Phys. Rev. Lett.*, vol. 120, p. 020501.

[10] M. Gutiérrez, M. Müller, and A. Bermúdez, "Transversality and lattice surgery: Exploring realistic routes toward coupled logical qubits with trapped-ion quantum processors," *Phys. Rev. A*, vol. 99, p. 022330.

[11] A. Gilliam, S. Woerner, and C. Gonciulea, "Grover Adaptive Search for Constrained Polynomial Binary Optimization," *Quantum*, vol. 5, p. 428.

[12] IBM, "Transpiler," May 2021. [Online]. Available: https://qiskit.org/documentation/stubs/qiskit.compiler.transpile.html

[13] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for nisq-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, New York, NY, USA, 2019.

[14] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1015–1029.