MultiQ: A Compiler-Runtime and Verifier for Multiprogramming Neutral Atom Architectures

Francisco Romão
Technical University of Munich
Munich, Germany

Daniel Vonk
Technical University of Munich
Munich, Germany

Emmanuil Giortamis
Technical University of Munich
Munich, Germany

Dennis Sprokholt
Technical University of Munich
Munich, Germany

Pramod Bhatotia
Technical University of Munich
Munich, Germany

Abstract

Neutral Atom (NA) Quantum Processing Units (QPUs) are emerging as a popular quantum computing technology due to their advantages (e.g., large qubit counts and flexible connectivity). However, a key performance trade-off exists: large circuits suffer significant fidelity drops, yet moderate-sized circuits, typical of multi-tenant cloud workloads, underutilize available hardware and are dominated by initialization latency. These issues result in inefficient hardware utilization and limit overall system performance. To address this challenge, we propose *multi-programming on NA QPUs*, i.e., co-executing multiple circuits on the same QPU, enabling increased resource utilization while preserving result fidelity through efficient compilation and hardware orchestration.

Yet, multi-programming techniques from classical and superconducting systems do not transfer to NA, which lacks the isolated functional units they depend on. To address this gap, we propose MultiQ, the first system to enable multi-programming on NA QPUs. MultiQ addresses three key challenges: *(i)* To maximize spatio-temporal hardware utilization, we introduce *virtual layouts*, and we bundle multiple virtual layouts to fit the available hardware optimally. *(ii)* To maximize throughput, we parallelize the execution of co-located circuits, making a single hardware instruction operate across independent circuits. *(iii)* To ensure parallelization did not introduce unwanted behaviors, we employ an algorithm that uses ZX-calculus to check whether the bundled circuits are functionally independent.

Our results show a throughput increase of up to 12.3× when multi-programming 14 circuits. MultiQ strongly preserves individual circuit fidelity, from a 1.3% improvement to a small loss of 3.5% for 14 circuits. Overall, MultiQ effectively co-executes multiple quantum circuits on a given hardware QPU, increasing throughput and hardware utilization.

1 Introduction

Quantum computing promises significant performance increases for key problems, such as integer factorization and quantum chemistry simulations [5, 65]. A variety of physical platforms, including superconducting [80], trapped ions [35], and neutral atoms (NA) [11], aim to realize this potential.

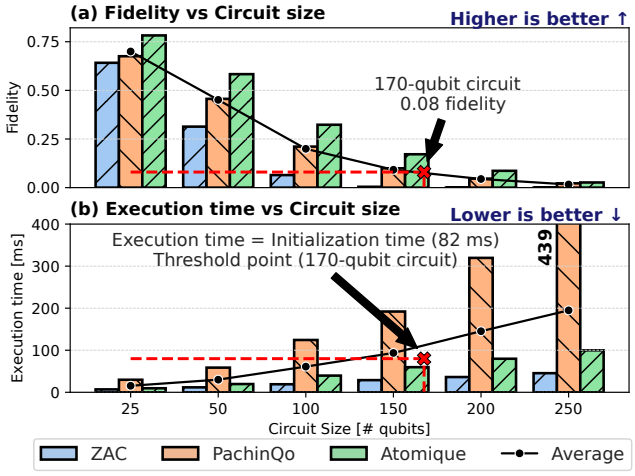


Figure 1. Limitations of neutral atom QPUs evaluated using state-of-the-art NA compilers (ZAC [48], PachinQo [53] and Atomique [93]). Average of GHZ, DJ, and W-State benchmarks. *(a) Fidelity drops sharply with circuit size, leading to QPU underutilization. (b) Execution time is lower compared to QPU initialization time for circuits up to 170 qubits.*

Among these, NA Quantum Processing Units (QPUs) are emerging as a leading technology [24, 71], offering several advantages, including long coherence times [25, 66], flexible connectivity with dynamic trap reconfiguration [9, 13], native multi-qubit gates [11, 30], and the scalability to hundreds or even thousands of qubits [55, 81]. Despite these advantages, current NA QPUs face two core limitations: low fidelity and low throughput, as we empirically demonstrate below.

The fidelity problem. Current NA QPUs suffer from relatively high operation noise, as each execution step (quantum gate) is not perfect, leading to small errors accumulating throughout execution. Fidelity quantifies how close the observed output is to the theoretical ideal, on a scale from 0–1, where 1 denotes a noiseless result. As circuits widen, they typically contain proportionally more gates; since each gate introduces error, fidelity drops sharply as circuit size increases, making much of a large QPU effectively unusable. Figure 1 (a) highlights this issue: on a 250-qubit device, with state-of-the-art compilers [48, 54, 93], on average, estimated fidelity falls

below 0.5 for circuits exceeding 50 qubits—just 20% of the total 250 available qubits. Consequently, quantum cloud environments are dominated by a highly heterogeneous mix of moderate-sized circuits.

The throughput problem. NA QPUs face throughput limitations from two factors: on one side, the fidelity problem restricts the hardware space that can be effectively utilized, and on the other side, NA QPUs have a time-consuming initialization process that creates a relatively high execution latency. Specifically, a NA initialization procedure must run before every circuit execution, starting by loading atoms into a vacuum chamber, imaging them, and sorting them into their correct positions [10, 77]. These tasks incur latency of tens of milliseconds [78, 98]. Figure 1 (b) illustrates this problem: initializing a 250-qubit NA QPU amounts to ~82 ms (blue dotted line) [78, 98]. In contrast, the actual circuit execution (black solid line) typically takes at most tens of milliseconds. As a result, the total runtime is dominated by QPU initialization overhead rather than the computation itself. Additionally, Figure 1 (a) shows that a circuit of 170 qubits would achieve ~0.08 fidelity, which is mostly unusable.

In summary, as NA QPUs scale rapidly, executing heterogeneous, moderate-sized circuits sequentially severely underutilizes the hardware, and initialization latency is a bottleneck.

A well-known technique for improving hardware utilization in classical computing is multi-programming: executing multiple independent programs concurrently on shared hardware. Multi-programming offers a natural fix for both problems above: co-executing several circuits on the same QPU increases the fraction of the array that is active and amortizes initialization costs across co-scheduled circuits, directly countering the throughput bottleneck. However, classical and superconducting multi-programming [21, 28, 51] rely on a fundamental guarantee: isolated functional units. Classical multithreading isolates state via private registers, and superconducting compilers schedule individually addressable qubits, with crosstalk mitigated by physical separation.

Bringing multi-programming to NA QPUs breaks this isolation assumption because two-qubit gates and atom shuttling both act through resources shared across multiple circuits (§ 2). Co-located circuits are not operationally isolated; uncoordinated instructions can physically collide or unintentionally entangle neighboring atoms, meaning operations must be explicitly orchestrated rather than simply interleaved. Furthermore, packing circuits into shared hardware introduces a unique fidelity-throughput trade-off (§ 3.1.2): denser packing increases throughput but lengthens shuttling paths, degrading fidelity. Consequently, existing spatial allocation, instruction scheduling, and correctness-checking techniques cannot simply be ported; they must be explicitly redesigned for the physically coupled NA hardware.

Realizing multi-programming on NA QPUs requires solving three key challenges that are unique to this architecture:

1. **Maximizing spatio-temporal hardware utilization.** Multi-programming requires allocating execution space, balancing circuits' performance against spatial footprint. Optimizing spatial (allocated space) and temporal (execution time) utilization creates a complex packing problem. 2. **Maximizing instruction parallelization.** Serialization of circuit execution prevents operation collisions but sacrifices parallelism and increases decoherence. Resources must be safely shared to parallelize circuit co-execution. 3. **Preserving functional independence.** Cross-circuit shared resource usage can easily increase edge cases and the probability of compilation errors. Multi-programmed circuits must yield the same results as in isolation, requiring reliable functional checks. We capture those challenges in the following question:

Research Question

How to multi-program NA QPUs, maximizing throughput, minimizing fidelity loss, and ensuring functional independence?

MultiQ: A compiler-runtime co-design. We introduce MultiQ, the first compiler-runtime system that efficiently realizes multi-programming for NA QPUs, addressing the three problems above through a co-designed compiler, runtime controller, and functional-independence checker. The compiler introduces virtual layouts for independent circuit compilation, balancing footprint against fidelity; the controller bundles and places these layouts to maximize spatio-temporal utilization while orchestrating shared AOD/Rydberg resources; and the checker certifies functional independence via the ZX-calculus. We integrate MultiQ with existing toolchains, including Qiskit for basic circuit optimizations [68] and the ZAC compiler [48] for solo circuit compilation. Our results, based on 11 standard applications, demonstrate that MultiQ delivers a significant throughput improvement, ranging from 3.8× to 12.3× by multi-programming 4 to 14 circuits, respectively. Additionally, it maintains high individual circuit fidelity, achieving an improvement of 1.3% with multi-programming four circuits and a minimal loss of 3.5% across 14 circuits.

Contributions. MultiQ makes the following contributions:

1) **Efficient NA multi-programming**: MultiQ is the first framework to efficiently and scalably co-execute multiple circuits on NA QPUs, while preserving the fidelity of individual circuits.
2) **Novel virtual zone layout**: We introduce the concept of a *virtual zone layout*, enabling independent compilation and optimization of multiple quantum circuits, allowing circuit bundling before being assigned to specific hardware.
3) **Instruction-parallelization optimizations**: We present new instruction-parallelization optimizations that enhance circuit fidelity in both solo and multi-programming environments, while existing compilation methods are unaware of multi-programming.

4) **Functional independence checker**: We present the first method to systematically check functional independence between multi-programmed quantum circuits, ensuring multi-programmed circuits behave as in solo execution.

2 Neutral Atom (NA) Architectures

2.1 Quantum Computation

A quantum computation is a *circuit* that applies a sequence of *gates*—single- or multi-qubit operations, each a rotation (or a combination of rotations) about the x, y, or z axes—to a set of qubits, and concludes by measuring them. While quantum computing can be realized on various hardware technologies, this paper focuses on NA technology, whose capabilities and limitations we detail next.

2.2 NA Architectures and Characteristics

NA quantum architectures utilize arrays of NAs, commonly alkali species such as rubidium, cesium, or strontium, which are excited into high-energy Rydberg states to encode qubits [15, 37, 72]. Atom arrays are held in place by static spatial light modulators (SLMs). This architecture enables multi-qubit gates and supports dynamic qubit rearrangement through acoustic-optical deflectors (AODs), allowing practical all-to-all qubit connectivity. Two-qubit gates are typically realized using an optical beam, also known as an entanglement pulse. To reduce crosstalk and noise between non-interacting atoms, modern architectures divide the system into distinct zones for entanglement, storage, and readout, thereby restricting entanglement and readout pulses to their respective zones.

Monolithic vs. zoned layouts. The hardware of common NA QPUs can be set up using either *monolithic layouts*, where all operations share a single zone, or *zoned layouts*, where atom arrays are physically separated into three different zones: *storage*, *entanglement*, and *measurement*, as shown in Figure 2 (a). Zoned architectures, which MultiQ uses, are increasingly preferred for improving fidelity by isolating idle qubits and allowing mid-circuit measurements [53, 83].

Gate operations. NA QPUs natively support single- and two-qubit gates. Two-qubit gates are based on the Rydberg blockade mechanism, as illustrated in Figure 2 (a): two atoms inside each other's blockade radius ($2-4\ \mu m$) cannot both be excited to Rydberg levels, enabling a controlled-Z entanglement gate [13, 75]. Single-qubit gates can be applied locally or globally. Local gates are limited to rotations around the Z-axis, which can be applied to multiple atoms selected using AOD lasers, while adhering to the AOD targeting rules [11, 25, 30]. As illustrated in Figure 2 (b), diagonal atoms cannot be selected without selecting the atoms on the opposing diagonal, and Figure 2 (c), the top row cannot cross the bottom row. In this work, we focus on single- and two-qubit gates.

Laser and trap system. NA arrays use two optical trap systems [9]. Spatial light modulators (SLM) create arbitrary 2D trap patterns to statically hold atom arrays, while acousto-optic deflectors (AOD) enable dynamic repositioning of qubits

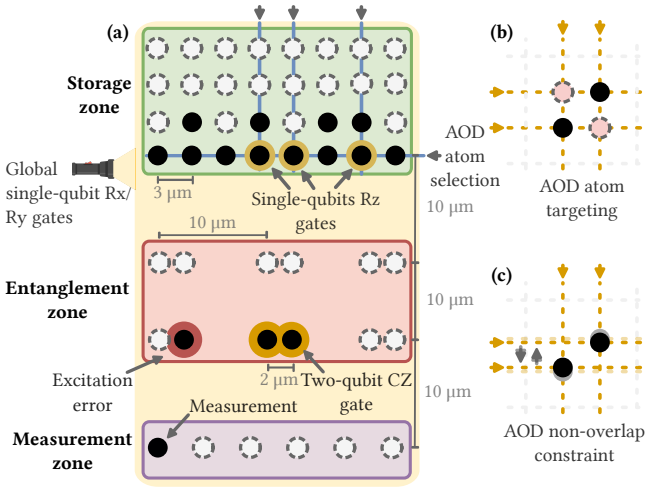


Figure 2. Neutral atoms architecture basics (§ 2) *Storage, entanglement, and measurement zones distributions and common spacings. Single-qubit gates and two-qubit gate operations. AOD laser targeting and the non-overlapping constraints.*

at runtime [11, 24]. Figure 2 (a) shows grids of SLM traps (white and black circles) and AOD lasers. A single AOD laser can manipulate multiple rows and columns of atoms in parallel [14, 26, 76]. AOD lasers are subject to constraints such as active lasers cannot cross over each other, or diagonally targeting can select unwanted atoms [13, 88, 98].

Initialization procedure. Initialization in NA QPUs contributes significantly to the overall runtime, as we show in Figure 1 (b). The process begins by loading atoms into a vacuum cell in which approximately 50% of the SLM trapping sites will be filled during this initial loading phase [77, 95]. The atom array is then imaged to determine the coordinates of the scattered atoms. Then, the sorting algorithm generates a set of arrangement instructions to build the desired atom layout [95], and finally, a second image is taken to verify the correct construction of the grid [14, 26, 98]. Notably, initialization has a constant cost that depends only on the atom array dimensions; it is independent of the size of the circuits to be executed.

Qubit movement. Qubit shuttling is essential in zoned NA architectures because qubits must be physically transported between spatially separated zones, for example, from the storage zone to the entanglement zone for two-qubit gate execution, and back again afterward. Shuttling is enabled by mobile optical tweezers (AOD lasers), effectively achieving near-perfect fidelity when performed below a speed limit [11, 88]. However, shuttling operations must avoid collision scenarios in which two AOD lasers come within a safe distance of each other, thereby increasing the risk of atom loss [13, 98]. Atoms can be transferred between SLM and AOD traps with ~99.9% fidelity, enabling complete dynamic reconfiguration [9, 88].

Programming primitives. Every NA program, normally, reduces to a schedule over four operations: `init`, which loads atoms into their starting positions; single-qubit rotation (`u3`)

(§6.3); `rydberg`, which applies a controlled-Z gate to every atom pair within blockade range inside the entanglement zone; and `move`, which shuttles atoms between zones.

3 Motivation

MultiQ mitigates the problem of QPU underutilization and low throughput by introducing multi-programming to the NA technology. Intuitively, *throughput* is the average number of circuits executed per time unit, and multi-programming increases it by co-scheduling multiple circuits onto the same grid, co-executing them without repeated initialization. However, multi-programming may also degrade *fidelity*, the closeness of the observed result to the ideal, since operational resources must be shared, increasing qubit decoherence.

MultiQ's core objective is summarized as follows:

Problem Statement

MultiQ multi-programs quantum circuits aiming to simultaneously maximize QPU throughput while preserving the total fidelity of the original circuits.

3.1 Design Challenges and Key Ideas

To address our problem statement, our design builds on several key ideas that each solve a technical challenge.

3.1.1 Hardware Utilization. To simultaneously execute multiple quantum circuits on a QPU, each circuit must first be mapped to a *layout*: the physical region of hardware space exclusively allocated to it. The layout affects both throughput and fidelity (Figure 3(a)). Narrow layouts (orange) use space better, fitting more circuits, but run slower due to long shuttling paths; wider layouts (blue) reduce shuttling but fit fewer circuits; balanced layouts (green) offer a middle ground. Figure 14 (a) (Appendix A) confirms this experimentally, comparing the relative fidelity of narrow (1:4) and wide (4:1) layouts against a square (1:1) one. Circuits must then be bundled to best fit the QPU space (maximizing spatial utilization) and matched by runtime (maximizing temporal utilization), as in Figure 3(b).

In summary, maximizing throughput and fidelity requires: (i) balancing single-circuit performance against a smaller layout footprint, (ii) ensuring each bundle best utilizes the QPU space and time resources. We thus phrase this challenge as:

Challenge #1. *How can we efficiently allocate space regions for multiple circuits and bundle them, while maximizing throughput and preserving fidelity?*

We address this challenge at two levels: *(i)* First, for each circuit, we compute a *virtual layout* that balances a layout footprint and high circuit performance. *(ii)* Second, given a large collection of circuits, we must bin them in such a way that each bin will fit on the QPU, maximizing throughput across all bins.

Fidelity and throughput of multi-programmed circuits. We aim to minimize the layout's footprint while maximizing

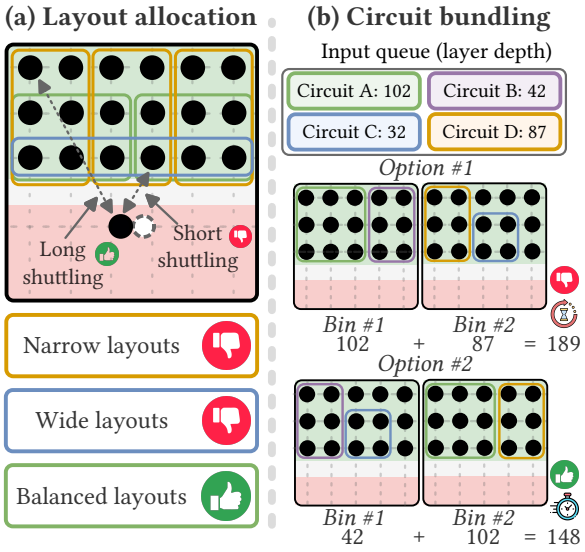


Figure 3. (a) Layout allocation: narrow (orange), wide (blue), and balanced (green) layouts, shown over the storage zone (green background). **(b) Circuit bundling:** packing the input queue into execution bins; the depth-aware *Option #2* cuts total runtime from 189 to 148 over the naive *Option #1*.

circuit performance (low runtime and high fidelity) for multiple circuits. However, exact fidelity is difficult to compute since it requires a circuit to be compiled first, which is especially difficult when multiple layouts need to be considered. Instead, we make the following observation:

Key idea #1A. Fidelity is primarily influenced by the width of a circuit, while throughput is primarily affected by the spatial utilization of QPUs. Computing exact throughput and fidelity for every candidate layout is prohibitively expensive; therefore, we use spatial utilization and temporal utilization as computable proxy metrics for throughput and fidelity, respectively. We use these insights to produce a *virtual layout* of the circuit that balances hardware utilization and fidelity, independent from the QPU hardware.

We implement this key observation in practice by heuristically navigating the space characterized by circuit width and spatial utilization. For a layout ℓ of width ℓ_w, we balance its estimated fidelity $F(c,\ell_w)$ (see Definition 2 in Appendix A) against its spatial utilization $S(\ell_w) = \ell_w / W_{\text{QPU}}$ (with W_{QPU} the total QPU width), selecting the layout that maximizes $\alpha \cdot F(c,\ell_w) + (1-\alpha) \cdot S(\ell_w)$, where $\alpha \in [0,1]$ trades off fidelity against throughput.

QPU utilization across all multi-circuits. At the global level, we are interested in the optimal set of bins for a given collection of circuits. In a multi-tenant quantum cloud environment, each QPU receives more circuits than it can execute concurrently, requiring us to partition the input queue into temporally separated bundles [28, 50]. Each bundle must fit within the QPU's spatial constraints, and its execution time

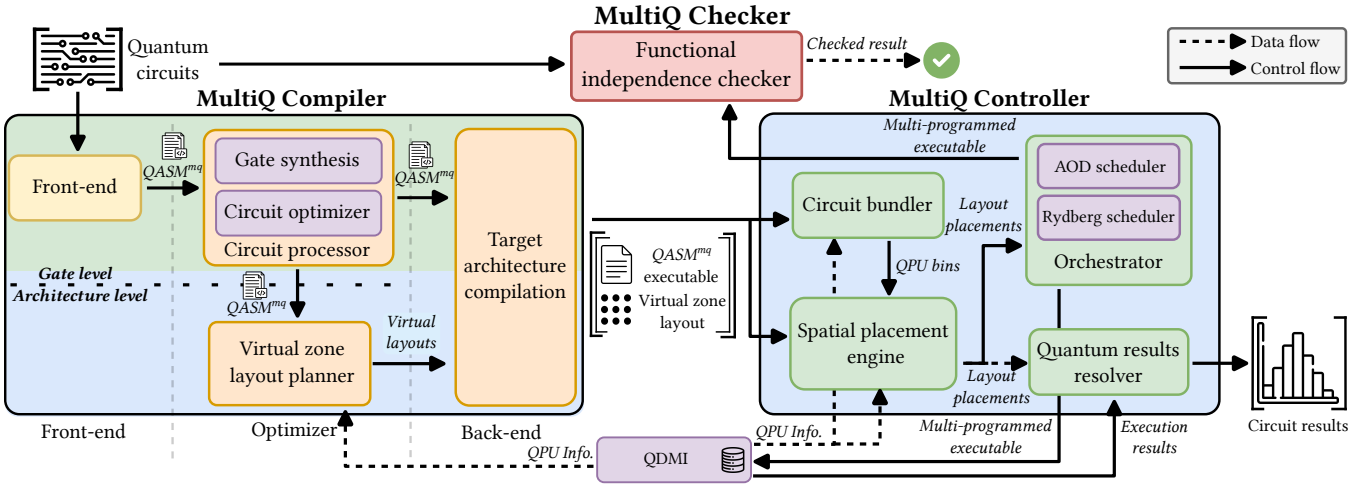


Figure 4. Overview of MultiQ (§ 4). *MultiQ is a co-designed compiler-controller system. The compiler (§ 5) optimizes and compiles quantum circuits. The controller (§ 6) then maps and efficiently multiprograms them on the hardware. A functional independence checker (§ 7) verifies that the instructions maintain circuit functionality.*

is dictated by the longest-running circuit within it. Consequently, the bundling strategy directly impacts total execution time. Figure 3 (b) exemplifies this effect: the naive FIFO bundling (Option #1) leads to higher total runtime compared to a latency-aware alternative (Option #2), despite both achieving identical spatial utilization.

Unlike the layout selection above, here we must reduce the *total* execution time across all circuits. We therefore reason about the spatial and temporal utilization of an entire bin rather than a single circuit, and seek the bundle that maximizes their weighted sum (again tuned by α) while fitting within the QPU width (we formalize both utilizations and this objective in §6.1). The difficulty lies in decreasing the computational complexity of exploring all possible bundle combinations, for which our key idea is:

Key idea #1B. MultiQ abstracts this problem to enable the use of simulated annealing to efficiently search the solution space of hardware-fitting circuit bundles, converging on a near-optimal grouping that maximizes both spatial and temporal QPU utilization.

3.1.2 Parallel Execution. Maximizing instruction parallelism between the multi-programmed circuits is essential to avoid trivial instruction sequentialization, which would lead to state decoherence. This is challenging due to the inherent NA hardware constraints on simultaneous single-qubit gates, entanglement pulses, and the concurrent movement of multiple atoms. Figure 2 (b) shows that AOD lasers target all the atoms in the intersections of the horizontal and vertical lasers, which can lead to unintentional atom targeting (pink circles). Moreover, Figure 2 (c) shows the AOD overlapping constraints, where they require a minimal distance to prevent frequency interference. Lastly, as explained in § 2.2, single-qubit rotations can only be applied in parallel, row-wise, and only around the same axis.

Our goal is to achieve ideal parallelization, where the circuits in a bin execute fully in parallel, and the bin finishes in the time of its longest circuit, while respecting instruction dependencies and hardware constraints.

Challenge #2. *How can we efficiently parallelize instructions in a multi-programming environment to execute all co-scheduled circuits in the least amount of time?*
Key idea #2. MultiQ approaches this NP-hard problem in a greedy manner by producing a dependency and constraint graph, from which we can extract the largest set of executable instructions.

3.1.3 Correctness. Multi-programming performance requires maximizing parallelism by resolving conflicts with hardware constraints (§ 3.1.2). However, such transformations risk altering the program's semantics or introducing unintended interference between co-executing programs. Unlike classical compilation, where correctness is typically ensured by well-defined static rules, quantum multi-programming must account for entanglement, gate noncommutativity, and shared physical resources. Ensuring correctness in this setting demands new abstractions and safeguards that reason about inter-program interactions. The explicit challenge and our key idea addressing it are as follows:

Challenge #3. *How can we ensure correctness in optimized circuit multi-programming by guaranteeing functional independence of each of the individual circuits?*
Key idea #3. MultiQ solves this by taking advantage of the quantum reversibility property as well as the ZX-calculus circuit processing capabilities. Functional independence is checked by simplifying, through ZX-calculus [17], a concatenated circuit composed of the ZX-diagrams of the original and the multi-programmed versions of a circuit. An empty circuit ensures functional equivalence.

4 MultiQ Overview

We propose MultiQ, which addresses the challenges of efficiently executing multiple circuits on a single QPU while preserving fidelity, reducing latency, and increasing throughput. MultiQ consists of three main components: the compiler, the controller, and the checker, as illustrated in Figure 4. Next, we explain the function of each main component and how it realizes each key idea.

MultiQ compiler (§5). The compiler starts by independently generating a *virtual zone layout* for each incoming circuit, balancing circuit fidelity and layout footprint, where larger footprints result in lower spatial utilization (key idea #1A). Afterward, the circuit can be target-compiled to the respective virtual layout. We optimize circuits at the gate level in our QASM dialect (QASM^{mq}), while concurrently generating architecture-level virtual layouts.

MultiQ (runtime) controller (§6). The controller starts by bundling circuits and their respective virtual layouts into execution bundles using a heuristic algorithm that optimizes the spatial and temporal utilization of hardware resources (key idea #1B). Secondly, given a set of formed bundles, the controller determines the near-optimal placement of the circuits within a bin to minimize instruction contention. The *Orchestrator* schedules hardware resources to produce a non-conflicting multi-programmed executable that executes all the bundled circuits simultaneously (key idea #2). Finally, the *Quantum results resolver* maps the results back to their original circuits, delivering separated outcomes to users.

MultiQ checker (§7). The functional independence checker ensures the input is semantically equivalent to its embedding in the multi-programmed circuit. This check is necessary to ensure that compiler transformations do not inadvertently alter the behavior of the multi-programmed executable.

We check this by reversing the multi-programmed executable, concatenating it with the solo input circuit, and iteratively eliminating canceling gates (key idea #3). We ensure functional independence if the result is an identity circuit.

5 MultiQ Compiler

The MultiQ compiler optimizes individual circuits and generates virtual layouts that balance circuit fidelity and utilization.

5.1 QASM^{mq}: MultiQ IR

MultiQ's front-end translates circuits from libraries like Qiskit [36] or Cirq [29] into QASM^{mq}, our IR, extending OpenQASM [20] with NA-specific annotations (e.g., `@init` for atom initial placement, `@move` for AOD movement). Table 2 lists available operations; Appendix B formalizes the IR grammar.

5.2 Virtual Zone Layout and Planning

Circuit execution time and fidelity depend on the physical qubit arrangement, but capturing how layout dimensions affect these metrics involves competing objectives (fidelity, spatial utilization, temporal utilization) with no closed-form solution. We define a formula balancing two boundary layouts: a minimum layout for dense packing (maximum QPU utilization) and an optimal layout for best performance (larger footprint, fewer co-execution opportunities). Figure 5b illustrates both layouts for a four-qubit circuit with up to three concurrent entanglement operations, showing both single- and double-storage configurations.

Layout width. We must select a width that can fit at least the number of qubits N_q in the storage rows N_r (the qubit-row count within a storage zone, a hardware-determined quantity distinct from the number of storage-zone *sides* $R \in \{1,2\}$ used by the controller's placement, §6.2), making the minimum width $W_{min} = \lfloor N_q/N_r \rfloor \cdot S_s$, where S_s is the storage atom spacing. However, a larger circuit often offers more parallelization opportunities, which benefits performance; for instance, by storing all qubits in a single row, it requires a width $W_s = (N_q - 1) \cdot S_s$. A circuit-layering algorithm (Algorithm 1, Appendix D) partitions the circuit into maximal parallel execution layers, respecting NA hardware capabilities and data-dependency constraints. This algorithm identifies W_e, the width for the largest concurrent entanglement operation. The best-performing layout width ($W_{best} = \max(W_s, W_e)$) thus offers maximal parallelism and entanglement.

The selected width is a weighted average: $W_{selected} = P_w \cdot W_{best} + (1 - P_w) \cdot W_{min}$, where $P_w \in [0,1]$ trades circuit performance (P_w=1) against QPU utilization (P_w=0). The optimized QASM^{mq} and virtual zone layout are then passed to the target-architecture compiler.

5.3 Back-end: Target Compilation

The final stage of our compiler produces the NA executable (gate schedule, laser pulse timing, and zone movements per layer). MultiQ remains agnostic to how this is implemented, delegating it to existing back-end compilers, such as ZAC [48], or PachinQo [54], via a compiler abstraction layer.

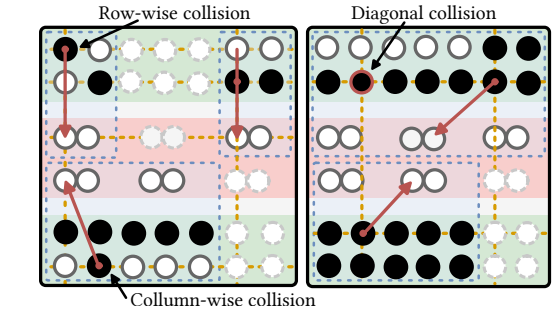
Target architecture compiler output. The compiler outputs tuples (QASM^{mq}, L) per circuit, pairing the QASM^{mq} executable with its virtual zone layout L; to this pairing of a circuit and its layout we refer to as a *tile*. MultiQ uses ZAC [48] as its default back-end; Appendix C provides the formal output and ZAC IR mapping.

6 MultiQ Controller

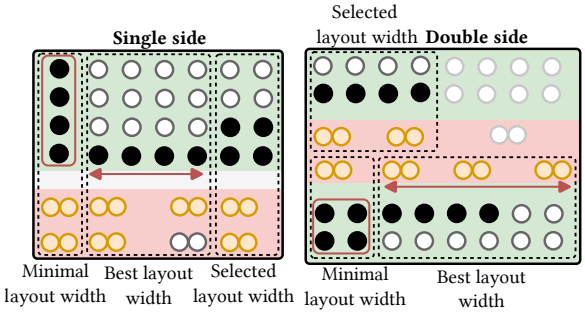
The controller is the multi-programming back-end, enabling the concurrent execution of the circuits produced by our compiler and with their virtual layouts (§5.3).

6.1 Circuit Bundler

Since the input tiles may not all fit on the hardware, the circuit bundler produces execution bins $B = \{B_1, \ldots, B_N\}$, where each bin $B_j = [c_0, \ldots, c_{n_j}]$ contains circuits that fit within the QPU. Naive bundling (*e.g.,* FIFO) often leads to suboptimal spatial and temporal utilization. Our bundler employs simulated annealing (SA) to jointly maximize spatial (S) and temporal (T)



(a) Examples of movement collisions (§ 6)



(b) Virtual zone planner (§ 5.2)

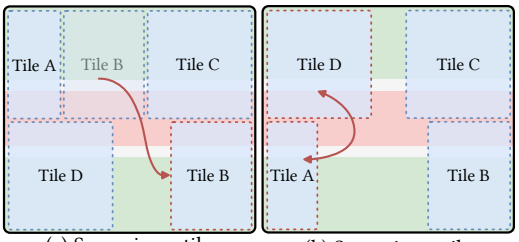
Figure 5. (a) *Movement collision types:* ***row-wise*** *(two movements ending on the same row),* ***diagonal*** *(ending on the same column), and* ***column-wise*** *(active AOD lasers cross an atom that must not move).* **(b)** *Minimal, best, and selected layouts*

utilization (Algorithm 2, Appendix D). SA minimizes an objective function by iteratively making small modifications in the bin assignment, always accepting improvements and probabilistically accepting worse modifications based on a temperature that cools over time, enabling escape from local optima and finding global solutions.

Spatial utilization (S_j). The spatial utilization for an execution bin B_j denotes the used proportion of the QPU area (from 0 to 1). When W is the total QPU width, and R is the number of storage-zone sides (one or two), the total QPU area is $W_{QPU} = W \cdot R$. The spatial utilization of bin B_j is $S_j = \sum_{i=0}^{n_j} w_i / W_{QPU}$, where w_i is the width of tile i. When $S_j = 1$, the tiles in B_j fit exactly the whole QPU space.

Temporal utilization (T_j). The temporal utilization captures timing differences between tiles in bin B_j, as large differences can lead to the QPU being underutilized while deeper executables are running. Let d_i be the depth of executable i, and $D_j = max_{i \in B_j}(d_i)$ be the depth of the deepest executable in bin B_j. The temporal utilization for bin B_j is $T_j = \frac{\sum_{i=0}^{n_j} d_i}{n_j \cdot D_j}$. When $T_j = 1$, the tiles in B_j all execute for the maximum duration D_j.

Simulated annealing objective. The objective function combines the spatial and temporal utilization for all bins, where we want to maximize $\mathcal{L} = \sum_{j=1}^{N} (\alpha \cdot S_j + (1-\alpha) \cdot T_j)/N$, where α weighs spatial vs. temporal utilization. SA starts from a FIFO distribution and at each step performs one of three actions: **(1)** move a tile to a new bin, **(2)** swap tiles between bins, or **(3)** move a tile to an existing bin.



(a) Swapping a tile with empty space

(b) Swapping a tile with another tile

Figure 6. Simulated annealing actions for placement generation (§ 6.2). *During tile placement, the SA algorithm takes one of two actions:* ***(a)*** *Swapping a tile with empty space or* ***(b)*** *Swapping a tile with another tile, as long as the modified tiles fit in their end positions.*

AOD laser constraints. After bundling all tiles, we must efficiently use shared resources. In particular, AOD lasers are shared resources where contention drastically impacts shuttle time. Atom movements must be compatible within and across tiles globally; Inside a tile, AOD movements must be checked for row and column compatibility. Conversely, across different tiles, AOD movements must be checked for compatibility on rows, columns, and diagonals in a global coordinate system. Figure 5a illustrates these rules, which are the base compatibility functions used by the placement generator and orchestrator.

6.2 Placement Generator

After bundling, MultiQ places each bin's tiles $B = \{t_1, \ldots, t_n\}$ on the hardware via our *collision-aware tile placement*. The QPU is modeled as a $R \times W$ grid ($R \in \{1,2\}$ storage-zone sides), with configurable column granularity that defines the placement search space granularity. The number of storage-zone sides R is either 1 or 2, depending on whether the QPU is configured with single- or double-storage. This restriction reflects zoned NA hardware, which provides a single entanglement zone [12, 25, 26, 98]: storage can be arranged only as a single zone adjacent to one side ($R = 1$) or split across both sides ($R = 2$), since no published device supports more than one entanglement zone. Extending beyond two storage sides would require multiple entanglement zones, introducing unverified assumptions. Each tile has a width $w(t_i) \in \mathbb{Z}$, measured in grid columns. Following the movement compatibility rules, each tile pair (t_i, t_j) has a conflict cost $C(t_i, t_j, p_i, p_j) \in \mathbb{R}_{\geq 0}$ representing the number of conflicts when placed at positions p_i, p_j. $z_i \in \{0,1\}$ indicates whether tile t_i is placed, with anchor position $x_i \in [0, W - w(t_i)], y_i \in \{0, \ldots, R-1\}$. When R=1, y_i collapses to a single value, so every tile is necessarily placed on the same side: no diagonal or row-wise collisions can happen, and placement reduces to ordering tiles left-to-right along the single available side via x_i alone. Only R=2 requires the placement search to additionally determine each tile's side and resolve conflicts between them. Since an efficient tile placement does not always yield optimal performance, sequentializing some operations may allow more tiles to fit on the QPU. Therefore,

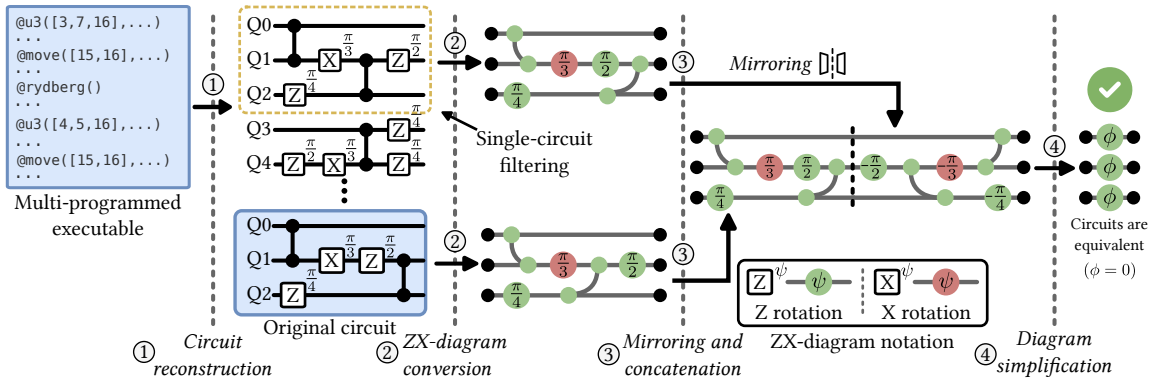


Figure 7. Checker workflow (§ 7). *The Checker takes as input the original circuit from the Compiler and the multi-programmed executable from the Controller.* ***(1)*** *The executable is reconstructed into a quantum circuit and constrained to qubit Q_k corresponding to the original circuit.* ***(2)*** *Both circuits are converted into their respective ZX-diagrams.* ***(3)*** *The ZX-diagram of the original circuit is concatenated with the mirroring of the ZX-diagram of the reconstructed circuit, swapping the signs of the rotation angles.* ***(4)*** *Finally, the concatenated ZX-diagram is simplified, which results in an empty circuit, meaning that both circuits are equivalent.*

we provide parameters α and β to control the trade-off between collisions and utilization (a higher $\beta:\alpha$ ratio favors throughput), such that the objective function becomes:

$$c^{\star\star} = min_{p_1,\ldots,p_n,z_1,\ldots,z_n}\alpha\cdot\sum_{i<j} z_i z_j \cdot C(t_i,t_j;p_i,p_j) - \beta\cdot\sum_{i=1}^{n} z_i$$

The placement algorithm (Algorithm 2, Appendix D) starts by greedily placing tiles by priority, then by width (smallest first). Then, at each step of the annealing phase, it alternates between two modifications (Figure 6): swapping a tile with empty space (a) or with another tile (b), subject to fit constraints. If this perturbation reduces the cost function, the swap is accepted; otherwise, the probability of acceptance is proportional to the temperature.

6.3 Orchestrator

Once tiles are placed, the orchestrator schedules NA resources to generate the multi-programmed executable. MultiQ operates in execution layers $L=\{L_1,\ldots,L_k\}$, each with four phases: **(1)** move atoms to the entanglement zone; **(2)** Rydberg pulse; **(3)** move atoms back; **(4)** apply single-qubit gates. To parallelize the forward and reverse movements across layers, we partition them into sub-rounds of compatible operations, modeled as a conflict graph $G^i=(M^i,E)$, where nodes represent NA operations and edges represent conflicts between them. The minimum of execution cycles equals the chromatic number $\chi(G^i)$. Since finding $\chi(G^i)$ is NP-hard, we employ a greedy approach that iteratively removes the maximum independent set until all NA operations are scheduled.

Single-qubit gates row optimization. When scheduling single-qubit gates, the orchestrator increases parallelization by leveraging a NA hardware capability that allows applying single-qubit gates to targeted atoms in the same row simultaneously (§ 2.2). However, only R_Z gates can be applied row-wise (R_Z^R) on targeted atoms. In contrast, R_Y rotations can only be applied globally (R_Y^G) to all the atoms in the array. Given the NA native single-qubit gate set (R_Z and R_Y), commonly used $U3(\theta,\phi,\lambda)$ gates, must be decomposed into a $R_Z^R(\phi)R_Y^G(\theta)R_Z^R(\lambda)$ gate sequence. Furthermore, since the middle $R_Y^G(\theta)$ needs to only affect targeted atoms, it needs to be further synthesized into the $R_Y^G(-\pi/2)R_Z^R(\theta)R_Y^G(\pi/2)$ gate sequence to maintain the state of the non-targeted qubits. The full $U3(\theta,\phi,\lambda)$ gate decomposition would be applied with the following row-optimized gate sequence: $R_Z^R(\phi)+R_Y^G(-\pi/2)+R_Z^R(\theta)+R_Y^G(\pi/2)+R_Z^R(\lambda)$.

7 MultiQ Checker

The checker ensures *functional independence*: each co-executing circuit c_k behaves identically under multi-programmed execution E_B as it would in isolation. Formally, circuit k is functionally independent if there exists a global phase ϕ such that:

$$U_k^{\text{multi}}\cdot(U_k^{\text{single}})^{\dagger} = e^{i\phi} I_k$$

where U_k^{single} and U_k^{multi} are the unitary transformations of circuit k in isolation and under multi-programming, respectively, and I_k is the identity on qubits Q_k (full formalization in Appendix A). We verify this condition leveraging quantum circuit reversibility: two circuits are functionally equivalent, iff composing one with the inverse of the other yields the identity. We check this using ZX-calculus, as detailed next.

7.1 Functional Independence Checker

We use ZX-diagrams because of their powerful simplification rules, enabling the concatenated diagram to be easily reduced, as formalized in Appendix A. If the simplified result is an empty graph, the identity operation, or a global phase, the circuits are functionally equivalent, confirming their functional independence. Appendix A formalizes this process. Figure 7 gives an example of the Checker workflow, as explained in more detail below:

#1: Circuit reconstruction. NA QPUs employ a relatively simple instruction set at the algorithmic level, which facilitates the recovery of the circuit semantics for each program in

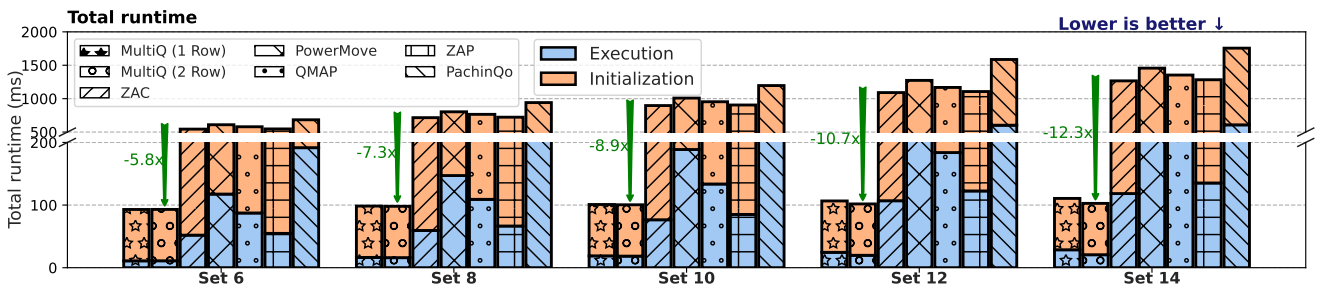


Figure 8. RQ#1: End-to-end total runtime evaluation comparing MultiQ, ZAC, PowerMove, Qmap, ZAP, and PachinQo.

the output executable. The checker statically analyzes which gates are executed by maintaining a virtual state of each atom and checking their positions for gate execution (e.g. within the Rydberg zone). In Figure 7, the example starts with a set of QASM^{mq} instructions that can be reconstructed into a circuit on the right. From this circuit, qubits Q_k are represented by the first three qubits inside the yellow box.

#2: ZX-diagram conversion. An interpretation function translates both the original $C_k^{original}$ and the multi-programmed circuit C_k^{actual} into ZX-diagrams using the standard circuit to ZX translation (provided by *PyZX* [67]). We aim to show that these diagrams are equivalent and represent the same transformations. In ZX notation, X rotations use red nodes, Z rotations use green nodes (with rotation angles), and CZ gates appear as two connected green nodes.

#3: Mirroring and concatenation. We create the inverse ZX-diagram with a "mirroring" operation of C_k^{actual}, where the inputs become outputs (and vice-versa) and negating the phases for all Z- and X-spiders. The two diagrams $C_k^{original}$ and $(C_k^{actual})^\dagger$ are then concatenated together. This is represented on step 3 of Figure 7.

#4: Diagram simplification. Finally, a ZX simplification pass runs on the result: if it reduces to the identity or a global Z phase, we conclude $U_k^{original} \cdot (U_k^{actual})^\dagger = e^{i\phi} I$, confirming functional equivalence. The ZX-calculus rewrite rules are sound for {X,Y,CZ}, and provide strong practical guarantees for general circuits [18], providing a scalable approach that detects unintended interactions.

8 Evaluation

We structure the evaluation in four core parts: **full end-to-end** (§ 8.2), **compiler** (§ 8.3), **controller** analysis (§ 8.4), and **framework overhead** (§ 8.5).

8.1 Experimental Methodology

Baselines. Across all evaluations, we compare MultiQ against ZAC [48], PowerMove [70], ZAP [34], Qmap [82], and PachinQo [53], the state-of-the-art compilers for zoned NA architectures.

Benchmarks. We use 11 benchmarks from two standard benchmark suites [44, 69] (benchmarks (*BV, DJ, WST*) include two qubit-size variants). For reference, Appendix E includes a more detailed list of the used benchmarks. For fairness, we use benchmarks similar to those used by the baselines.

Table 1. QPU parameters used on MultiQ's evaluation, modeled on published hardware work [12, 25, 26, 98]

Parameters	Value
Single- and Two-qubit gate fidelity	0.9991, 0.995
Atom transfer time and fidelity	$17\mu s$, 0.999
T_2 decohrence time	$1.5s$
Atom movement max speed	$0.55\mu m/\mu s$
Atom movement acceleration	$2750m/s^2$
Single- and Two-qubit gate time	$52\mu s$, $360ns$
Entanglement zone height	$40\mu m$
QPU size (width × height)	$230\mu m \times 155\mu m$

Fidelity model. Similarly to the baselines, we use an analytical model parameterized with published hardware error rates and gate/movement timing to estimate fidelity (Table 1). The fidelity model considers four main sources of error: one-qubit gate error (E_1), two-qubit gate error (E_2), atom transfer error (E_{trans}), and decoherence time (T_2). For a circuit with N qubits, where qubit i is subject to n_i gates with per-gate fidelity f_{gk} and idle time t_i under dephasing time $T2_i$, we estimate:

$$F_{total} \approx \prod_{i=0}^{N} \left(\prod_{k=0}^{n_i} f_{g_k} \cdot \exp\left(-\frac{t_i}{T_{2_i}}\right) \right),$$

QPU hardware setup. We evaluate the QPU architecture with single and double storage zones, as illustrated in Figure 5. We conservatively estimate the initialization overhead to be 82 ms for a 280-qubit QPU [42, 97]. The QPU hardware setup and parameters are detailed in Table 1.

Metrics. We evaluate MultiQ across five metrics: **(1)** fidelity (§ 8.1); **(2)** circuit execution time; **(3)** total duration, comprising execution time and QPU initialization time; **(4)** spatial utilization; and finally, **(5)** temporal utilization (§ 6.1).

8.2 Full System Evaluation

RQ1: *What is MultiQ's multi-programming runtime improvement?* We compare MultiQ's co-scheduled sets of 4 to 14 circuits against baselines' sequential (solo) execution, randomly bundling benchmarks without duplicate circuits within a set.

Total runtime improvement. Multi-programming 14 circuits reduces total system runtime by up to 12.3× (Figure 8). As noted in Section 1, initialization dominates total runtime;

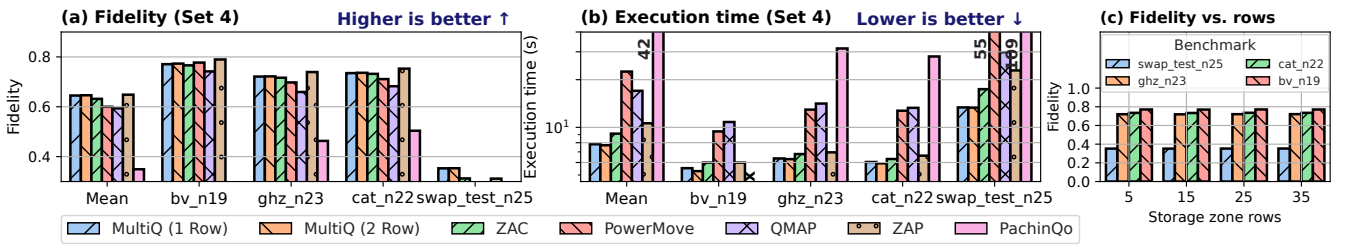


Figure 9. RQ#2: End-to-end evaluation: fidelity and circuit duration (§ 8.2). ***(a)*** *Fidelity of each benchmark co-executed by* MultiQ *(multiprogramming a set of four circuits) vs ZAC, Powermove, Qmap, ZAP, and PachinQo solo executions.* ***(b)*** *Circuit execution time of each benchmark as executed by* MultiQ*, ZAC, PowerMove, ZAP, PowerMove, and PachinQo.****(c)*** *Fidelity of* MultiQ *(ZAC backend) as the number of storage-zone rows available is varied, holding the benchmark and layout width fixed.*

thus, this gain is achieved mainly by collapsing 14 QPU initializations into one; parallel execution overhead is negligible compared to the cost of sequential initialization.

Robustness to continuous-loading architectures. Emerging continuous-loading NA hardware, which maintains an atom reservoir and avoids pausing for reinitialization between circuits, progressively reduces the speed-up benefit of avoiding circuit reinitialization. Section 8.4 (RQ5) isolates this case, reporting only execution time, comparing MultiQ against the baselines. MultiQ still reduces execution time by up to 2.1× on a set of 14 when compared to ZAC [48].

RQ2: *What is* MultiQ*'s multi-programmer's fidelity performance with respect to solo execution?* We assess MultiQ's fidelity and circuit execution time, comparing co-execution against solo execution, using the same methodology as RQ1.

Fidelity performance. Figure 9 shows fidelity and execution time for a set of 4 circuits. Table 4 (Appendix E) reports the same metrics for sets of 6 to 14 circuits, and Figure 17 (Appendix E) extends this plot to sets of 8 and 12 circuits. At this set size, MultiQ achieves both higher fidelity (64.51–64.62%) and lower execution time (7.71–7.82 ms) than ZAC (63.18%, 9.13 ms), trailing only ZAP's fidelity (64.85%) by a fraction of a point while remaining faster than ZAP overall. PowerMove, QMAP, and PachinQo show worse results than MultiQ on both metrics. For larger sets, MultiQ shows a slight drop in fidelity, averaging −2.6% compared to isolated ZAC's execution, with an additional +3.4 ms of execution time, indicating slightly higher decoherence error due to instruction contention.

Storage-zone row count. Figure 9 (c) reports MultiQ's fidelity (ZAC backend) varying the number of storage zones from 5 to 35. Fidelity remains effectively unchanged across this range. This is a consequence of how atom placement works, for MultiQ and similarly to all baselines: qubits are placed in the rows closest to the entanglement zone, minimizing shuttling distance (§ 3.1.1), reducing the number of available rows only removes rows that were not being used.

8.3 Compiler Evaluation

RQ3: *What is the effect of the virtual zone planner on trading off circuit performance and QPU utilization?* We evaluate this trade-off by running single circuits while varying the planner's performance weight, measuring decoherence error, and free QPU space. We selected six benchmarks that exhibit distinct trade-off behaviors.

Analysis of the virtual layout planner results. Figures 10 (a) and (b) show the tradeoff of increasing the performance weight on the layout planning formula explained in Section 5.2, as higher values in performance weight lead to wider layouts. Figure 10 (a) shows a slight decrease in decoherence error, in most benchmarks, (≤5%) as performance weight increases. On the other hand, Figure 10(b) reveals a sharp loss of free QPU space, as wider layouts reduce the number of circuits that can fit in a single execution bundle: average spatial utilization drops from 92% at spatial weight 0.8 to 65% at 0.2, especially for larger benchmarks. In conclusion, a spatial utilization weight of 0.6–0.8 yields narrow layouts with only 2% decoherence cost (vs. weight 1.0) but 20% higher spatial utilization.

8.4 Controller Evaluation

RQ4: *What is the effect of the circuit bundler on maximizing spatial and temporal QPU utilization?* Here, we investigate the effectiveness of the circuit bundler at grouping quantum circuits into execution bins. We compare the circuit bundler's effectiveness against a FIFO approach using sets of 6 to 14 circuits. We set up the planner to produce larger zone layouts (performance-focused layout planning), where one execution cycle would not fit all the circuits.

Analysis of the circuit bundler results. Figure 11(a) shows up to 3× higher QPU spatial utilization over FIFO (14 circuits, 0.8 spatial weight). As the algorithm prioritizes spatial optimization, it bundles circuits with better hardware-fitting layouts. On the other hand, Figure 11(b) shows only a ∼10% temporal utilization decrease between weights 0.8 and 0.2, suggesting spatial optimization does not strongly compromise temporal efficiency. We hypothesize this depends on the depth distribution of the input pool: when circuits have similar runtimes, the spatial-to-temporal weight ratio barely affects temporal metrics but strongly affects space utilization. Overall, the SA bundler achieves 2−3× higher spatial utilization and improved temporal utilization compared to FIFO.

RQ5: *What is the effect of the controller at layout placement and independent circuit execution parallelization?* We compare

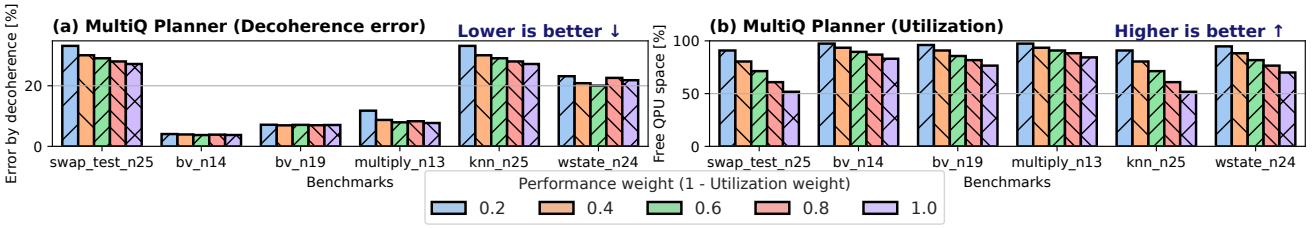


Figure 10. RQ#3: Virtual layout planner (§ 5) evaluation. ***(a)*** *Shows the effect of decoherence error on different benchmarks by increasing the performance weight on the formula in 5.2.* ***(b)*** *Shows the effect on free QPU utilization.*

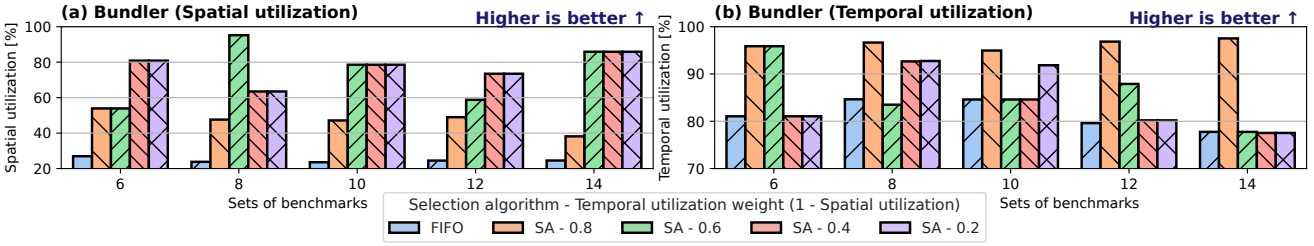


Figure 11. RQ#4: Circuit bundler (§ 6.1) evaluation. ***(a)*** *Shows the circuit bundler results of QPU spatial utilization for a FIFO and simulated annealing (with different cost weights on temporal utilization) algorithms.* ***(b)*** *Shows the effects of the different bundling of the algorithms on temporal utilization.*

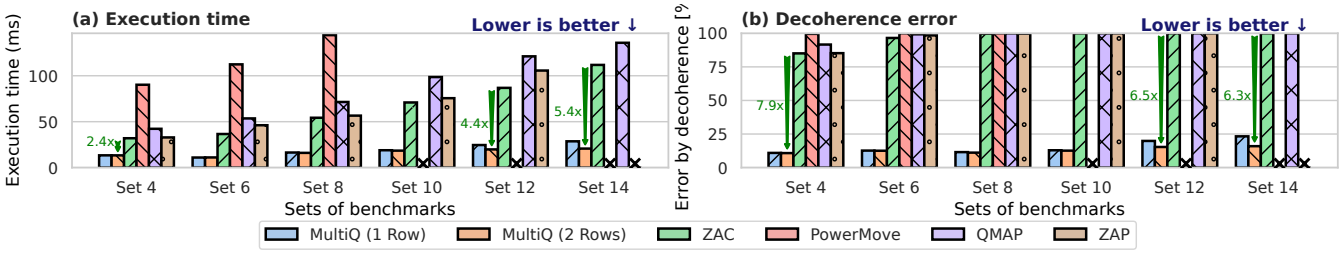


Figure 12. RQ#5: Controller's evaluation of execution time and decoherence error on increasing number of circuits (§ 8.4). ***(a)*** *Execution time for MultiQ's controller and ZAC compiler.* ***(b)*** *Decoherence error results.*

the controller's layout placement and execution parallelization against a naive circuit-merging approach, on random sets of 4–14 circuits. Unlike RQ1 and RQ2, the baseline here merges all circuits into a single parallel circuit rather than running them sequentially.

Parallelization performance. Figure 12(a) shows up to 5.2× reduction in execution time vs. ZAC. This leads to a strong decoherence error reduction, from near 100% (ZAC, ≥6 circuits) to ~15% (MultiQ, Figure 12(b)). ZAC is constrained by limited space and cannot spatially isolate circuits, leading to high resource contention and longer runtimes. PowerMove, QMAP, and ZAP show the same limitation, achieving execution time and decoherence error comparable to ZAC, since none of these compilers support multi-programming. Overall, MultiQ's placement and parallelization approach achieves substantial improvements in total execution time and decoherence errors compared to a naive circuit-merging solution.

8.5 Framework Overhead

RQ6: *What classical computational overhead does MultiQ introduce?* We measure wall-clock time across MultiQ's five pipeline stages (planning, bundling, backend compilation, placement, and routing) for single- (1R) and double-storage (2R) configurations, against a backend-only baseline (B).

Multiprogramming overheads. Figure 13 (a) shows the breakdown of the overheads for the same circuit sets 4 to 14 circuits used in the previous evaluation plots (single- and double-storage configurations). Planning, bundling, and routing (shared framework steps across both single- and double-storage-zone configurations) contribute negligible overhead throughout. However, SA placement (required only for double-storage configurations, § 6.2) grows substantially faster than the other framework steps as the number of circuits increases.

MultiQ's scalability. Figures 13 (b and c) show: MultiQ's scalability when multiprogramming a fixed set of 6 circuits with increasing sizes (20, 50, and 100 qubits per circuit); and MultiQ's scalability when increasing the target QPU size (fixed set of 6 circuits of 30 qubits each), respectively. Similarly to Figure 13, the classical overhead, when compared to a backend-compilation-only runtime (B), is dominated by the SA placement on double storage configuration (2R), while the overheads for single storage are at most 13% when multiprogramming 6 100 qubit circuits. On the other hand, increasing

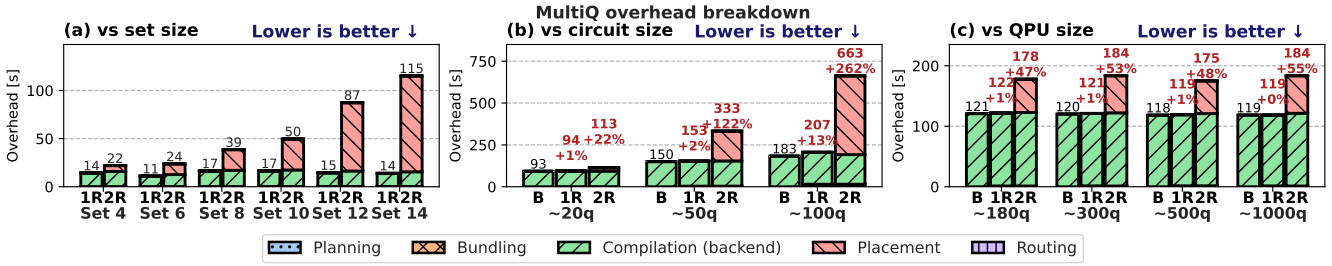


Figure 13. RQ#6: MultiQ's classical overhead breakdown for single-storage (1R) and double-storage (2R) configurations §8.5 ***(a)*** *Stages breakdown over sets of increasing number of circuits.* ***(b)*** *Stages breakdown of a fixed set of six circuits with increasing sizes.* ***(c)*** *Stages breakdown with a fixed set of six 30-qubit circuits, over increasing target QPU size.*

the target QPU size, thus increasing the layout placement search space, doesn't seem to increase the classical overheads.

In summary, while setting up MultiQ to operate in a double-storage configuration appears to offer slight fidelity benefits when multiprogramming multiple circuits together, with a 5% improvement for a set of 14 circuits, allowing for better collision-avoidant placements, the trade-off between classical overheads and fidelity improvement needs to be considered.

9 Related Work

Quantum compilers. Quantum compilers translate high-level quantum circuits into hardware-executable operations. Numerous compilers target superconducting qubits [33, 45, 49, 58, 59, 68, 79, 90, 91, 99], trapped ions [16, 31, 41, 56, 73, 74], and photonic systems [100–102], but are tied to their platforms' topological constraints, making them unsuitable for NA architectures. Existing NA compilers, in turn, either target static hardware or support only limited dynamic capabilities such as qubit shuttling or zoned layouts [6, 38, 40, 48, 52, 54, 62, 63, 84, 87–89, 93], yet none simultaneously exploit zoned NA architectures and circuit multi-programming. In contrast, MultiQ leverages state-of-the-art NA capabilities to efficiently co-schedule heterogeneous circuits in zoned NA architectures, maximizing QPU utilization with minimal fidelity loss.

Multi-programming QPUs. While multi-programming has been explored for superconducting qubits [21, 28, 50], migrating this concept to NAs introduces hardware-specific challenges (§ 1 and § 3), rendering previous superconducting approaches inapplicable. Closest to our work, Quantum Virtual Machines [92] also compile and schedule independent circuits onto shared hardware, but for superconducting qubits with per-qubit control; its virtualization layer does not need to reason about the shared, physically-coupled resources (AOD lasers, entanglement pulses) that drive MultiQ's placement and orchestration design (§ 1).

Quantum HW-SW co-design. Quantum hardware-software co-design improves application fidelity and resource efficiency across architecture design, error correction, and distributed computing [3, 46, 47, 85, 86, 94]. The vast majority of these efforts focus exclusively on superconducting QPUs running single-program workloads. MultiQ enables HW-SW co-design by tackling the complex spatial and temporal orchestration required for multi-tenant NA QPUs.

Formal methods in quantum computing. Formal methods in this domain primarily span formal verification (proving a circuit universally implements an intended mathematical function [2, 4, 43]) and equivalence checking (confirming two specific circuit instances are functionally identical). Exact equivalence checking is QMA-hard [39] and computationally exhaustive [61], MultiQ circumvents this bottleneck using ZX-calculus [23, 64]. We are the first to prove functional independence in a multi-programmed hardware environment.

SA placement in classical computing. SA itself is a well-established placement technique in FPGA placement [8] and CGRA mapping [27]. These formulations place isolated components optimizing for interaction distance or resource contention; they do not need to model collisions between shared hardware resources that arise when placed units are co-located, which is what MultiQ's placement objective captures.

10 Discussion

We present MultiQ, a compiler-controller co-design that enables high-throughput, fidelity-aware multi-programming on NA QPUs. MultiQ partitions and maps multiple circuits to non-overlapping QPU regions, co-optimizing for utilization, fidelity, and latency, while ensuring functional independence. Our evaluation shows that MultiQ improves QPU throughput by up to 12.3× with a 1.3% fidelity improvement to a 3.5% fidelity loss when co-executing 4 to 14 circuits.

Toward fault-tolerant integration. As FT circuits scale, compiling them monolithically will become prohibitively expensive, motivating parallel compilation of independent QEC subtasks (e.g., state preparation, T-state generation, syndrome extraction, and the main logical computation) rather than one circuit-wide pass. MultiQ's virtual layouts and orchestrator already support this: independently compiling and spatially co-scheduling separable units is exactly what FT compilation needs. Realizing it requires targeted extensions: physical qubits are expanded to logical code patches, the orchestrator (§6.3) needs to support inter-layout communication, and a

checker (§7) extended to mid-circuit measurement and classical feedback, an open problem beyond MultiQ. With these extensions, T-factories can even be shared across concurrently executing logical circuits, amortizing their overhead.

Artifact and Supplementary Material

MultiQ will be publicly released with our experimental setup. The appendices include formal definitions of key terms, our QASM^{mq} IR specifications, the target compiler output, pseudocode of the algorithms, and additional evaluation information and results.

Artifact Appendix

Abstract

Our artifact includes the MultiQ implementation (layout planner, bundler, controller, and ZX-calculus-based checker), the baseline NA compilers used for comparison (ZAC [48], PachinQo [53], PowerMove [70], ZAP [34], QMAP [82], and Atomique [93]), the benchmark circuits, and the scripts that reproduce all the figures and tables of the evaluation. A single script runs the experiments and plots the figures. The artifact runs on a standard CPU and requires no quantum hardware, since fidelity and execution time are estimated with the analytical model, as explained in Section 8.1. The artifact supports the paper's claims: up to 12.3× lower total runtime (higher throughput) when multi-programming 14 circuits, individual circuit fidelity preserved within +1.3% to −3.5% of solo execution, up to 5.2× lower execution time than ZAC in the controller evaluation, and 2–3× higher spatial utilization of the SA bundler over FIFO.

Artifact Checklist (Meta-information)

- **Algorithm:** MultiQ's virtual layout planner, SA-based circuit bundler, controller (placement, routing, and parallel scheduling), and ZX-calculus-based functional independence checker.
- **Program:** MultiQ's source code (Python). The source code of ZAC, PachinQo, PowerMove, ZAP, and Atomique is included in the artifact; QMAP is installed from PyPI (`mqt-qmap==3.8.0`).
- **Data set:** 11 benchmark algorithms (16 circuits, Table 3) from QASMBench [44] and MQT Bench [69], included in the artifact as QASM files.
- **Run-time environment:** Linux or macOS, Python 3.12 (and Python 3.10 for Atomique), managed with uv.
- **Hardware:** No specific hardware is required; all experiments run on a CPU. Our results were produced on a shared server with an AMD EPYC 7713P CPU (64 cores, up to 3.7 GHz) and 900 GB of RAM.
- **Execution:** The scripts compile every benchmark set with MultiQ and each baseline, then plot the results. The complete evaluation takes approximately 2 hours, most of which is spent on the overhead and bundler experiments.
- **Metrics:** Fidelity, circuit execution time, total runtime (circuit execution plus QPU initialization), spatial utilization, temporal utilization, and classical compilation times.
- **Output:** The figures and the table of the evaluation as PDF and LaTeX files in `results/plots/`, together with the raw results (CSV/JSON) in `results/`.
- **Experiments:** Figures 1, 8, 9, 10, 11, 12, 13, and 17, and Table 4.
- **How much disk space is required (approximately)?:** About 1.1 GB for the source code, benchmarks, and baselines, plus about 0.5 GB for each of the two Python environments.
- **How much time is needed to prepare the workflow (approximately)?:** About 5 minutes to download the artifact and install the Python environments.
- **How much time is needed to complete experiments (approximately)?:** About 2 hours on the server described above. The results shipped in `results/` allow regenerating every plot in a few seconds.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License. Each baseline keeps its upstream license (see `compilers/README.md`).
- **Archived (provide DOI)?:** 10.5281/zenodo.24197547

Description

How to access. The artifact is available at `https://doi.org/10.5281/zenodo.24197547`

Hardware dependencies. None. All compilers and the fidelity model run on a standard CPU.

Software dependencies. Python 3.12 and the uv package manager. The dependencies and their versions are pinned in `pyproject.toml` and `uv.lock`; the main packages include: Qiskit 2.x, MQT Bench, MQT QCEC, and MQT QMAP. Atomique requires Qiskit 0.38, which does not install on Python 3.12, so it runs as a subprocess in a separate Python 3.10 environment. uv downloads and manages both Python versions if they are not present.

Data sets. The benchmark circuits are in `data/benchmarks/single/`, and the benchmark sets used by each experiment are listed in `data/*_bench_list*.txt`. Circuits missing from the directory are regenerated with MQT Bench.

Installation

After downloading and extracting the artifact, install MultiQ and the baselines that share its environment (ZAC, PachinQo, PowerMove, QMAP, and ZAP):

```
cd MultiQ
uv sync
```

Then, create the separate environment for Atomique (only needed for Figure 1):

```
uv venv --python 3.10 compilers/atomique/.venv
uv pip install --python compilers/atomique/.venv/bin
    /python -r compilers/atomique/requirements-runner.txt
```

Experiment Workflow

All experiments are run by `scripts/run_evaluation.py`. To run the complete evaluation:

```
uv run python scripts/run_evaluation.py --all
```

The option `-only NAME[, NAME...]` runs a subset of the experiments (`-list` prints them), `-data-only` runs the compilers without plotting, and `-plots-only` regenerates the plots from the results already in `results/`. The terminal only shows the experiment and run in progress; the full compiler

output is written to `results/logs/<experiment>.log`. The experiments are executed in the following order:

1) **`introduction`** [≈ 5 min]: Compiles circuits of increasing size with ZAC, PachinQo, and Atomique (Figure 1).
2) **`rows_sweep`** [≈ 1 min]: Runs MultiQ with an increasing number of storage-zone rows (Figure 9(c)).
3) **`e2e`** [≈ 17 min]: Compiles every benchmark set (4 to 14 circuits) with MultiQ (single and double storage) and all baselines (Figure 9(a,b)). The experiments `e2e_total_runtime` (Figure 8), `e2e_means_table` (Table 4), and `e2e_detailed_full` (Figure 17) only plot the output of this experiment and take a few seconds.
4) **`planner`** [≈ 11 min]: Varies the performance weight of the virtual layout planner (Figure 10).
5) **`bundler`** [≈ 32 min]: Compares the SA bundler against FIFO for different spatial weights (Figure 11).
6) **`controller`** [≈ 5 min]: Compares MultiQ's controller against merging the circuits and compiling them with each baseline (Figure 12).
7) **`overhead_circuit_size`** and **`overhead_qpu_size`** [≈ 54 min]: Measure MultiQ's classical overhead for increasing circuit and QPU sizes. `overhead_combined` plots them together with the `e2e` data (Figure 13).

Evaluation and Expected Results

All plots and tables should closely match their counterparts in the paper. The planner and bundler use simulated annealing, and some baselines rely on time-limited heuristics, so a new run may differ slightly from the paper; however, fidelity and execution time should stay within about 5% of the reported values. The compilation times in Figure 13 depend on the CPU and on the machine load, so their absolute values can differ by more than 5%. The total runtime speedups over ZAC (Figure 8) are also stored in `results/logs/e2e_total_runtime.log`.

Experiment Customization

MultiQ can compile other circuits and target other QPUs. It takes a list of QASM files and a YAML configuration file (see `config/multiq/e2e_config.yaml`), whose main parameters are:

- `qpu_width`, `qpu_height`: QPU size in μm.
- `grid_rows`, `grid_cols`: multi-programming grid.
- `fidelity_{1q_gate,2q_gate,atom_transfer}`: hardware error model.
- `perf_weight`: weight of single-circuit performance against spatial utilization in the planner (Section 5.2).
- `selection_algorithm`: bundling algorithm, `fifo` or simulated annealing (`sa`).
- `enable_verification`: run the checker on the compiled program.

References

[1] [n. d.]. Quantum Complexity Theory | SIAM Journal on Computing. https://epubs.siam.org/doi/10.1137/S0097539796300921

[2] Parosh Aziz Abdulla, Yo-Ga Chen, Yu-Fang Chen, Lukáš Holík, Ondřej Lengál, Jyun-Ao Lin, Fang-Yi Lo, and Wei-Lun Tsai. 2024. Verifying Quantum Circuits with Level-Synchronized Tree Automata (Technical Report). doi:10.48550/arXiv.2410.18540 arXiv:2410.18540 [cs].

[3] James Ang, Gabriella Carini, Yanzhu Chen, Isaac Chuang, Michael Demarco, Sophia Economou, Alec Eickbusch, Andrei Faraon, Kai-Mei Fu, Steven Girvin, Michael Hatridge, Andrew Houck, Paul Hilaire, Kevin Krsulich, Ang Li, Chenxu Liu, Yuan Liu, Margaret Martonosi, David McKay, Jim Misewich, Mark Ritter, Robert Schoelkopf, Samuel Stein, Sara Sussman, Hong Tang, Wei Tang, Teague Tomesh, Norm Tubman, Chen Wang, Nathan Wiebe, Yongxin Yao, Dillon Yost, and Yiyu Zhou. 2024. ARQUIN: Architectures for Multinode Superconducting Quantum Computers. *ACM Transactions on Quantum Computing* 5, 3, Article 19 (Sept. 2024), 59 pages. doi:10.1145/3674151

[4] Anonymous. 2025. Efficient Formal Verification of Quantum Error Correcting Programs | Proceedings of the ACM on Programming Languages. https://dl.acm.org/doi/10.1145/3729293

[5] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[6] Jonathan M. Baker, Andrew Litteken, Casey Duckering, Henry Hoffmann, Hannes Bernien, and Frederic T. Chong. 2021. Exploiting Long-Distance Interactions and Tolerating Atom Loss in Neutral Atom Quantum Architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 818–831. doi:10.1109/ISCA52012.2021.00069

[7] Dominic W. Berry, Graeme Ahokas, Richard Cleve, and Barry C. Sanders. 2007. Efficient quantum algorithms for simulating sparse Hamiltonians. *Communications in Mathematical Physics* 270, 2 (March 2007), 359–371. doi:10.1007/s00220-006-0150-x arXiv:quant-ph/0508139.

[8] Vaughn Betz and Jonathan Rose. 1997. VPR: a new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner (Eds.). Springer, Berlin, Heidelberg, 213–222. doi:10.1007/3-540-63465-7_226

[9] Jérôme Beugnon, Charles Tuchendler, Harold Marion, Alpha Gaëtan, Yevhen Miroshnychenko, Yvan RP Sortais, Andrew M Lance, Matthew PA Jones, Gaetan Messin, Antoine Browaeys, et al. 2007. Two-dimensional transport and transfer of a single atomic qubit in optical tweezers. *Nature Physics* 3, 10 (2007), 696–699.

[10] Damien Bloch, Britton Hofer, Sam R. Cohen, Antoine Browaeys, and Igor Ferrier-Barbut. 2023. Trapping and Imaging Single Dysprosium Atoms in Optical Tweezer Arrays. *Phys. Rev. Lett.* 131 (Nov 2023), 203401. Issue 20. doi:10.1103/PhysRevLett.131.203401

[11] Dolev Bluvstein, Simon J Evered, Alexandra A Geim, Sophie H Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, Madelyn Cain, Marcin Kalinowski, Dominik Hangleiter, et al. 2024. Logical quantum processor based on reconfigurable atom arrays. *Nature* 626, 7997 (2024), 58–65.

[12] Dolev Bluvstein, Simon J. Evered, Alexandra A. Geim, Sophie H. Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, Madelyn Cain, Marcin Kalinowski, Dominik Hangleiter, J. Pablo Bonilla Ataides, Nishad Maskara, Iris Cong, Xun Gao, Pedro Sales Rodriguez, Thomas Karolyshyn, Giulia Semeghini, Michael J. Gullans, Markus Greiner, Vladan Vuletic, and Mikhail D. Lukin. 2024. Logical quantum processor based on reconfigurable atom arrays. *Nature* 626, 7997 (Feb. 2024), 58–65. doi:10.1038/s41586-023-06927-3 arXiv:2312.03982 [quant-ph].

[13] Dolev Bluvstein, Harry Levine, Giulia Semeghini, Tout T Wang, Sepehr Ebadi, Marcin Kalinowski, Alexander Keesling, Nishad Maskara, Hannes Pichler, Markus Greiner, et al. 2022. A quantum processor based on coherent transport of entangled atom arrays. *Nature* 604, 7906 (2022), 451–456.

[14] Dolev Bluvstein, Harry Levine, Giulia Semeghini, Tout T. Wang, Sepehr Ebadi, Marcin Kalinowski, Alexander Keesling, Nishad Maskara, Hannes Pichler, Markus Greiner, Vladan Vuletic, and Mikhail D. Lukin. 2022. A quantum processor based on coherent transport of entangled atom arrays. *Nature* 604, 7906 (April 2022), 451–456. doi:10.1038/s41586-022-04592-6 arXiv:2112.03923 [quant-ph].

[15] H.-J. Briegel, T. Calarco, D. Jaksch, J. I. Cirac, and P. Zoller. 2000. Quantum computing with neutral atoms. *Journal of Modern Optics* 47, 2-3 (2000), 415–451. doi:10.1080/09500340008244052

[16] Che-Ming Chang, Jie-Hong Roland Jiang, Dah-Wei Chiou, Ting Hsu, and Guin-Dar Lin. 2025. Quantum Circuit Compilation for Trapped-Ion Processors With the Drive-Through Architecture. *IEEE Transactions on Quantum Engineering* 6 (2025), 1–14. doi:10.1109/TQE.2025.3548423

[17] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits (Technical Report). doi:10.48550/arXiv.2301.07747 arXiv:2301.07747 [cs].

[18] Bob Coecke and Ross Duncan. 2011. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics* 13, 4 (April 2011), 043016. doi:10.1088/1367-2630/13/4/043016 arXiv:0906.4725 [quant-ph].

[19] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (sep 2022), 50 pages. doi:10.1145/3505636

[20] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 [quant-ph] https://arxiv.org/abs/1707.03429

[21] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. 2019. A Case for Multi-Programming Quantum Computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 291–303. doi:10.1145/3352460.3358287

[22] David Deutsch and Richard Jozsa. 1997. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (Jan. 1997), 553–558. doi:10.1098/rspa.1992.0167 Publisher: Royal Society.

[23] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. 2020. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum* 4 (June 2020), 279. doi:10.22331/q-2020-06-04-279 arXiv:1902.03178 [quant-ph].

[24] Sepehr Ebadi, Tout T Wang, Harry Levine, Alexander Keesling, Giulia Semeghini, Ahmed Omran, Dolev Bluvstein, Rhine Samajdar, Hannes Pichler, Wen Wei Ho, et al. 2021. Quantum phases of matter on a 256-atom programmable quantum simulator. *Nature* 595, 7866 (2021), 227–232.

[25] Simon J Evered, Dolev Bluvstein, Marcin Kalinowski, Sepehr Ebadi, Tom Manovitz, Hengyun Zhou, Sophie H Li, Alexandra A Geim, Tout T Wang, Nishad Maskara, et al. 2023. High-fidelity parallel entangling gates on a neutral-atom quantum computer. *Nature* 622, 7982 (2023), 268–272.

[26] Simon J. Evered, Dolev Bluvstein, Marcin Kalinowski, Sepehr Ebadi, Tom Manovitz, Hengyun Zhou, Sophie H. Li, Alexandra A. Geim, Tout T. Wang, Nishad Maskara, Harry Levine, Giulia Semeghini, Markus Greiner, Vladan Vuletic, and Mikhail D. Lukin. 2023. High-fidelity parallel entangling gates on a neutral atom quantum computer. *Nature* 622, 7982 (Oct. 2023), 268–272. doi:10.1038/s41586-023-06481-y arXiv:2304.05420 [quant-ph].

[27] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '09)*. Association for Computing Machinery, New York, NY, USA, 191–200. doi:10.1145/1508128.1508158

[28] Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia. 2025. QOS: Quantum Operating System. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 429–447. https://www.usenix.org/system/files/osdi25-giortamis.pdf

[29] Google Quantum AI. 2025. Cirq | Google Quantum AI — quantumai.google. https://quantumai.google/cirq. [Accessed 14-04-2025].

[30] TM Graham, Y Song, J Scott, C Poole, L Phuttitarn, K Jooya, P Eichler, X Jiang, A Marra, B Grinkemeyer, et al. 2022. Multi-qubit entanglement and algorithms on a neutral-atom quantum computer. *Nature* 604, 7906 (2022), 457–462.

[31] Koen Groenland, Freek Witteveen, Kareljan Schoutens, and Rene Gerritsma. 2020. Signal processing techniques for efficient compilation of controlled rotations in trapped ions. *New Journal of Physics* 22, 6 (jun 2020), 063006. doi:10.1088/1367-2630/ab8830

[32] M. Hein, W. Dür, J. Eisert, R. Raussendorf, M. Van den Nest, and H.-J. Briegel. 2006. Entanglement in Graph States and its Applications. doi:10.48550/arXiv.quant-ph/0602096 arXiv:quant-ph/0602096.

[33] Fei Hua, Yuwei Jin, Yanhao Chen, Suhas Vittal, Kevin Krsulich, Lev S Bishop, John Lapeyre, Ali Javadi-Abhari, and Eddy Z Zhang. 2023. CaQR: A Compiler-Assisted Approach for Qubit Reuse through Dynamic Circuit. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 59–71.

[34] Chen Huang, Xi Zhao, Hongze Xu, Weifeng Zhuang, Meng-Jun Hu, Dong E. Liu, and Jingbo Wang. 2026. ZAP: Zoned Architecture and Performant Compiler for Field Programmable Atom Array. *IEEE Transactions on Quantum Engineering* 7 (2026), 3103619–3103619. doi:10.1109/TQE.2026.3696707 arXiv:2411.14037 [quant-ph].

[35] H. Häffner, C.F. Roos, and R. Blatt. 2008. Quantum computing with trapped ions. *Physics Reports* 469, 4 (2008), 155–203. doi:10.1016/j.physrep.2008.09.003

[36] IBM Quantum. 2025. transpiler (latest version) | IBM Quantum Documentation — docs.quantum.ibm.com. https://docs.quantum.ibm.com/api/qiskit/transpiler. [Accessed 14-04-2025].

[37] D. Jaksch, J. I. Cirac, P. Zoller, S. L. Rolston, R. Côté, and M. D. Lukin. 2000. Fast Quantum Gates for Neutral Atoms. *Phys. Rev. Lett.* 85 (Sep 2000), 2208–2211. Issue 10. doi:10.1103/PhysRevLett.85.2208

[38] Enhyeok Jang, Youngmin Kim, Hyungseok Kim, Seungwoo Choi, Yipeng Huang, and Won Woo Ro. 2025. Qubit Movement-Optimized Program Generation on Zoned Neutral Atom Processors. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 459–475. doi:10.1145/3696443.3708937

[39] Dominik Janzing, Pawel Wocjan, and Thomas Beth. 2005. "NON-IDENTITY-CHECK" IS QMA-COMPLETE. *International Journal of Quantum Information* 03, 03 (Sept. 2005), 463–473. doi:10.1142/S0219749905001067

[40] Oğuzcan Kırmemiş, Francisco Romão, Emmanouil Giortamis, and Pramod Bhatotia. 2025. Weaver: A Retargetable Compiler Framework for FPQA Quantum Architectures. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery,

New York, NY, USA, 299–316. doi:10.1145/3696443.3708965

[41] Fabian Kreppel, Christian Melzer, Diego Olvera Millán, Janis Wagner, Janine Hilder, Ulrich Poschinger, Ferdinand Schmidt-Kaler, and André Brinkmann. 2023. Quantum Circuit Compiler for a Shuttling-Based Trapped-Ion Quantum Computer. *Quantum* 7 (Nov. 2023), 1176. doi:10.22331/q-2023-11-08-1176

[42] Henning Labuhn, Daniel Barredo, Sylvain Ravets, Sylvain de Léséleuc, Tommaso Macrì, Thierry Lahaye, and Antoine Browaeys. 2016. Realizing quantum Ising models in tunable two-dimensional arrays of single Rydberg atoms. *Nature* 534, 7609 (June 2016), 667–670. doi:10.1038/nature18274 arXiv:1509.04543 [cond-mat].

[43] Neilson Carlos Leite Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. 2025. Testing and Debugging Quantum Programs: The Road to 2030. *ACM Trans. Softw. Eng. Methodol.* 34, 5 (May 2025), 155:1–155:46. doi:10.1145/3715106

[44] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing* 4, 2, Article 10 (feb 2023), 26 pages. doi:10.1145/3550488

[45] Gushu Li, Yufei Ding, and Yuan Xie. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1001–1014. doi:10.1145/3297858.3304023

[46] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 2021. On the Co-Design of Quantum Software and Hardware. In *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication* (Virtual Event, Italy) *(NANOCOM '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. doi:10.1145/3477206.3477464

[47] Sophia Fuhui Lin, Joshua Viszlai, Kaitlin N. Smith, Gokul Subramanian Ravi, Charles Yuan, Frederic T. Chong, and Benjamin J. Brown. 2024. Codesign of quantum error-correcting codes and modular chiplets in the presence of defects. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 216–231. doi:10.1145/3620665.3640362

[48] Wan-Hsuan Lin, Daniel Bochen Tan, and Jason Cong. 2025. Reuse-Aware Compilation for Zoned Quantum Architectures Based on Neutral Atoms. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 127–142. doi:10.1109/HPCA61900.2025.00021

[49] Ji Liu, Peiyi Li, and Huiyang Zhou. 2022. Not All SWAPs Have the Same Cost: A Case for Optimization-Aware Qubit Routing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 709–725. doi:10.1109/HPCA53966.2022.00058

[50] Lei Liu and Xinglei Dou. 2021. QuCloud: A New Qubit Mapping Mechanism for Multi-programming Quantum Computing in Cloud Environment. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 167–178. doi:10.1109/HPCA51647.2021.00024

[51] Lei Liu and Xinglei Dou. 2024. QuCloud+: A Holistic Qubit Mapping Scheme for Single/Multi-programming on 2D/3D NISQ Quantum Computers. *ACM Trans. Archit. Code Optim.* 21, 1 (Jan. 2024), 9:1–9:27. doi:10.1145/3631525

[52] Jason Ludmir and Tirthak Patel. 2024. Parallax: A Compiler for Neutral Atom Quantum Computers under Hardware Constraints. doi:10.48550/arXiv.2409.04578 arXiv:2409.04578 [quant-ph].

[53] Jason Zev Ludmir, Yuqian Huo, Nicholas S. DiBrita, and Tirthak Patel. 2024. Modeling and Simulating Rydberg Atom Quantum Computers for Hardware-Software Co-design with PachinQo. doi:10.48550/arXiv.2412.07181 arXiv:2412.07181 [quant-ph].

[54] Jason Zev Ludmir, Yuqian Huo, Nicholas S. DiBrita, and Tirthak Patel. 2024. Modeling and Simulating Rydberg Atom Quantum Computers for Hardware-Software Co-design with PachinQo. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 39 (Dec. 2024), 25 pages. doi:10.1145/3700421

[55] Hannah J. Manetsch, Gyohei Nomura, Elie Bataille, Kon H. Leung, Xudong Lv, and Manuel Endres. 2024. A tweezer array with 6100 highly coherent atomic qubits. arXiv:2403.12021 [quant-ph] https://arxiv.org/abs/2403.12021

[56] Dmitri Maslov. 2017. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics* 19, 2 (feb 2017), 023035. doi:10.1088/1367-2630/aa5e47

[57] C. Monroe, D. M. Meekhof, B. E. King, and David J. Wineland. 1996. A Schrödinger Cat" Superposition State of an Atom. *NIST* 272 (Jan. 1996), 1131–1136. https://www.nist.gov/publications/schrodinger-cat-superposition-state-atom Last Modified: 2021-10-12T11:10-04:00 Publisher: C Monroe, D M. Meekhof, B E. King, David J. Wineland.

[58] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1015–1029. doi:10.1145/3297858.3304075

[59] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1001–1016. doi:10.1145/3373376.3378477

[60] OpenQASM. [n. d.]. OpenQasm 3.0 Grammar. Retrieved June 26, 2024 from https://openqasm.com/grammar/index.html

[61] Muhammad Osama, Dimitrios Thanos, and Alfons Laarman. 2025. Parallel Equivalence Checking of Stabilizer Quantum Circuits on GPUs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Arie Gurfinkel and Marijn Heule (Eds.). Springer Nature Switzerland, Cham, 109–128. doi:10.1007/978-3-031-90660-2_6

[62] Tirthak Patel, Daniel Silver, and Devesh Tiwari. 2022. Geyser: A Compilation Framework for Quantum Computing with Neutral Atoms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 383–395. doi:10.1145/3470496.3527428

[63] Tirthak Patel, Daniel Silver, and Devesh Tiwari. 2023. GRAPHINE: Enhanced Neutral Atom Quantum Computing using Application-Specific Rydberg Atom Arrangement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 61, 15 pages. doi:10.1145/3581784.3607032

[64] Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence Checking of Quantum Circuits with the ZX-Calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 3 (Sept. 2022), 662–675. doi:10.1109/JETCAS.2022.3202204 arXiv:2208.12820 [quant-ph].

[65] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 4213.

[66] C J Picken, R Legaie, K McDonnell, and J D Pritchard. 2018. Entanglement of neutral-atom qubits with long ground-Rydberg coherence times. *Quantum Science and Technology* 4, 1 (dec 2018), 015011.

doi:10.1088/2058-9565/aaf019

[67] PyZX. 2025. PyZX — PyZX 0.8.0 documentation. https://pyzx.readthedocs.io/en/latest/ Accessed November 8, 2025.

[68] qiskit-transpiler [n. d.]. Qiskit Transpiler. https://qiskit.org/documentation/apidoc/transpiler.html. Accessed: 2022-06-09.

[69] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* 7 (July 2023), 1062. doi:10.22331/q-2023-07-20-1062 arXiv:2204.13719 [quant-ph].

[70] Jixuan Ruan, Xiang Fang, Hezi Zhang, Ang Li, Travis Humble, and Yufei Ding. 2024. PowerMove: Optimizing Compilation for Neutral Atom Quantum Computers with Zoned Architecture. doi:10.48550/arXiv.2411.12263 arXiv:2411.12263 [quant-ph].

[71] M Saffman. 2016. Quantum computing with atomic qubits and Rydberg interactions: progress and challenges. *Journal of Physics B: Atomic, Molecular and Optical Physics* 49, 20 (oct 2016), 202001. doi:10.1088/0953-4075/49/20/202001

[72] Mark Saffman, Thad G Walker, and Klaus Mølmer. 2010. Quantum information with Rydberg atoms. *Reviews of modern physics* 82, 3 (2010), 2313.

[73] Abdullah Ash Saki, Rasit Onur Topaloglu, and Swaroop Ghosh. 2022. Muzzle the Shuttle: Efficient Compilation for Multi-Trap Trapped-Ion Quantum Computers. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 322–327. doi:10.23919/DATE54114.2022.9774619

[74] Tobias Schmale, Bence Temesi, Alakesh Baishya, Nicolas Pulido-Mateo, Ludwig Krinner, Timko Dubielzig, Christian Ospelkaus, Hendrik Weimer, and Daniel Borcherding. 2022. Backend compiler phases for trapped-ion quantum computers. In *2022 IEEE International Conference on Quantum Software (QSW)*. 32–37. doi:10.1109/QSW55613.2022.00020

[75] Ludwig Schmid, David F Locher, Manuel Rispler, Sebastian Blatt, Johannes Zeiher, Markus Müller, and Robert Wille. 2024. Computational capabilities and compiler development for neutral atom quantum processors—connecting tool developers and hardware experts. *Quantum Science and Technology* 9, 3 (2024), 033001.

[76] Ludwig Schmid, David F. Locher, Manuel Rispler, Sebastian Blatt, Johannes Zeiher, Markus Müller, and Robert Wille. 2024. Computational Capabilities and Compiler Development for Neutral Atom Quantum Processors: Connecting Tool Developers and Hardware Experts. *Quantum Science and Technology* 9, 3 (July 2024), 033001. doi:10.1088/2058-9565/ad33ac arXiv:2309.08656 [quant-ph].

[77] Kai-Niklas Schymik, Vincent Lienhard, Daniel Barredo, Pascal Scholl, Hannah Williams, Antoine Browaeys, and Thierry Lahaye. 2020. Enhanced atom-by-atom assembly of arbitrary tweezer arrays. *Phys. Rev. A* 102 (Dec 2020), 063107. Issue 6. doi:10.1103/PhysRevA.102.063107

[78] Cheng Sheng, Jiayi Hou, Xiaodong He, Peng Xu, Kunpeng Wang, Jun Zhuang, Xiao Li, Min Liu, Jin Wang, and Mingsheng Zhan. 2021. Efficient preparation of two-dimensional defect-free atom arrays with near-fewest sorting-atom moves. *Phys. Rev. Res.* 3 (Apr 2021), 023008. Issue 2. doi:10.1103/PhysRevResearch.3.023008

[79] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. 2019. Optimized Compilation of Aggregated Instructions for Realistic Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1031–1044. doi:10.1145/3297858.3304018

[80] Irfan Siddiqi. 2021. Engineering high-coherence superconducting qubits. *Nature Reviews Materials* 6, 10 (2021), 875–891.

[81] Kevin Singh, Shraddha Anand, Andrew Pocklington, Jordan T. Kemp, and Hannes Bernien. 2022. Dual-Element, Two-Dimensional Atom Array with Continuous-Mode Operation. *Phys. Rev. X* 12 (Mar 2022), 011040. Issue 1. doi:10.1103/PhysRevX.12.011040

[82] Yannick Stade, Wan-Hsuan Lin, Jason Cong, and Robert Wille. 2025. Routing-Aware Placement for Zoned Neutral Atom-based Quantum Computing. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD66269.2025.11240721 arXiv:2505.22715 [quant-ph].

[83] Yannick Stade, Ludwig Schmid, Lukas Burgholzer, and Robert Wille. 2024. An Abstract Model and Efficient Routing for Logical Entangling Gates on Zoned Neutral Atom Architectures. doi:10.48550/arXiv.2405.08068 arXiv:2405.08068 [quant-ph].

[84] Yannick Stade, Ludwig Schmid, Lukas Burgholzer, and Robert Wille. 2024. An Abstract Model and Efficient Routing for Logical Entangling Gates on Zoned Neutral Atom Architectures. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 01. 784–795. doi:10.1109/QCE60285.2024.00098

[85] Samuel Stein, Sara Sussman, Teague Tomesh, Charles Guinn, Esin Tureci, Sophia Fuhui Lin, Wei Tang, James Ang, Srivatsan Chakram, Ang Li, Margaret Martonosi, Fred Chong, Andrew A. Houck, Isaac L. Chuang, and Michael Demarco. 2023. HetArch: Heterogeneous Microarchitectures for Superconducting Quantum Systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 539–554. doi:10.1145/3613424.3614300

[86] Samuel Stein, Shifan Xu, Andrew W. Cross, Theodore J. Yoder, Ali Javadi-Abhari, Chenxu Liu, Kun Liu, Zeyuan Zhou, Charlie Guinn, Yufei Ding, Yongshan Ding, and Ang Li. 2025. HetEC: Architectures for Heterogeneous Quantum Error Correction Codes. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 515–528. doi:10.1145/3676641.3716001

[87] Bochen Tan, Dolev Bluvstein, Mikhail D. Lukin, and Jason Cong. 2022. Qubit Mapping for Reconfigurable Atom Arrays. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 107, 9 pages. doi:10.1145/3508352.3549331

[88] Daniel Bochen Tan, Dolev Bluvstein, Mikhail D. Lukin, and Jason Cong. 2024. Compiling Quantum Circuits for Dynamically Field-Programmable Neutral Atoms Array Processors. *Quantum* 8 (March 2024), 1281. doi:10.22331/q-2024-03-14-1281

[89] Daniel Bochen Tan, Wan-Hsuan Lin, and Jason Cong. 2025. *Compilation for Dynamically Field-Programmable Qubit Arrays with Efficient and Provably Near-Optimal Scheduling*. Association for Computing Machinery, New York, NY, USA, 921–929. https://doi.org/10.1145/3658617.3697778

[90] Swamit S. Tannu and Moinuddin Qureshi. 2019. Ensemble of Diverse Mappings: Improving Reliability of Quantum Computers by Orchestrating Dissimilar Mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 253–265. doi:10.1145/3352460.3358257

[91] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 987–999. doi:10.1145/3297858.3304007

[92] Runzhou Tao, Hongzheng Zhu, Jason Nieh, Jianan Yao, and Ronghui Gu. 2025. Quantum Virtual Machines. 411–428. https://www.usenix.org/conference/osdi25/presentation/tao

[93] Hanrui Wang, Pengyu Liu, Daniel Bochen Tan, Yilian Liu, Jiaqi Gu, David Z. Pan, Jason Cong, Umut A. Acar, and Song Han. 2024. Atomique: A Quantum Compiler for Reconfigurable Neutral Atom Arrays. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 293–309. doi:10.1109/ISCA59077.2024.00030

[94] Meng Wang, Chenxu Liu, Samuel Stein, Yufei Ding, Poulami Das, Prashant J. Nair, and Ang Li. 2024. Optimizing FTQC Programs through QEC Transpiler and Architecture Codesign. arXiv:2412.15434 [quant-ph] https://arxiv.org/abs/2412.15434

[95] Shuai Wang, Wenjun Zhang, Tao Zhang, Shuyao Mei, Yuqing Wang, Jiazhong Hu, and Wenlan Chen. 2023. Accelerating the assembly of defect-free atomic arrays with maximum parallelisms. *Physical Review Applied* 19, 5 (May 2023), 054032. doi:10.1103/PhysRevApplied.19.054032 arXiv:2210.10364 [physics].

[96] Zhihui Wang, Stuart Hadfield, Zhang Jiang, and Eleanor G. Rieffel. 2018. Quantum Approximate Optimization Algorithm for MaxCut: A Fermionic View. *Physical Review A* 97, 2 (Feb. 2018), 022304. doi:10.1103/PhysRevA.97.022304 arXiv:1706.02998 [quant-ph].

[97] Karen Wintersperger, Florian Dommert, Thomas Ehmer, Andrey Hoursanov, Johannes Klepsch, Wolfgang Mauerer, Georg Reuber, Thomas Strohm, Ming Yin, and Sebastian Luber. 2023. Neutral Atom Quantum Computing Hardware: Performance and End-User Perspective. *EPJ Quantum Technology* 10, 1 (Dec. 2023), 32. doi:10.1140/epjqt/s40507-023-00190-1 arXiv:2304.14360 [quant-ph].

[98] Jonathan Wurtz, Alexei Bylinskii, Boris Braverman, Jesse Amato-Grill, Sergio H. Cantu, Florian Huber, Alexander Lukin, Fangli Liu, Phillip Weinberg, John Long, Sheng-Tao Wang, Nathan Gemelke, and Alexander Keesling. 2023. Aquila: QuEra's 256-qubit neutral-atom quantum computer. arXiv:2306.11727 [quant-ph] doi:10.48550/arXiv.2306.11727

[99] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 360–374. doi:10.1145/3445814.3446706

[100] Hezi Zhang, Jixuan Ruan, Hassan Shapourian, Ramana Rao Kompella, and Yufei Ding. 2024. OnePerc: A Randomness-aware Compiler for Photonic Quantum Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 738–754. doi:10.1145/3620666.3651372

[101] Hezi Zhang, Anbang Wu, Yuke Wang, Gushu Li, Hassan Shapourian, Alireza Shabani, and Yufei Ding. 2023. OneQ: A Compilation Framework for Photonic One-Way Quantum Computation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 12, 14 pages. doi:10.1145/3579371.3589047

[102] Felix Zilk, Korbinian Staudacher, Tobias Guggemos, Karl Fürlinger, Dieter Kranzlmüller, and Philip Walther. 2022. A compiler for universal photonic quantum computers. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*. 57–67. doi:10.1109/QCS56647.2022.00012

Appendix

In this appendix, we present supplementary materials that provide further details on the theoretical concepts, intermediate representations, algorithms, compiler mapping, and experimental setup discussed throughout the paper.

Contents

- Appendix A: Formal definition of key terms used in the main paper body.
- Appendix B: QASM^{mq} instruction and argument list, pre- and post-conditions and the IR's formal definition.
- Appendix C: Target compiler output description and formal grammar as well as ZAC's integration as a back-end compiler
- Appendix D: Pseudo-codes of the layer-splitting and layout placement algorithms
- Appendix E: Additional evaluation information and results.

A Formal Definitions

The following sections present the formal definitions of relevant key concepts used by MultiQ, including throughput, result fidelity, and functional independence.

Throughput. We simultaneously execute multiple circuit tiles in a *bin*, a group of circuits scheduled for concurrent execution on the same QPU $B_j = \{c_{j1},...,c_{jn}\}$. Each circuit is compiled into a tile c with width $w(c) \in \mathbb{Z}_{>0}$ and execution time $t(c)>0$. The tiles in bin B_j can execute simultaneously if they fit in the total QPU width, $\sum_{c\in B_j} w(c) \le W_{QPU}$. Total QPU width is computed as: $W_{\text{QPU}} = R \cdot W$, where W is the physical width of the hardware space, and $R \in \{1,2\}$ is the number of storage-zone sides. The wall-time of executing the bundled circuit B_j is thus $T(B_j) = t_{\text{init}} + \max_{c\in B_j} t(c)$.

We can then define the throughput as the number of circuits executed per unit time. If we had executed only a single circuit c, our throughput would simply be $\frac{1}{t_{\text{init}} + t(c)}$. However, when scheduling N bins, each with multiple circuit tiles that can co-execute, we define the throughput of that entire set as:

> ▶ **Definition 1** (Throughput). Given N bins B_j (for $1 \le j \le N$), the throughput is calculated as:
>
> $$\tau = \frac{\sum_{j=1}^{N} |B_j|}{\sum_{j=1}^{N} T(B_j)}$$

Fidelity. Quantum fidelity measures the closeness of a noisy quantum state to the desired ideal target state, expressed as a value between 0 and 1.

When the ideal state is $|\psi_{ideal}\rangle$ and the noisy state is $|\psi_{noisy}\rangle$, then the fidelity F is defined as:

$$F(|\psi_{ideal}\rangle, |\psi_{noisy}\rangle) = |\langle\psi_{ideal}|\psi_{noisy}|\psi_{ideal}\rangle|^2$$

In practice, especially for large circuits, computing the ideal state $|\psi_{ideal}\rangle$ is not feasible. Therefore, state-of-the-art compilers often estimate overall circuit fidelity based on the known error rates of individual quantum operations and decoherence [48, 54, 93]. The general approach to estimate fidelity is: For each qubit $i \in [0..N[$, track all applied gates $g_1^{(i)},...,g_n^{(i)}$; each operation has an associated operation fidelity f_{g_k}, and each qubit experiences some decoherence $d^{(i)}(t) = 1 - e^{-t/T2_i}$, where t_i is the idle time of qubit and T_2 is the dephasing time.

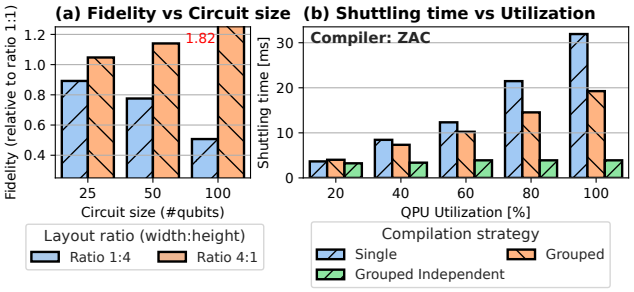


Figure 14. (a) Relative fidelity of two layouts compared to the square layout (ratio 1:1), with increasing circuit size. Narrow layouts (blue bars, 1:4 ratio) achieve lower fidelity than the square ones, while wide layouts (orange bars, 4:1 ratio) achieve higher. **(b)** Total shuttling time with increasing QPU utilization for ZAC [48], executing circuit sequentially (single), circuits merged in parallel (grouped), and concurrently and independently (grouped independent)

> ▶ **Definition 2** (Estimated Total Fidelity). We *estimate* the total fidelity for a circuit with N qubits as follows
>
> $$F_{total} \approx \prod_{i=0}^{N} \left(\prod_{k=0}^{n_i} f_{g_k} \cdot \exp\left(-\frac{t_i}{T_{2_i}} \right) \right),$$
>
> where each qubit $i \in [0..N[$ is applied to n_i gates.

Figure 14 supports the layout-width motivation of §3.1.1: **(a)** narrow (1:4) and wide (4:1) layouts achieve, respectively, lower and higher relative fidelity than a square (1:1) layout as circuit size grows, since a narrower layout lengthens the shuttling distance to the entanglement zone. **(b)** For ZAC, shuttling time grows with QPU utilization regardless of whether circuits execute sequentially (single), as one merged circuit (grouped), or concurrently and independently (grouped independent), confirming that shuttling distance, not the co-execution strategy, is the dominant cost.

Functional Independence. Semantic correctness requires that each circuit $c_k^{\text{single}} \in B$ preserves its functional behavior under the multi-programmed execution E_B. Each isolated original circuit c_k^{single} consists of an ordered sequence of gate-level instructions $G_k = \{g_{k_0}, g_{k_1}, ..., g_{k_{m_k}}\}$ acting on its local qubits Q_k. Its overall unitary transformation is:

$$U_k^{\text{single}} = \prod_{i=m_k}^{0} U(g_{k_i})$$

In a multi-programming environment, a multi-programmed executable E_B is defined as a global sequence of lower-level neutral atom instructions applied to the union of all qubits $Q =$

$\bigcup_k Q_k$ mapped to the co-executing circuits, where $Q_k \cap Q_{k'} = \emptyset$ for all $k \neq k'$. Because these instructions operate at a lower level than quantum gates, they must first be translated back into equivalent quantum gates to reconstruct a global gate-level circuit with gate sequence $G_B = \{g_{B_0}, g_{B_1}, ..., g_{B_M}\}$. The multi-programmed unitary transformation is then defined as $U^{\text{multi}} = \prod_{i=M}^{1} U(g_{B_i})$, from which we derive U_k^{multi}, denoting its restriction to the subset of qubits Q_k.

To guarantee functional independence, we must ensure that the multi-programmed execution is equivalent to the original isolated circuit for each co-executing task, as defined below:

> ▶ **Definition 3** (Functional Independence). The individual circuits in a multi-programmed environment are functionally independent when transformations only observably affect those qubits assigned to circuit c_k.
>
> Functional independence of circuit k holds if there exists a global phase $\phi \in [-\pi, \pi]$ such that the unitary operator U_k^{single} satisfies:
>
> $$U_k^{\text{multi}} \cdot (U_k^{\text{single}})^\dagger = e^{i\phi} I_k$$
>
> where I_k is the identity operation on the Hilbert space of qubits Q_k and $U^\dagger$ is the conjugate transpose (adjoint) of U.

Intuitively, while the unitary transformations are defined over the global qubit state Q, only those transformations relevant for each individual circuit should interact with its assigned qubits.

B QASM^{mq} IR

Here we present the syntax and grammar of the QASM^{mq} intermediate representation used in this work. Table 2 lists the detailed annotations for the QASM^{mq} extensions available. For instance, `@init` sets up atom locations on the SLM trap grid, while `@move` moves one or more atoms. The `@u3` operation performs qubit-ID targeted qubit rotations, which can later be optimized to be performed row-by-row or globally. Finally, `@rydberg` applies a controlled-Z (CZ) gate between qubits within the Rydberg interaction range. Figure 2 summarizes the available instructions, their arguments, and pre- and post-conditions. Figure 15 formalizes the QASM^{mq} grammar in EBNF format.

C Compiler Formal Output

Figure 16a(a) details the formal grammar of the output generated by the target architecture compilation in EBNF format. To illustrate how a compiler can be integrated as a back-end, we use ZAC [48] as a case study.

ZAIR to QASM^{mq} mapping. ZAC's compilation approach processes instructions layer by layer. Each layer begins with a movement phase to bring the required qubits into the entanglement zone, followed by a Rydberg pulse that applies the 2Q gates. A reverse movement phase then returns qubits to their storage zone. Once all layers are finished, its qubits are moved to the measurement zone for readout. This sequence of gates and movements fully defines the standalone execution of a compiled circuit.

```
⟨program⟩ ::= ⟨version⟩ ? ⟨statementOrScope⟩*
⟨version⟩ ::= 'OpenQASM' ⟨versionSpecifier⟩ ';'
⟨statementOrScope⟩ ::= ⟨statement⟩ | ⟨scope⟩
⟨scope⟩ ::= '{' ⟨statementOrScope⟩ * '}'
⟨statement⟩ ::= ⟨pragma⟩
    | ⟨annotation⟩* ( ⟨ioDeclarationStatement⟩ | ⟨gateStatement⟩ |
    ⟨gateCallStatement⟩)
⟨annotation⟩ ::= ⟨initDefinition⟩ | ⟨aodMove⟩ | ⟨u3⟩ | ⟨rydberg⟩

⟨initDefinition⟩ ::= '@init' ⟨qubitPositions⟩
⟨qubitPositions⟩ ::= '[' ⟨position⟩ ( ',' ⟨position⟩ )* ']'
⟨position⟩ ::= '(' ⟨float⟩ ',' ⟨float⟩ ')'
⟨aodMove⟩ ::= '@move' ( 'row' | 'column' ) ⟨integer⟩ ⟨integer⟩
⟨u3⟩ ::= '[' ⟨rotations⟩ ( ',' ⟨rotations⟩ )* ']'
⟨rotations⟩ ::= '(' ⟨float⟩ ⟨float⟩ ⟨float⟩ ')'
⟨rydberg⟩ ::= '@rydberg'
```

Figure 15. Abstract grammar for our QASM^{mq} in EBNF format. Note that the non-terminals highlighted in purple are renamed from the OpenQASM grammar for simplification purposes. Their definitions, the remaining rules, and the full version of the OpenQASM grammar can be found in OpenQASM specifications [19, 20, 60].

Integrating an NA compiler as a target architecture compiler requires mapping its output ZAIR (as formally represented in Figure 16a(b) [48]) to QASM^{mq}. We can directly map most of ZAC's [48] NA instructions to QASM^{mq}. *ZAIR* contains four main instructions: (`init`, `1qGate`, `rydberg`, and `move`), which we map to the corresponding `init`, `u3`, `rydberg`, and `move` instructions in QASM^{mq} by slightly transforming their arguments.

The mapping from ZAIR [48] instructions to QASM^{mq} can be formally defined through the functions described in Equation 1. For reference, Figure 16a includes the formal definition of the ZAIR [48].

$$
\begin{aligned}
&T_{\text{init}} : \text{init}_{\text{ZAIR}}(init_locs) \mapsto \text{init}_{\text{QASM}^{mq}}(init_locs) \\
&T_{\text{1qGate}} : \text{1qGate}_{\text{ZAIR}}(u3, init_locs) \mapsto \\
&\text{u3}_{\text{QASM}^{mq}}(init_locs, [\forall i \in u3 (u3.x, u3.y, u3.z)]) \\
&T_{\text{rydberg}} : \text{rydberg}_{\text{ZAIR}}(zone_id) \mapsto \text{rydberg}_{\text{QASM}^{mq}}(zone_id) \\
&T_{\text{move}} : \text{move}_{\text{ZAIR}}(zone_id, row_id, row_y_begin, row_y_end, \\
&col_id, col_x_begin, col_x_end) \mapsto \\
&\text{move}_{\text{PhysIR}}(row_y_begin, col_x_begin, row_y_end, col_x_end)
\end{aligned}
\tag{1}
$$

Table 2. QASM^{mq} extensions for NA QPUs.

Instruction	Arguments	Description	Pre-condition	Post-condition
`@init`	`[(x,y)`$_{0...n}$`]` `[id`$_0$`,...,id`$_n$`]`	Places and initializes all atoms in the ground state $\|0\rangle$	-	$\forall i \in [0, n]$: `pos(id`$_i$`)=(x`$_i$`,y`$_i$`)`, $\|\Psi_i\rangle \rightarrow \|0\rangle$
`@move`	`[(x,y)`$_{0...n}$`]` `[(x',y')`$_{0...n}$`]`	Shuttle qubits to input coordinates	Non-overlapping movement constraint (see Section 2)	$\forall i \in [0, n]$: `pos(id`$_i$`)=(x`$_i$`,y`$_i$`)` `state_is_preserved(id`$_i$`)`
`@u3`	`[id`$_0$`,...,id`$_n$`]` `[(`θ, ϕ, λ`)`$_{0...n}$`]`	Apply U3 gates to qubits	-	$\forall i \in [0, n]$: $\|\Psi_i\rangle \rightarrow U_3(\theta,\phi,\lambda)_i \|\Psi_i\rangle$
`@rydberg`	-	Apply Rydberg pulse to the entanglement zone	-	CZ is applied to all atoms within blockade radius (see 2)

⟨***output***⟩ ::= { ⟨*tileInfo*⟩ [',' ⟨*tileInfo*⟩] }

⟨***tileInfo***⟩ ::= ⟨*QASM^{mq}* ⟩ ',' ⟨*virtualZoneLayout*⟩

⟨***virtualZoneLayout***⟩ ::= ⟨*qpuVariables*⟩ ',' ⟨*storageVariables*⟩ ',' ⟨*entanglementVariables*⟩

⟨***storageVariables***⟩ ::= {⟨*storageVariable*⟩ [',' ⟨*storageVariable*⟩]}

⟨***entanglementVariables***⟩ ::= {⟨*entanglementVariable*⟩ [',' ⟨*entanglementVariable*⟩]}

⟨***qpuVariables***⟩ ::= ⟨*width*⟩ ',' ⟨*height*⟩ ',' ⟨*nqubits*⟩ ',' ⟨*nAODs*⟩ ',' ⟨*zoneSeparation*⟩

⟨***storageVariable***⟩ ::= ⟨*width*⟩ ',' ⟨*height*⟩ ',' ⟨*position*⟩

⟨***entanglementVariable***⟩ ::= ⟨*width*⟩ ',' ⟨*height*⟩ ',' ⟨*position*⟩

⟨***position***⟩ ::= '(' ⟨*float*⟩ ',' ⟨*float*⟩ ')'

⟨***zoneSeparation***⟩ ::= '(' ⟨*float*⟩ ',' ⟨*float*⟩ ')'

(a) *Target architecture compilation* output

```
<init>::={init_locs:list[(x,y)]}
<1qGate>::={unitary:u3,
            init_locs:list[(x,y)]}
<rydberg>::={zone_id:int}
<move>::={row_id:list[int],
          row_y_begin:list[float],
          row_y_end:list[float],
          col_id:list[int],
          col_x_begin:list[float],
          col_x_end:list[float]}
```

(b) *ZAIR* [48] IR

Figure 16. (a) Formal grammar of the output of the *Target architecture compilation* in EBNF format. The formal definition of the QASM^{mq} grammar is defined in 5.1. **(b)** Instruction definitions of ZAC's IR, named ZAIR [48]

D Algorithms

In this section, we present detailed pseudocode for the core algorithms used in our compilation pipeline, enabling concurrent tile scheduling and execution.

D.1 Layer Splitting Algorithm

Algorithm 1 presents the pseudocode for the layer splitting algorithm, which divides the quantum circuit into parallelizable execution layers. This algorithm iterates over the circuit's dependency graph, dynamically grouping independent operations, such as initialization, local single-qubit gates, and two-qubit Rydberg gates, into cohesive bins to maximize instruction throughput.

Algorithm 1: Split circuit into layers

Data: *circuit*, *window_size* **Result:** L, list of layers of simultaneously executable gates
$L \leftarrow \emptyset$, $D \leftarrow$ layers(circuit) ▷ Convert to DAG layers
pred(n,D): Set of predecessors of node n
layer_compatible(n,l): (n matches gate flag (S or M)) $\wedge$ (all pred(n) have been executed)
while $|D| > 0$ **do**
  $l \leftarrow \emptyset$, $W \leftarrow$ window(D,k,w_size) ▷ Fetch window of gates to consider for the current layer
  foreach *layer* $\in W$ *(sorted by gate size)* **do**
    foreach $n \in$ *layer* **do**
      if $|l| = 0 \vee$ *layer_compatible*(n,l) **then**
        Move $\{n\}$ from $D[k+p]$ to l ▷ Remove n from D and add it to the current layer
        Update S and M depending on gate size
        continue
      end
    end
    $L \leftarrow L \cup \{l\}$ ▷ Add current layer l to the execution layers L
    if $D[k]$ *is empty* **then** $D \leftarrow D \setminus \{D[k]\}$ ▷ Remove empty from D ;
  end
  return L
end

D.2 Placement Algorithm

The placement algorithm (Algorithm 2) based on simulated annealing maps virtual layouts into bins and optimizes their positions in the hardware space. This stochastic optimization approach iteratively refines the positions of the virtual layouts within the available resource space.

Algorithm 2: SA for collision-aware tile placement

Data: Tiles τ with priorities p_i, grid size $R \times W$, weights α, β, cost function C
Result: Tile placement $\{(x_i, y_i, z_i)\}$ minimizing total cost
Initialize grid G with a greedy placement of tiles sorted by p_i / w_i
Initialize temperature $T \leftarrow T_0$, initial placement $P \leftarrow G$
Compute objective $\mathcal{E}(P) \leftarrow \alpha \cdot C(P) - \beta \cdot \sum_i p_i z_i$
for $k = 1$ *to max iterations* **do**
  Generate move $P' \leftarrow \text{Perturb}(P)$ ▷ Swap or move to empty space
  if P' *is feasible (in-bounds, not overlapping)* **then**
    Compute $\mathcal{E}(P') \leftarrow \alpha \cdot C(P') - \beta \cdot \sum_i p_i z_i$
    $\Delta \leftarrow \mathcal{E}(P') - \mathcal{E}(P)$ ▷ Compute new objective to compare
    if $\Delta < 0$ ***or*** $\exp(-\Delta / T) > rand()$ **then**
      $P \leftarrow P'$ ▷ Accept move if new objective is lower
    end
  end
  $T \leftarrow \gamma \cdot T$ ▷ Reduce temperature
end
return *Final placement* P

E Benchmarks and Experimental setup

Table 3 lists the full set of quantum algorithms used as benchmarks. Additionally, Figure 8 illustrates a detailed end-to-end total runtime comparison between MultiQ and other state-of-the-art compilers, such as ZAC and PachinQo.

E.1 Additional Evaluation

To complement the evaluation plots in the main text, we include additional plots capturing MultiQ's performance.

Figure 17 displays the fidelity and execution times for all evaluated sets, ranging from 4 to 14 circuits. Table 4 summarizes the resulting values of this evaluation. Similar to the findings discussed earlier, we observe a very slight decrease or maintained fidelity when comparing isolated ZAC-compiled circuits to MultiQ's concurrent execution results. As the number of multiprogrammed circuits increases. This fidelity drop is directly correlated with an increase in total execution time, suggesting that the difference is predominantly due to higher decoherence errors during the extended waiting periods.

However, Figure 17(a) also shows that MultiQ can achieve a slight improvement in fidelity compared to ZAC in certain configurations. This improvement is likely due to the row-wise single-qubit gate optimization implemented by MultiQ, which effectively reduces the time required for single-gate operations and consequently lowers overall decoherence errors.

Furthermore, total runtimes scaling from 4 to 12 circuits demonstrate that MultiQ improves total runtime by a significant margin compared to ZAC, and even more so compared to PachinQo. The runtime reduction over ZAC is primarily driven by our strong optimization of the initialization phase. Conversely, PachinQo exhibits a significantly longer execution time per circuit, exacerbating its total runtime overhead as the number of circuits scales up.

Table 3. List of benchmarks used for evaluating MultiQ. These benchmarks were sourced from the QASMBench [44] open-source benchmark suite.

Algorithm	# of Qubits
BV (Bernstein-Vazirani) [1]	14, 19
CAT (Schrödinger Cat Superposition) [57]	22
QAOA (MaxCut) [96]	14
DJ (Deutsch-Jozsa) [22]	16, 26
HamSim (Hamiltonian Simulation) [7]	18
Graph State [32]	20
GHZ (Greenberger-Horne-Zeilinger)	23
KNN (Quantum k-nearest Neighbors)	25
SWP (Swap Test)	25
WST (W-state)	24, 27
Multiply	13

Table 4. RQ #1: Fidelity and circuit execution time means for different multi-programming sets of circuits.

	Compiler	4	6	8	10	12	14
Fidelity (%)	ZAC	65.26	67.27	69.87	69.74	64.56	65.87
	MultiQ (1 Row)	64.51	64.46	66.63	65.74	57.00	55.07
	MultiQ (2 Row)	64.62	64.56	66.91	65.91	59.92	60.58
	PowerMove	60.00	61.17	65.92	65.44	58.02	60.19
	QMAP	59.37	62.20	65.31	65.75	60.59	61.76
	ZAP	64.85	67.43	69.87	69.71	64.37	65.68
	PachinQo	34.93	40.69	49.07	45.93	37.10	41.45
Time (ms)	ZAC	9.13	8.66	7.43	7.65	8.91	8.45
	MultiQ (1 Row)	7.82	8.90	8.93	10.08	16.07	19.01
	MultiQ (2 Row)	7.71	8.80	8.61	9.87	12.13	12.41
	PowerMove	22.50	19.59	18.39	18.87	24.12	22.06
	QMAP	17.01	14.56	13.64	13.36	15.32	14.68
	ZAP	10.61	9.14	8.32	8.51	10.21	9.66
	PachinQo	56.33	31.94	35.64	37.60	50.21	43.43

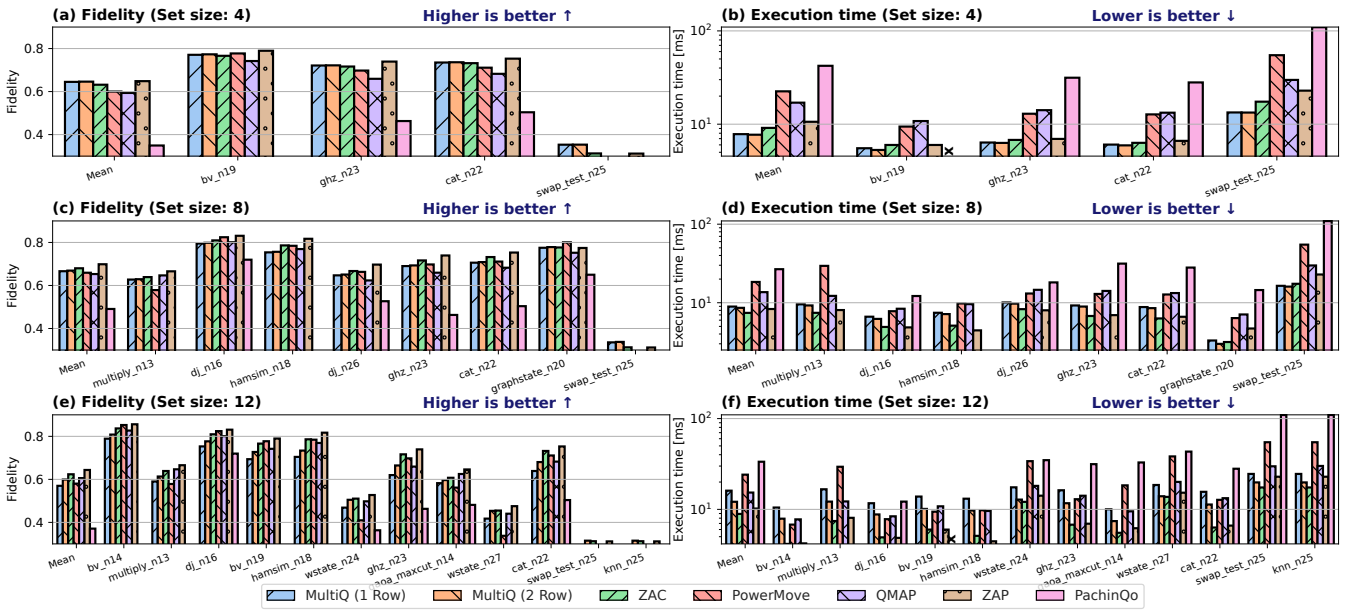


Figure 17. Full detailed end-to-end fidelity and execution evaluation.