ConiQ: Enabling Concatenated Quantum Error Correction on Neutral Atom Arrays

Pengyu Liu
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
pengyuliu@cmu.edu

Mingkuan Xu
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
mingkuan@cmu.edu

Hengyun Zhou
QuEra Computing Inc
Boston, MA
hyharryzhou@gmail.com

Hanrui Wang
Computer Science Department
University of California, Los Angeles
Los Angeles, CA
wang@cs.ucla.edu

Umut A. Acar
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
umut@cmu.edu

Yunong Shi
Amazon Web Services
New York, NY
shiyunon@amazon.com

***Abstract*—Recent progress on concatenated codes, especially many-hypercube codes, achieves unprecedented space efficiency. Yet two critical challenges persist in practice. First, these codes lack efficient implementations of addressable logical gates. Second, the required high degree of parallelism and long-range interactions pose significant challenges for current hardware platforms. In this paper, we propose an efficient compilation approach for concatenated codes, specifically many-hypercube codes, targeted at neutral atom arrays, which provide the necessary parallelism and long-range interactions. Our approach builds on two key innovations. First, we introduce Automorphism-assisted Hierarchical Addressing (AHA) logical `CNOT` gates that significantly reduce spacetime overhead compared to conventional distillation-based methods. Second, we develop Virtual Atom Intermediate Representation (VAIR) that enables level-wise optimization and legalization. We implement these innovations in ConiQ, a hardware-aware quantum compiler designed to compile fault-tolerant quantum circuits for neutral atom arrays using many-hypercube codes. Our evaluation demonstrates that ConiQ achieves up to $2000\times$ reduction in spacetime overhead and up to $10^6\times$ reduction in compilation time compared to state-of-the-art compilers, with our AHA gates providing an additional overhead reduction of up to $20\times$. These results establish concatenated codes as a promising approach for fault-tolerant quantum computing in the near future.**

***Index Terms*—quantum error correction, neutral atom arrays, concatenated codes, fault-tolerant quantum computing, quantum compilation.**

I. Introduction

Quantum computing inherently suffers from noise, making quantum error correction (QEC) indispensable for practical quantum computation. Surface codes have long been the leading candidate due to their high error threshold, but their low encoding rate imposes substantial qubit overhead. For instance, achieving a 10^{-9} logical error rate at a physical error rate of 10^{-3} requires a surface code of distance 15, incurring $225\times$ space overhead [12]. Quantum Low-Density Parity Check (qLDPC) codes [26], [9], [18], despite promising asymptotic properties, face significant near-term limitations, including poor hardware adaptability and insufficient understanding of practical logical gate implementations [16].

Recently, concatenated QEC codes have emerged as compelling alternatives due to their impressive encoding rates. For example, many-hypercube codes [14] represent a family of QEC codes constructed by recursively concatenating quantum error-detecting codes with minimal sizes and high qubit efficiency, such as the [[6,4,2]] codes (encoding 4 logical qubits into 6 physical qubits with distance 2). This approach enables a substantial reduction in physical qubit overhead. Four levels of concatenation yield an encoding rate of $(4/6)^4 \approx 20\%$, allowing many-hypercube codes to encode 1 logical qubit using approximately 5 physical qubits on average at a code distance of 16, significantly outperforming both surface codes and qLDPC codes. However, two critical obstacles hinder the practical deployment of concatenated codes. First, current proposals rely on distillation for individually addressable logical gates, particularly `CNOT` gates, which incur considerable spacetime overhead [14]. Second, the requirements for long-range interaction and high parallelism pose substantial challenges for current hardware and compilers. These limitations have thus far restricted concatenated codes to theoretical study, impeding their application in real-world quantum systems.

The second challenge can potentially be addressed by leveraging neutral atom arrays as the hardware platform. Neutral atom arrays offer numerous advantages for fault-tolerant quantum computing: recent experiments demonstrate their long coherence times, impressive scalability, and low-error operations [24], [6], [17], [10]. Most importantly, the unique capability of neutral atom arrays to dynamically reposition qubits during computation provides long-range interactions and extensive parallelism—properties particularly well-suited for concatenated codes. However, the long-range interactions and parallelism are in a restricted form, and require special compilation techniques to exploit the capabilities provided by neutral atom arrays. Although compilers for neutral atom

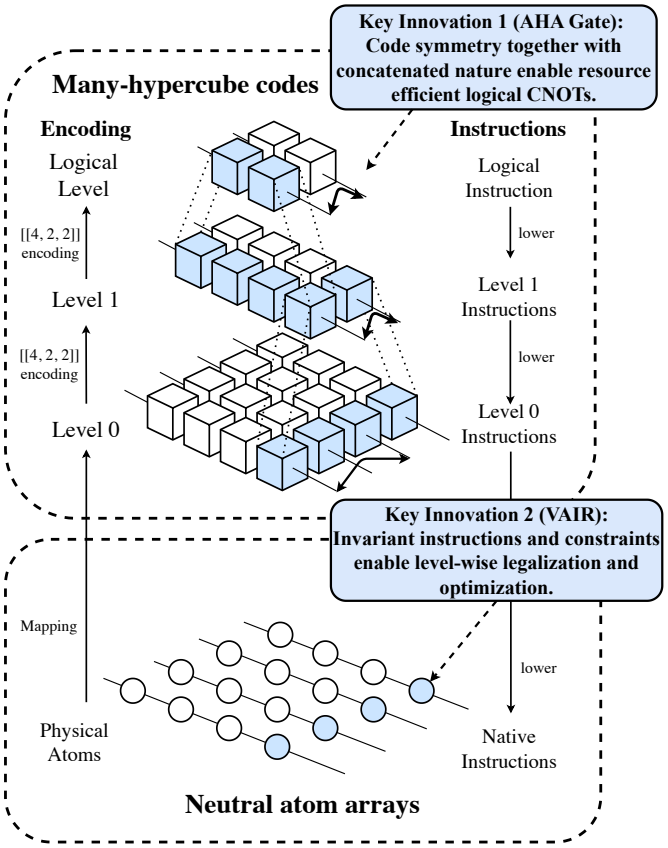

Fig. 1: Overview of ConiQ and its key innovations: (1) The Automorphism-assisted Hierarchical Addressing (AHA) scheme for efficient individually addressable logical gates. (2) The Virtual Atom Intermediate Representation (VAIR) for efficient optimization and legalization across concatenation levels.

arrays such as Atomique [31] and Enola [30] exist, they focus on the compilation of general quantum circuits, and result in prohibitive overheads in the context of concatenated codes.

To harness the advantages of concatenated QEC codes and neutral atom arrays, we introduce ConiQ, a novel compiler framework specifically designed for efficiently implementing concatenated QEC codes on neutral atom arrays. ConiQ builds upon two key innovations with an overview shown in Fig. 1:

(1) By leveraging code symmetry and concatenation, we develop an Automorphism-assisted Hierarchical Addressing (AHA) logical `CNOT` gate scheme that is individually addressable, requiring only a few error correction cycles of overhead, dramatically improving efficiency compared to prior distillation-based schemes.

(2) We introduce a Virtual Atom Intermediate Representation (VAIR) that virtualizes logical registers as physical atoms across concatenation levels. Crucially, we prove that VAIR preserves consistent program states, instruction sets, and hardware constraints at each concatenation level, enabling efficient level-wise legalization and optimization.

Compared to the state-of-the-art compilers Atomique [31] and Enola [30] with prior distillation-based `CNOT` gates,

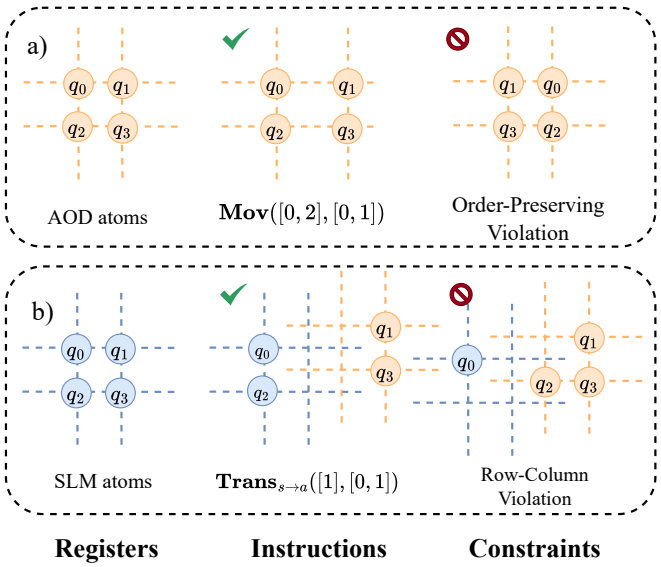

Fig. 2: Neutral atom array architecture, instructions, and constraints: (a) An allowed move operation (left) and a forbidden move operation that violates the order-preserving constraint (right). (b) An allowed transfer operation affecting entire rows and columns (left) and a forbidden operation that violates the row-column constraint (right).

ConiQ achieves over $10^4\times$ improvement in spacetime overhead and $10^6\times$ reduction in compilation time. Thus, ConiQ realizes the potential of high-encoding-rate quantum error correction with dramatically reduced overhead, enabling practical deployment of concatenated QEC codes on near-term neutral atom quantum computers.

II. Background

A. Neutral Atom Arrays

Neutral atom arrays possess the unique capability to move qubits during computation, providing effective long-range connectivity and massive parallelism. Here we formally describe the neutral atom array architecture, program states, instruction set, and constraints at the physical level. In Section VI, we extend these concepts to the logical level and design the Virtual Atom Intermediate Representation (VAIR) for efficiently compiling concatenated codes.

Architecture. Neutral atom arrays utilize individually-focused laser spots to trap and manipulate atoms. Two types of traps are typically employed: fixed spatial light modulators (SLM) and movable acousto-optic deflectors (AOD). Different lasers are typically used for SLM and AOD traps, so different type of traps won't interfere with each other. Each type of trap forms a two-dimensional square grid. Atoms can reside in either type of laser trap. AOD atoms can be dynamically repositioned by adjusting the corresponding lasers. Conceptually, this configuration creates two planar atom layers: a stationary SLM layer and a movable AOD layer, slightly offset along the z axis.

Program State Representation. We represent the collective system state, particularly atom positions, as a four-tuple (I, J, A, S):

- I: Ordered x-coordinates of AOD traps. $I[i]$ denotes the x coordinate of the i-th column, where $I[i] < I[i']$ for $i < i'$.
- J: Ordered y-coordinates of AOD traps. $J[j]$ denotes the y coordinate of the j-th row, where $J[j] < J[j']$ for $j < j'$.
- A: Positions of AOD atoms, where $A[i][j]$ identifies the atom at the i-th column and j-th row, located at $(I[i], J[j])$ or is empty $(\emptyset)$ when no atom is present.
- S: Positions of SLM atoms on an integer grid, where $S[i][j]$ identifies the atom index at integer coordinates (i, j) (assuming SLM atoms are placed in a unit grid) or is empty $(\emptyset)$ when no atom is present.

Instruction Set. Neutral atom arrays support quantum gate operations as well as parallel atom transfer and movement through the following instruction set. Here, we use I' and J' to represent subsets of I and J, respectively:

- $\mathbf{Trans}_{\mathrm{a}\to\mathrm{s}}(I', J')$ and $\mathbf{Trans}_{\mathrm{s}\to\mathrm{a}}(I', J')$: Transfer atoms between SLM and AOD traps at intersections of rows I' and columns J'. $a \to s$ indicates transfer from AOD to SLM and vice versa. For $\mathbf{Trans}_{\mathrm{s}\to\mathrm{a}}(I', J')$, if an SLM atom exists at $S[i'][j'] = n$ where $i' \in I', j' \in J'$, and there is an empty AOD trap $A[i][j] = \emptyset$ where $I[i] = i'$ and $J[j] = j'$ (the SLM atom and the empty AOD trap overlap in the x-y plane), then the atom is transferred by setting $A[i][j] = n$ and $S[i'][j'] = \emptyset$. $\mathbf{Trans}_{\mathrm{a}\to\mathrm{s}}(I', J')$ performs the reverse transfer.
- $\mathbf{Mov}(I, J)$: Relocates AOD atoms to new positions, updating the state to (I, J, A, S). The movement must satisfy the order-preserving constraint: I and J remain ordered.
- $\mathbf{1Qgate}(U, I', J')$: Applies single-qubit operations U to SLM atoms at specified grid coordinates. For all $i' \in I', j' \in J'$, if there is an SLM atom $S[i'][j'] = n$, then U is applied to qubit n.
- $\mathbf{ParallelCZ}(I', J')$: For all SLM atoms $S[i'][j']$, where $i' \in I', j' \in J'$, if there is an AOD atom overlapping with it, a `CZ` gate is applied between the SLM atom and the AOD atom.[1]

Program Constraints. The instruction set for neutral atom arrays offers significant parallelism but with specific constraints (illustrated in Fig. 2):

- *Order-Preserving Constraint*: Move instructions must maintain the relative order of AOD atoms to prevent collisions (Fig. 2 (a)).
- *Row-Column Constraint*: Instructions are executed on all intersections of entire rows and columns (Fig. 2 (b)).

While available instructions provide parallelism that could potentially be advantageous for concatenated codes, the constraints also pose significant challenges for efficient circuit compilation: unlike in superconducting qubits, where logically independent physical operations acting on neighboring qubits can always be scheduled in parallel.

[1]In current neutral atom arrays, `CZ` gates are implemented globally. The $\mathbf{ParallelCZ}(I', J')$ operation can be realized using a global `CZ` gate with two move operations [31]. While one-qubit gates can also be applied to AOD atoms, and two-qubit gates between SLM-SLM or AOD-AOD pairs are possible, we omit these capabilities in this paper for simplicity.

B. QEC Basics and Many-Hypercube Codes

We will focus on many-hypercube codes in this paper, which are the leading candidate in concatenated QEC codes, but our approach, especially the VAIR compilation, can be applied to other concatenated QEC codes.

Error Detecting Codes. Error detecting codes are QEC codes capable of detecting errors but not correcting them. The simplest family of error-detecting codes is the $[[2m, 2m-2, 2]]$ code [2], which encodes $2m-2$ logical qubits into $2m$ physical qubits with code distance 2. We denote this family as D_{2m}. The D_{2m} code has two stabilizers: $\mathtt{X}_1 \cdots \mathtt{X}_{2m}$ and $\mathtt{Z}_1 \cdots \mathtt{Z}_{2m}$. As an example, Table I presents the logical operators of D_4, the smallest instance of the D_{2m} family.

Automorphism of codes. Automorphisms of codes are symmetries that can be exploited to construct efficient fault-tolerant gadgets through physical qubit permutation. Take the D_4 code as an example: by swapping the first and third physical qubits, we exchange the logical operators of the two logical qubits [22], which creates a logical `SWAP` gate. This automorphism property will be leveraged in our `CNOT` gate implementation.

Logical Op.	q_1	q_2	q_3	q_4
$\mathtt{X}_{L_1}$	X	X		
$\mathtt{X}_{L_2}$		X	X	
$\mathtt{Z}_{L_1}$		Z	Z	
$\mathtt{Z}_{L_2}$	Z	Z		

TABLE I: Logical operators of the D_4 error detecting code. Each row shows the combination of physical operators that implement a logical operation. For example, applying `X` gates to physical qubits 1 and 2 implements an `X` operation on the first logical qubit.

Many-hypercube Codes. The D_{2m} code alone cannot correct errors and is therefore unsuitable for fault-tolerant quantum computation. Concatenation offers a powerful technique for constructing codes with enhanced error correction capabilities from simpler codes. The concatenated D_6 codes are known as many-hypercube codes [14]. Concatenation encodes physical qubits using an initial code, then treats the resulting logical qubits as physical inputs for the next encoding level. This process can be repeated to enhance error correction capabilities. We use *level* to denote each concatenation iteration, and *register* to refer to a block of logical qubits treated collectively at a particular level.

In this paper, we extend the definition of many-hypercube codes to include both D_6 and D_4 codes and denote an l-level many-hypercube code as $D_{n_1,n_2,\ldots,n_l}$, indicating that level-i uses a D_{n_i} code. For example, the code $D_{4,4,6,6}$ employs two levels of D_4 (near the physical level), followed by two levels of D_6 encoding (near the logical level). We use $q_{i_1\cdots i_l}$

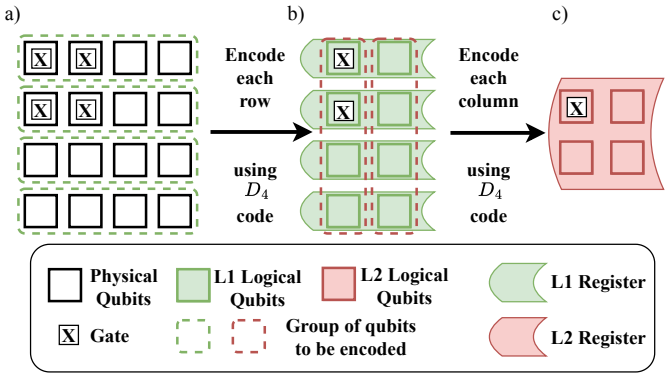

Fig. 3: Construction of a level-2 $D_{4,4}$ many-hypercube code: (a) Initial arrangement of physical qubits. (b) First level of concatenation. (c) Second level of concatenation.

to denote the i_l-th logical qubit in the code consisting of the level-$(l-1)$ logical qubits $q_{i_1 \cdots i_{l-1}}$.

Fig. 3 illustrates a $D_{4,4}$ code with 16 physical qubits. First, groups of four qubits are encoded using the D_4 code, creating level-1 registers with 2 logical qubits each. Then the two columns of level-1 logical qubits are encoded using the D_4 code separately, resulting in a level-2 register with four logical qubits. This demonstrates why we refer to the four logical qubits as a register: the information of these logical qubits is interleaved across the physical qubits and must be processed collectively. In this example, logical Pauli gates can be applied based on Table I. As shown in the figure, to apply a logical `X` gate to the first logical qubit in a level-2 register, we apply two level-1 `X` gates, which are further decomposed into four physical `X` gates.

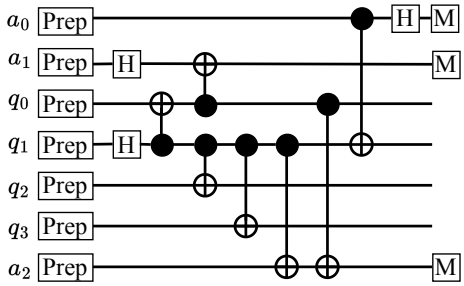

Fig. 4: State preparation gadget for a level-2 register in $D_{4,4}$ code.

Fault-tolerant Gadgets of Many-hypercube Codes. Fault-tolerant gadgets represent atomic instructions at the logical level that guarantee the desired fault-tolerant properties [15]. We focus on two types of gadgets: state preparation and logical instructions.

State Preparation. State preparation constitutes the first step of a fault-tolerant circuit and is essential for other logical instructions. [14] proposed a state preparation gadget that offers an effective trade-off between logical error rate and overhead. Taking the level-2 state preparation protocol as an example, as illustrated in Fig. 4, the inputs q_0 to q_3 and a_0 to a_2 are all level-1 registers. These registers first undergo a level-1 encoding circuit, followed by a series of transversal gates. Conditioned on the measurement results of the ancilla registers, we can prepare the level-2 encoded state fault-tolerantly. From this example, we observe two critical properties of many-hypercube codes gadgets: (1) Higher-level gadgets often require identical copies of lower-level gadgets that can, in principle, be batched—a property that should be exploited during compilation. (2) Numerous long-range interactions are required, which may be expensive to implement.

Logical Instructions. Since each code block in the D_{2m} family encodes multiple logical qubits in one register, the ability to individually address specific logical qubits within a block is crucial for computation. The conventional approach for implementing addressable logical `CNOT` gates relies on plus state distillation [14], which requires exponentially increasing resources. However, the D_{2m} code, being a CSS code, permits simpler implementations of certain gates. Logical `CNOT` gates between all corresponding logical qubits of two registers can be implemented transversally by applying physical `CNOT` gates between all corresponding physical-qubit pairs [7]. Additionally, the D_{2m} codes feature transversal `H` gates (up to a permutation of the logical qubits).

III. Challenges in Practical Implementations of Concatenated Codes on Neutral Atom Arrays

Although concatenated QEC codes offer excellent space efficiency as quantum memory, two key challenges hinder their practical deployment: (1) Existing logical gates, particularly individually addressable gates, incur significant spacetime overhead for computation. (2) The absence of efficient compilation methods prevents effective utilization of the intrinsic parallelism in neutral atom arrays. We will examine these challenges in detail in the following sections.

A. Efficient Addressable Logical Gate Implementation

The current distillation-based approach for implementing individually addressable logical gates introduces significant overhead, primarily because state distillation must be repeated at each level of concatenation [14], [35]. At every level, high-fidelity resource states are generated by consuming multiple noisy copies from the lower level. This recursive structure compounds the cost exponentially, causing overall resource requirements to grow rapidly with concatenation levels. Consequently, it is crucial to develop low-cost logical gate schemes for practical near-term implementation.

B. Effectively Leveraging Neutral Atom Arrays' Intrinsic Parallelism

Compiling concatenated QEC codes typically involves two tasks: register mapping and gadget scheduling. One approach is to fully unfold all registers and gadgets to the physical level before performing mapping and scheduling. Current compilers, such as Atomique [31] and Enola [30], follow this method. However, this approach neglects the hierarchical structure inherent in concatenated QEC codes, resulting in significant spacetime and compilation overhead.

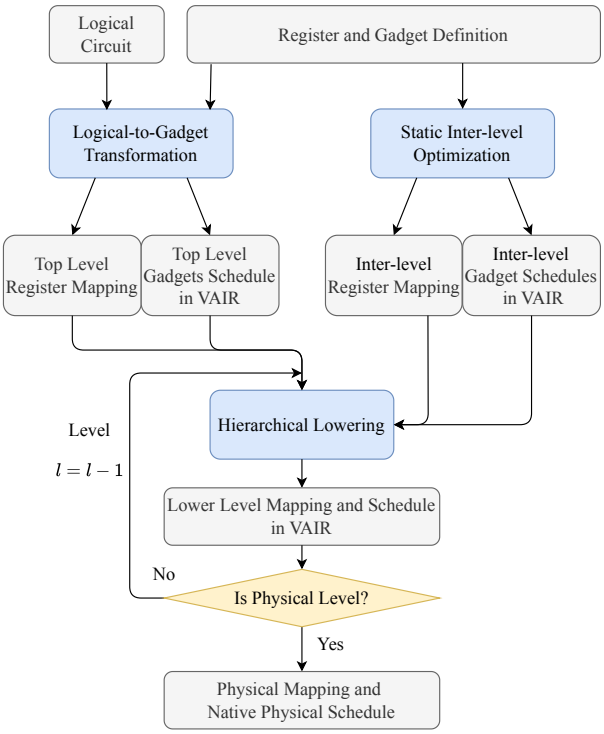

Fig. 5: Three-phase compilation workflow of ConiQ.

A more effective compilation strategy maps registers and schedules gadgets level-by-level, closely aligning with the hierarchical structure of concatenated QEC codes. However, current compilation methods lack a program representation that efficiently exposes physical constraints of neutral atom arrays to higher logical levels, making optimization (batching parallelizable gadgets) and legalization (ensuring physical-level feasibility) nearly impossible.

Without properly exposing these physical constraints, only two naive strategies remain: (1) prioritizing legalization by sequentially scheduling all gadgets, thus severely underutilizing hardware parallelism, or (2) prioritizing optimization by batching all logically independent gadgets, potentially generating illegal schedules that must be resolved at the physical level.

Both strategies lead to catastrophic consequences—ignoring legalization produces unsound and unusable compilation outcomes, whereas sacrificing parallelism causes exponential runtime growth due to multiplicative inefficiencies compounding across concatenation levels. We call this exponential inefficiency *cascading latency amplification*.

Therefore, it is crucial to develop a compilation method capable of level-by-level compilation that simultaneously achieves legalization and optimization by clearly exposing allowed instructions and physical constraints at every concatenation level.

IV. ConiQ Architecture

ConiQ is specifically designed to address the challenges outlined above through two key innovations:

1) **Automorphism-assisted Hierarchical Addressing (AHA) Gates**: By leveraging the symmetry and concatenated structure of codes, ConiQ introduces an Automorphism-assisted Hierarchical Addressing logical gate scheme, significantly reducing overhead compared to distillation-based methods.
2) **Virtual Atom Intermediate Representation (VAIR)**: At the core of ConiQ is VAIR, a hierarchical intermediate representation explicitly designed to expose allowed instructions and physical constraints of neutral atom arrays to each concatenation level. We prove that scheduling instructions compliant with VAIR constraints at higher levels directly ensures both legality and parallelism at the physical level.

Built upon these key innovations, ConiQ employs a structured, level-wise compilation workflow, illustrated in Fig. 5. The workflow comprises three phases:

(1) Logical-to-Gadget Transformation: This phase converts the input logical circuit into a sequence of top-level fault-tolerant gadgets. We leverage our AHA to implement addressable logical gates with minimized gadget cost while ensuring fault tolerance requirements [15].

(2) Static Inter-level Optimization: Starting with gadget definitions for each concatenation level, this phase generates templates of optimized register mappings and schedules using the VAIR framework. VAIR enables independent optimization and legalization of each level, ensuring that physical runtime decreases proportionally with each level's runtime reduction.

(3) Hierarchical Lowering: Using the optimized templates produced in the previous phase, hierarchical lowering recursively translates instructions from higher levels down to physical instructions. VAIR guarantees that no additional overhead is introduced when compiling to lower levels and that all physical instructions are native to the neutral atom arrays.

This structured, three-phase compilation not only exploits the hierarchical structure of concatenated codes, but also reuses the templates produced in the static optimization phase, significantly reducing compilation overhead compared to existing compilers, effectively unlocking the full potential of concatenated QEC codes on neutral atom array hardware.

V. Automorphism-assisted Hierarchical Addressing Gate Scheme

In this section, we introduce the Automorphism-assisted Hierarchical Addressing (AHA) scheme, which enables individually addressable logical instructions with significantly reduced overhead. For clarity, we illustrate the scheme using the D_4 code, though the methods readily extend to the D_6 code.

We present this section in two parts: first summarizing the fundamental instructions required to implement the AHA scheme, then detailing the construction process of individually addressable gates.

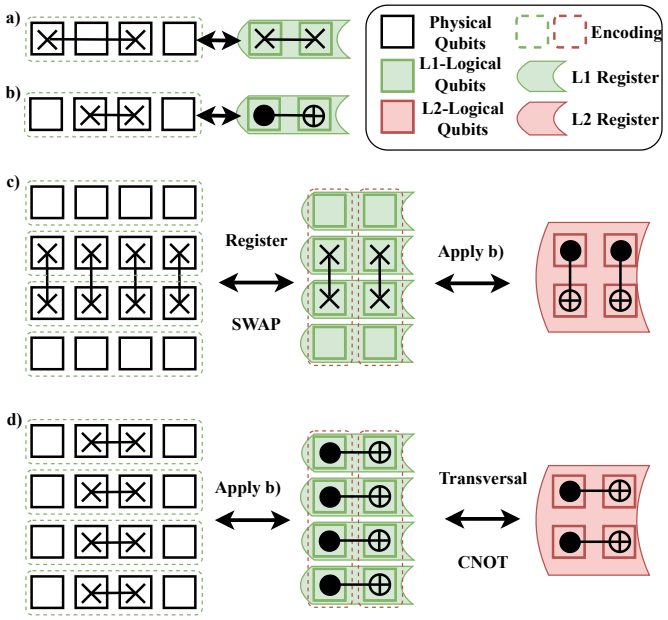

Fig. 6: Implementation of logical operations through physical qubit permutations: (a) A logical SWAP gate implemented by swapping the first and third physical qubits. (b) A logical CNOT gate implemented by swapping the second and third physical qubits. (c) Intra-register batched CNOT gates achieved by swapping the second and third rows of physical qubits in a 4×4 grid. (d) Inter-register batched CNOT gates along another dimension, implemented by swapping the second and third columns of physical qubits.

Fundamental Instructions for AHA. The AHA scheme requires the following essential logical instructions:

1) State preparation and error correction.
2) Transversal CNOT gates.
3) Intra-register batched SWAP operations along a dimension: $\prod \texttt{SWAP}(q_{i_1,\ldots,0,\ldots,i_l}, q_{i_1,\ldots,1,\ldots,i_l})$.
4) Intra-register batched CNOT operations along a dimension: $\prod \texttt{CNOT}(q_{i_1,\ldots,0,\ldots,i_l}, q_{i_1,\ldots,1,\ldots,i_l})$.

The original many-hypercube codes construction [14] directly provides instructions 1–3. Instruction 4, critical for the AHA scheme, is newly introduced in this work and detailed below.

Intra-register Instructions via Automorphisms. As illustrated in Fig. 6 (a, b) and Section II-B, the automorphisms of the D_4 code enable intra-register logical SWAP and logical CNOT through strategically selected physical SWAP operations [22]. At higher concatenation levels, we generalize these patterns by combining lower-level intra-register SWAP and CNOT operations, as depicted in Fig. 6 (c, d). By carefully selecting physical qubit pairs for SWAP, we can implement *intra-register batched* logical CNOT gates along any desired dimension, naturally scaling to arbitrary concatenation levels.

Addressable Logical CNOT via Hierarchical Addressing. With the fundamental intra-register operations established, we next describe the construction of individually addressable logical CNOT gates using hierarchical addressing. Although intra-register batched gates alone cannot address individual logical qubits, they effectively separate logical qubits into two distinct halves. To achieve full addressability, we iteratively refine this division in a binary-search-like process termed *hierarchical addressing*.

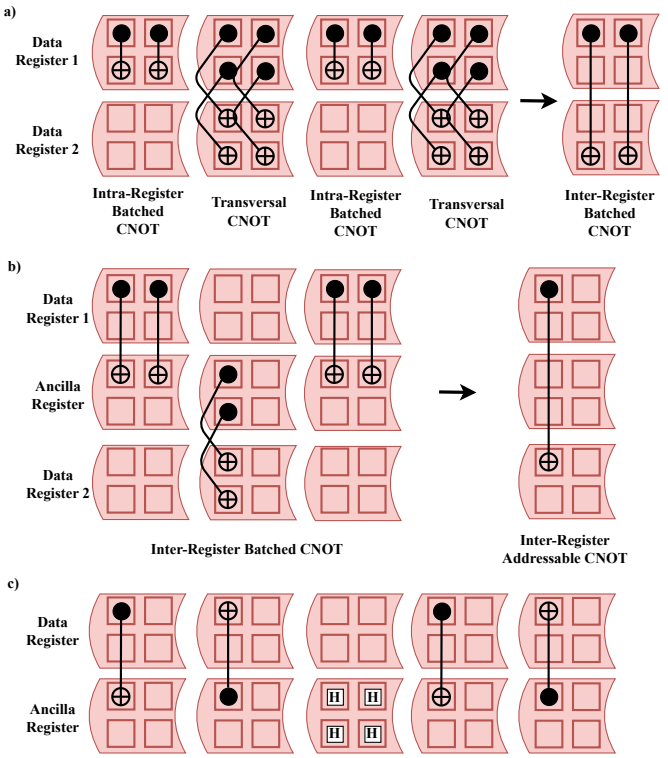

Fig. 7: Implementation of individually addressable logical gates: (a) Inter-register batched CNOT gates on multiple logical qubits. (b) Individually addressable CNOT gates achieved through hierarchical addressing. (c) Individually addressable H gates implemented through a combination of logical swaps and transversal operations. Note that error correction cycles are required for fault tolerance but are omitted here for clarity.

We first construct the *inter-register batched* CNOT gate, illustrated in Fig. 7 (a). This design is inspired by the bridge gate [21], which allows us to selectively operate on half of the logical qubits. Using ancilla registers, we continue this selection process until we can address a single logical qubit, as shown in Fig. 7 (b). For higher concatenation levels, this selection process can be repeated at each level of the hierarchy. Notably, SWAP and transversal CNOT operations can be performed in constant time with minimal overhead. Thus, the cost of hierarchical addressing is primarily determined by error correction.

Other Addressable Logical Instructions. Using the addressable logical CNOT together with the transversal H gadget, we can efficiently implement addressable logical H on specific logical qubits, as illustrated in Fig. 7 (c). Beyond Clifford instructions, non-Clifford instructions such as the T gate can be implemented through magic state preparation and teleportation using the addressable CNOT described above.

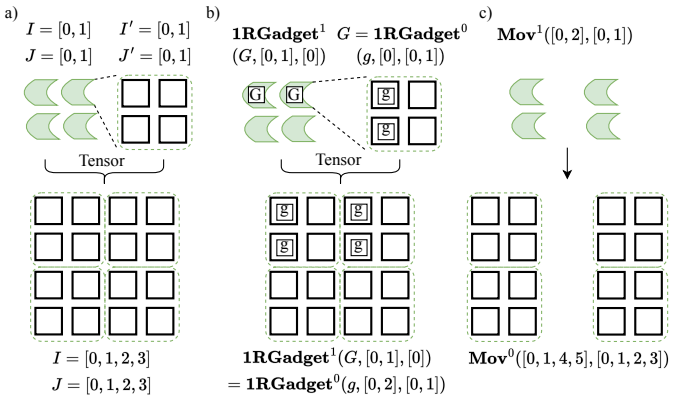

Fig. 8: VAIR lowering process. (a) Register mapping from four level-1 registers to their constituent level-0 registers. (b) An example showing how to implement level-1 gadgets using level-0 gadgets. (c) An example showing how to implement level-1 movements using level-0 movements.

VI. VAIR: Virtual Atom Intermediate Representation

At the core of ConiQ is the Virtual Atom Intermediate Representation (VAIR), which enables the efficient, level-wise compilation while ensuring legality and optimized compilation outcomes. By explicitly modeling logical states, instructions, and register constraints at every concatenation level, VAIR guarantees that any optimization or legalization performed at higher levels inherently respects physical-level constraints and preserves hardware parallelism.

We first define the VAIR abstraction formally, and then detail its key properties that enable level-wise compilation.

A. VAIR Definition

VAIR extends the neutral atom model to higher levels of abstraction, treating logical registers as virtual atoms to maintain consistent constraints across all levels. At each level-l, the system state is represented by a four-tuple $\mathcal{R}^l = (I, J, A, S)$, defined as follows:

- SLM registers are arranged on a unit-distance integer grid, with the i-th column and j-th row at (i, j).
- I and J denote the ordered x- and y-coordinates of the AOD registers, respectively, maintaining the ordering constraints ($I[i] < I[i']$ for $i < i'$, and similarly for J).
- $A[i][j]$ maps the location of the AOD register to its index. $A[i][j] = n$ means the n-th register is of type AOD and lies at $(I[i], J[j])$. An empty position is denoted by $\emptyset$.
- $S[i][j]$ maps the location of the SLM register to its index. $S[i][j] = n$ means the n-th register is of type SLM and lies at (i, j).

At every concatenation level-l, the supported instructions mirror their physical counterparts at level-0. Here, we use I' and J' to represent subsets of I and J, respectively:

- $\mathbf{Trans}^l_{\mathrm{a}\to\mathrm{s}}(I', J')$ and $\mathbf{Trans}^l_{\mathrm{s}\to\mathrm{a}}(I', J')$: These operations transfer registers between AOD and SLM types. The subscript indicates the transfer direction ($a \to s$ for AOD to SLM, $s \to a$ for SLM to AOD). For $\mathbf{Trans}^l_{\mathrm{s}\to\mathrm{a}}(I', J')$, if an SLM register exists where $S[i'][j'] = n$ with $i' \in I', j' \in J'$, and there exists an empty AOD position $A[i][j] = \emptyset$ where $I[i] = i'$ and $J[j] = j'$, then the register is transferred by setting $A[i][j] = n$ and $S[i'][j'] = \emptyset$. $\mathbf{Trans}^l_{\mathrm{a}\to\mathrm{s}}(I', J')$ performs the inverse operation.
- $\mathbf{Mov}^l(I, J)$: Moves AOD registers to new positions specified by I and J, transforming the state to (I, J, A, S). The operation must preserve ordering: $I[i] < I[i']$ for $i < i'$ and $J[j] < J[j']$ for $j < j'$.
- $\mathbf{1Rgadget}^l(G, I', J')$: For $S[i'][j'] = n$ where $i' \in I'$ and $j' \in J'$, apply the single-register gadget G to register n.
- $\mathbf{2Rgadget}^l(G, I', J')$: Applies a two-register gadget G between overlapping AOD-SLM register pairs specified by I' and J'. Specifically, for any SLM register $S[i'][j'] = m$ where $i' \in I'$ and $j' \in J'$, if there exists an AOD register n that overlaps with m in the x-y plane, the operation $G(n, m)$ is applied.

B. Level-wise Optimization and Legalization using VAIR

The uniformity of the VAIR abstraction across all concatenation levels enables efficient and independent compilation at each level. At level-0, VAIR reduces to the physical neutral atom model, guaranteeing physical feasibility of all compiled instructions. The compilation process comprises two main steps at each level: register mapping and gadget scheduling.

Register Mapping. To map a single level-l register to its constituent level-$(l-1)$ registers, we define a systematic coordinate-based approach. For a single level-l register consisting of N subregisters with coordinates (i'_k, j'_k), the k-th subregister of a level-l register at position (i, j) is mapped to the coordinates:

$$(i \cdot I^*_m + i'_k, j \cdot J^*_m + j'_k)$$

We denote $I^* = \{i'_1, \ldots, i'_N\}$, $J^* = \{j'_1, \ldots, j'_N\}$, $I^*_m = \max(I^*)$ and $J^*_m = \max(J^*)$. This mapping is illustrated in Fig. 8 (a).

The entire mapping of level-l registers can be concisely expressed using tensor products:

$$I \otimes I^* = \{i \cdot I^*_m + i' | i \in I, i' \in I^*\}$$

And similarly for J. This tensor product formulation allows us to describe a coarse-grained representation of high-level registers without needing to specify their lower-level implementations.

ConiQ adopts an alternating linear mapping strategy, alternating between x and y axes for different concatenation levels for its simplicity while producing an approximately square physical layout.

Gadget Scheduling. As a result of expressing register mappings as tensor products, each level-l instruction can be scheduled to level-$(l-1)$ instructions efficiently by the following rules:

- $\mathbf{Trans}^l_{\mathrm{a}\to\mathrm{s}}(I, J)$ and $\mathbf{Trans}^l_{\mathrm{s}\to\mathrm{a}}(I, J)$: These transfers are implemented as $\mathbf{Trans}^{l-1}_{\mathrm{a}\to\mathrm{s}}(I \otimes I^*, J \otimes J^*)$.

Algorithm 1: Greedy Scheduling of Lower-Level Instructions

Input: A sequence of level-$(l-1)$ instructions $[g_1, g_2, \ldots, g_n]$ that implements a higher-level gadget G

Output: Optimized schedule S of level-$(l-1)$ instruction represented in VAIR

```
S ← [] ;      // Initialize empty schedule
remaining ← [g_1, g_2, ..., g_n] ;   // Instructions to be scheduled
while remaining ≠ ∅ do
    F ← frontLayer(remaining) ;      // Extract instructions with no dependencies
    instructionType ← randomSelect(F) ;
     // Randomly choose an instruction type
    batchedSet ←
     maximalAddressableSet(F, instructionType) ;
     // Find largest set satisfying row-column constraint
    thisInstruction ←
     generateInstruction(batchedSet, instructionType) ;
     // Generate the instruction represented in VAIR
    append(S, thisInstruction) ;    // Add batched instruction to schedule
    remaining ← remaining \ thisInstruction ;
     // Remove scheduled instructions
return S
```

- $\mathbf{Mov}^l(I, J)$: Implemented as $\mathbf{Mov}^{l-1}(I \otimes I^*, J \otimes J^*)$. We can readily verify that $I \otimes I^*$ and $J \otimes J^*$ satisfy the order-preserving constraint if the input I and J satisfy the constraint. An example is illustrated in Fig. 8 (c).
- $\mathbf{1Rgadget}^l(G, I, J)$: If G is composed of n level-$(l-1)$ gadgets $\prod_k \mathbf{1Rgadget}^{l-1}(g_k, I'_k, J'_k)$, executable in c cycles, then $\mathbf{1Rgadget}^l(G, I, J)$ can be executed in the same c cycles by being decomposed as $\prod_k \mathbf{1Rgadget}^{l-1}(g_k, I \otimes I'_k, J \otimes J'_k)$. An example is illustrated in Fig. 8 (b).
- $\mathbf{2Rgadget}^l(G, I, J)$: If G is composed of n level-$(l-1)$ gadgets $\prod_k \mathbf{2Rgadget}^{l-1}(g_k, I'_k, J'_k)$, executable in c cycles, then $\mathbf{2Rgadget}^l(G, I, J)$ can be executed in the same c cycles by being decomposed as $\prod_k \mathbf{2Rgadget}^{l-1}(g_k, I \otimes I'_k, J \otimes J'_k)$.

To optimize scheduling within each concatenation level, we utilize a greedy batching algorithm (illustrated in Algorithm 1), ensuring high parallelism while maintaining legality constraints.

C. Key Properties and Guarantees of VAIR

By iteratively applying the mapping and lowering steps enabled by VAIR, we achieve two critical properties for the compiled schedules:

e_{1Q}	e_{2Q}	e_{move}	e_{reset}	e_{meas}
0.03%	0.5%	0.1%	0.25%	0.25%

TABLE II: Error rates used in our neutral atom array simulation model, based on recent experimental results [6].

- **Level-wise legalization ensures global legalization:** Any level-l gadget expressed in VAIR guarantees native executability on neutral atom arrays once compiled down to physical instructions.
- **Level-wise optimization ensures global optimization:** Optimizing schedules at level-l directly improves their compiled physical-level implementations, ensuring monotonic performance gains.

These properties empower ConiQ to efficiently compile concatenated codes into physically realizable instructions, significantly outperforming conventional compilation approaches.

VII. Evaluation

We present a comprehensive evaluation of ConiQ. First, we compare its performance against state-of-the-art baselines on key fault-tolerant gadgets and logical-level subroutines. Subsequently, we conduct an ablation study to quantify the individual contributions of the two main innovations in ConiQ: the AHA `CNOT` gate scheme and the compilation framework based on VAIR.

A. Evaluation Methodology

Baselines. Our evaluation examines three distinct compilers: Atomique [31], Enola [30], and ConiQ. For `CNOT` gate schemes, we evaluate both distillation-based approaches and our AHA scheme. All compilation experiments were conducted on a machine equipped with an AMD EPYC 7763 CPU and 256GB RAM with a time limit of 24 hours. For both Atomique and Enola, since the default configuration can't complete most of our benchmarks within the time limit, we use their most scalable configurations.

Benchmarks. Our evaluation focuses on three representative fault-tolerant gadgets: (1) *state preparation*, the fundamental building block for initializing logical qubits; (2) *logical `CNOT` gates*, essential for universal quantum computation; and (3) *16-qubit GHZ state preparation*, which probes multi-qubit entanglement and synchronization capabilities. We evaluate these benchmarks across four different code configurations: $D_{4,4}$, $D_{4,4,4}$, $D_{4,4,4,4}$, and $D_{4,4,6,6}$[2]. For comparing many-hypercube codes with alternative quantum QEC codes, we employ the memory experiment, which measures the logical error rate after one round of error correction as a function of the physical error rate [1].

Error Model. For the memory experiment, we consider five primary error sources in neutral atom arrays: two-qubit gate errors (e_{2Q}), single-qubit gate errors (e_{1Q}), atom movement errors (e_{move}), qubit reset errors (e_{reset}), and measurement

[2]We choose $D_{4,4,6,6}$ instead of $D_{6,6,4,4}$ because D_4 codes have better threshold than D_6 codes, and concatenating codes with higher threshold in the lower levels is more advantageous [36], [35].

errors (e_{meas}). Given the long coherence times of neutral atoms, decoherence errors are negligible and thus omitted from our model. We use the concrete error rates reported in [6], summarized in Table II. For experiments requiring a range of physical error rates, we treat the two-qubit gate error rate as the reference point and scale all other error sources proportionally according to Table II.

Simulation Tools. Surface code simulations utilize the open-source Stim framework [11] in conjunction with pyMatching [19] for logical error rate estimation. For the many-hypercube code, which leverages a concatenated construction, we developed a specialized simulator that exploits the underlying hierarchical structure, enabling accelerated sampling.

Decoding. We implement the level-by-level decoding approach proposed in [14]. This decoding algorithm iteratively identifies the minimum-distance codeword at lower levels, then propagates these results to decode codewords at higher levels and has been demonstrated to outperform both hard-decision and Bayesian-based decoding methods for concatenated codes.

B. ConiQ's Overall Performance

Table III summarizes the performance of ConiQ against the baselines, demonstrating that ConiQ significantly outperforms conventional compilers across all metrics. For state preparation on $D_{4,4,6,6}$, compilation times decrease from several hours with Atomique or 50.9 seconds with Enola to merely 0.001 seconds with ConiQ. While Atomique fails to compile level-4 CNOT and GHZ circuits within the 24-hour time limit, ConiQ consistently completes these compilations within 0.2 seconds, and reduces the spacetime product by up to 2×10^3 (CNOT gate on $D_{4,4,6,6}$) compared to Enola. Moreover, our AHA CNOT gate scheme reduces the spacetime product by up to $8\times$ for a single CNOT gate and up to $20\times$ for GHZ state preparation.

These substantial performance gains stem from two core innovations in ConiQ: efficient addressable gates enabled by the AHA gate scheme and parallelism-preserving scheduling enabled by the VAIR. We analyze these two components in greater detail in the following sections.

C. Impact of AHA CNOT Gate Scheme and Scheduling with VAIR

Since Enola consistently outperforms Atomique, we focus our comparative analysis on ConiQ with Enola.

Level-wise Scheduling. ConiQ's VAIR framework enables level-wise optimization and legalization that effectively mitigates cascading latency amplification. As shown in Fig. 9 (a), space-time reduction grows significantly with increasing concatenation levels, reaching 10^3 at level-4. The nearly straight line in logarithmic scale confirms that ConiQ successfully avoids the multiplicative overhead that would otherwise compound across levels.

AHA CNOT Gate Scheme. Our AHA scheme maps logical CNOT operations directly to controlled atom movements compatible with row-column addressability constraints. This reduces overhead compared to distillation-based approaches, yielding up to $8\times$ reduction in spacetime product for CNOT operations and $20\times$ for GHZ state preparation, with benefits increasing at higher concatenation levels. Fig. 9 (b) shows the AHA scheme performs better with ConiQ than Enola because ConiQ can better exploit the higher parallelism of transversal gates.

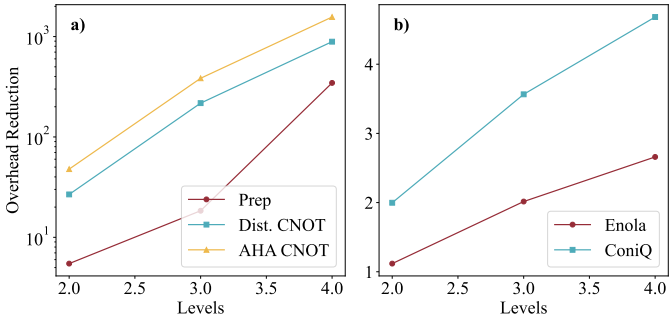

Fig. 9: Overhead reduction factors (higher is better) on concatenated D_4 codes: (a) Spacetime overhead reduction achieved by VAIR compared to Enola across concatenation levels. (b) Efficiency comparison between AHA and distillation-based CNOT gate implementations.

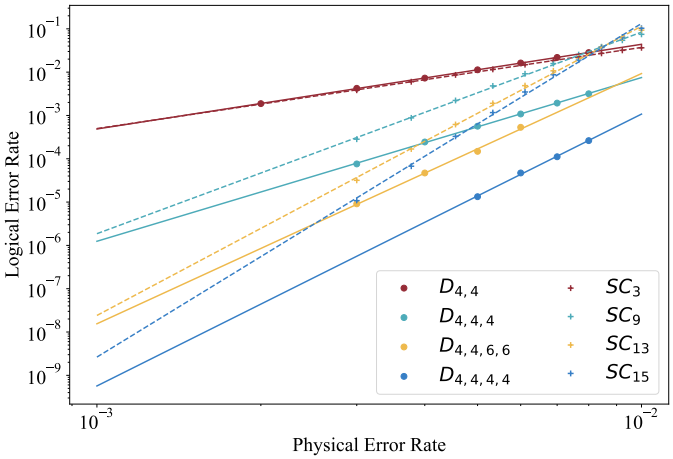

Fig. 10: Logical error rate as a function of physical error rate. SC_d denotes surface codes with distance d.

Together, the AHA scheme and parallelism-preserving scheduling deliver substantial resource savings, demonstrating the importance of hardware-aware optimizations for scalable quantum error correction on neutral atom arrays.

D. Discussion: Many-hypercube Codes vs. Other Codes

We further assess the performance of many-hypercube codes by comparing their logical error rate and space overhead against other leading candidate codes for neutral atom arrays.

Comparison with Surface Codes. Fig. 10 compares logical error rates of many-hypercube codes and surface code during memory experiments. Many-hypercube codes use Steane-style error correction while the surface codes use Shor-style error correction with d rounds of stabilizer measurements per logical time step [20].

We extrapolate the logical-to-physical error rate relationship as $p_{\text{l}} = \beta(\frac{p_{\text{ph}}}{p_{\text{th}}})^{\alpha}$, where α is the code distance scaling factor, β is the error coefficient, and p_{th} is the threshold error rate.

Compiler	Enola				ConiQ			
CNOT Scheme	Distillation		AHA		Distillation		AHA	
Metric	Comp. Time	S.T. Prod.	Comp. Time	S.T. Prod.	Comp. Time	S.T. Prod.	Comp. Time	S.T. Prod.
State Prep($D_{4,4}$)	0.06	0.004	0.06	0.004	0.0001	0.0007	0.0001	0.0007
State Prep($D_{4,4,4}$)	0.4	0.1	0.4	0.1	0.0005	0.007	0.0005	0.007
State Prep($D_{4,4,4,4}$)	22.1	34.7	22.1	34.7	0.001	0.1	0.001	0.1
State Prep($D_{4,4,6,6}$)	50.9	87.6	50.9	87.6	0.001	0.2	0.001	0.2
CNOT($D_{4,4}$)	1.6	1.4	1.5	1.3	0.002	0.05	0.0007	0.03
CNOT($D_{4,4,4}$)	147.0	239.0	68.1	118.6	0.008	1.1	0.005	0.3
CNOT($D_{4,4,4,4}$)	12508.0	17820.8	3999.6	6700.1	0.03	20.0	0.04	4.3
CNOT($D_{4,4,6,6}$)	73371.1	78612.1	9908.7	15002.6	0.04	54.6	0.06	6.8
GHZ($D_{4,4,4,4}$)	50561.1	72188.2	529.3	905.0	0.1	83.2	0.01	3.1
GHZ($D_{4,4,6,6}$)	T.O.	T.O.	3403.1	11156.5	0.2	227.6	0.02	9.2

TABLE III: Performance comparison of different compilers and different CNOT gate implementation schemes. Results show compilation time (Comp. Time, in seconds) and space-time overhead (S.T. Prod., in 10^6 qubit-cycles) across different codes and benchmark circuits. "T.O." indicates timeout (> 24 hours). Note: the overhead of state preparation does not depend on the CNOT implementation scheme.

At a physical error rate of 0.1%, $D_{4,4,4,4}$ achieves better logical error rates than SC_{15} while using only 16 physical qubits per logical qubit compared to 225 for the surface code—a $14\times$ reduction. The $D_{4,4,6,6}$ code achieves an even greater $19\times$ reduction in physical qubit requirements.

Comparison with HGP Codes. Fig. 11 compares our $D_{4,4,4,4}$ code with the hypergraph product (HGP) codes [34]. The HGP codes treat the entire qubit array as a single logical block with error rates improving with system size, while $D_{4,4,4,4}$ maintains constant logical error rates using fixed-size blocks of 4^4 physical qubits. Though HGP has advantages in scenarios with lower physical error rates and high qubit counts, our analysis shows that it requires over 10^5 physical qubits at 0.1% error rate or 5×10^4 qubits at 0.01% to outperform $D_{4,4,4,4}$, far exceeding what will be available in the near future.

Summary. The many-hypercube codes achieves competitive logical error rates with substantially lower space overhead compared to surface codes while avoiding the prohibitive qubit requirements of HGP codes, making it well-suited for near-term neutral atom quantum devices with limited qubit availability and quality. However, it is important to note that the relative advantage changes when implementing logical gates. A comprehensive evaluation of space-time costs in the context of complete quantum algorithms remains an important direction for future research, particularly considering the difference between Shor- and Steane-style error correction, and new decoding algorithms that achieve effective single-shot error correction in surface codes should also be considered [37].

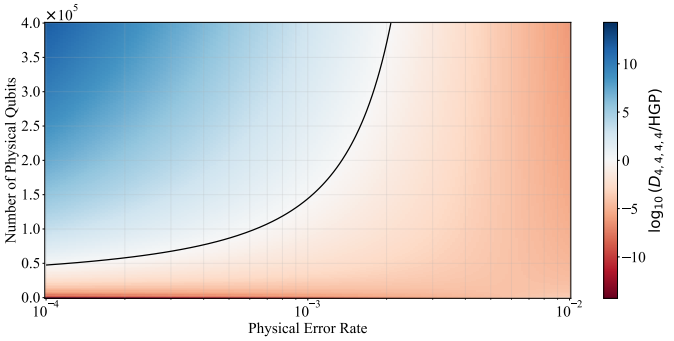

Fig. 11: Comparative performance of $D_{4,4,4,4}$ and HGP codes under varying physical error rates and qubit counts. Color indicates the ratio of logical error rates ($D_{4,4,4,4}$ to HGP). Blue region indicates HGP outperforms $D_{4,4,4,4}$ and the black line indicates the boundary where the two codes have the same logical error rate.

VIII. Related Work

Compilers for Neutral Atom Arrays. Several compilers for neutral atom arrays have been proposed [3], [27], [29], [31], [32], [30]. These compilers could be seamlessly integrated into the VAIR framework to enhance the simple greedy algorithm currently employed in ConiQ for optimizing register mapping and scheduling. Additionally, the state preparation scheme of many-hypercube codes relies on post-selection, which bears significant similarities to atom loss management in neutral atom arrays. Techniques developed for handling atom loss [3], [27] may provide valuable insights for efficiently managing post-selection in our context. Further, the parallel control of multiple physical qubits to implement parallel logical operations demonstrated in Ref. [5] can also be viewed as a special case of the VAIR model.

Compilers for Fault-Tolerant Quantum Computing. Prior research on fault-tolerant quantum computing compilers has predominantly focused on surface codes, addressing diverse aspects including resource estimation [4], compilation for

trapped ion [23] and superconducting [33], [25] architectures.

Concatenated QEC Codes. Concatenated codes have a rich history in fault-tolerant quantum computing, and our compilation framework can be readily applied to various concatenated code constructions. By concatenating the 7-qubit Steane code with the 15-qubit Reed-Muller code, Ref. [8] demonstrates a 105-qubit code capable of performing `CNOT` gates efficiently and implementing `H` and `T` gates with relative ease. Through concatenation of high-threshold codes with high-rate codes, Ref. [36], [35] achieves a 2.4% threshold (7 times higher than the surface code) while requiring 10 times less space overhead than conventional surface codes. Ref. [28], [13] introduced hierarchical codes and yoked surface code respectively, which concatenate surface code with other codes, enabling high code rates with 2D topological constraints.

IX. Conclusion and Outlook

In this paper, we have presented ConiQ, a specialized compiler for efficiently implementing concatenated QEC codes on neutral atom arrays. Through the VAIR model and the AHA gate scheme, we effectively manage the complex constraints imposed by neutral atom array architectures and achieve orders of magnitude improvement in both spacetime overhead and compilation time compared to state-of-the-art compilers. These results establish the many-hypercube codes as a promising candidate for fault-tolerant quantum computation in the near future.

While our current evaluation focuses primarily on Clifford circuits, our compiler framework can be straightforwardly extended to support arbitrary quantum circuits through consuming magic states.

Acknowledgments

This research was supported by the following NSF grants CCF-1901381, CCF-2115104, CCF-2119352, CCF-2107241. We are grateful to Chameleon Cloud for providing the compute cycles needed for the experiments.

References

[1] R. Acharya, L. Aghababaie-Beni, I. Aleiner, T. I. Andersen, M. Ansmann, F. Arute, K. Arya, A. Asfaw, N. Astrakhantsev, J. Atalaya *et al.*, "Quantum error correction below the surface code threshold," *arXiv preprint arXiv:2408.13687*, 2024.

[2] V. V. Albert and P. Faist, Eds., *The Error Correction Zoo*, 2025. [Online]. Available: https://errorcorrectionzoo.org/

[3] J. M. Baker, A. Litteken, C. Duckering, H. Hoffmann, H. Bernien, and F. T. Chong, "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)*. IEEE, 2021, pp. 818–831.

[4] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, "Assessing requirements to scale to practical quantum advantage," *arXiv preprint arXiv:2211.07629*, 2022.

[5] D. Bluvstein, S. J. Evered, A. A. Geim, S. H. Li, H. Zhou, T. Manovitz, S. Ebadi, M. Cain, M. Kalinowski, D. Hangleiter *et al.*, "Logical quantum processor based on reconfigurable atom arrays," *Nature*, vol. 626, no. 7997, pp. 58–65, 2024.

[6] D. Bluvstein, H. Levine, G. Semeghini, T. T. Wang, S. Ebadi, M. Kalinowski, A. Keesling, N. Maskara, H. Pichler, M. Greiner *et al.*, "A quantum processor based on coherent transport of entangled atom arrays," *Nature*, vol. 604, no. 7906, pp. 451–456, 2022.

[7] A. R. Calderbank and P. W. Shor, "Good quantum error-correcting codes exist," *Physical Review A*, vol. 54, no. 2, p. 1098, 1996.

[8] C. Chamberland, T. Jochym-O'Connor, and R. Laflamme, "Thresholds for universal concatenated quantum codes," *Physical review letters*, vol. 117, no. 1, p. 010501, 2016.

[9] I. Dinur, M.-H. Hsieh, T.-C. Lin, and T. Vidick, "Good quantum ldpc codes with linear time decoders," in *Proceedings of the 55th annual ACM symposium on theory of computing*, 2023, pp. 905–918.

[10] S. J. Evered, D. Bluvstein, M. Kalinowski, S. Ebadi, T. Manovitz, H. Zhou, S. H. Li, A. A. Geim, T. T. Wang, N. Maskara *et al.*, "High-fidelity parallel entangling gates on a neutral-atom quantum computer," *Nature*, vol. 622, no. 7982, pp. 268–272, 2023.

[11] C. Gidney, "Stim: a fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, Jul. 2021. [Online]. Available: https://doi.org/10.22331/q-2021-07-06-497

[12] C. Gidney and M. Ekerå, "How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, p. 433, 2021.

[13] C. Gidney, M. Newman, P. Brooks, and C. Jones, "Yoked surface codes," *arXiv preprint arXiv:2312.04522*, 2023.

[14] H. Goto, "Many-hypercube codes: High-rate quantum error-correcting codes for high-performance fault-tolerant quantum computation," *arXiv preprint arXiv:2403.16054*, 2024.

[15] D. Gottesman, "An introduction to quantum error correction," in *Proceedings of Symposia in Applied Mathematics*, vol. 58, 2002, pp. 221–236.

[16] ——, "Opportunities and challenges in fault-tolerant quantum computation," *arXiv preprint arXiv:2210.15844*, 2022.

[17] T. M. Graham, Y. Song, J. Scott, C. Poole, L. Phuttitarn, K. Jooya, P. Eichler, X. Jiang, A. Marra, B. Grinkemeyer, M. Kwon, M. Ebert, J. Cherek, M. T. Lichtman, M. Gillette, J. Gilbert, D. Bowman, T. Ballance, C. Campbell, E. D. Dahl, O. Crawford, N. S. Blunt, B. Rogers, T. Noel, and M. Saffman, "Multi-qubit entanglement and algorithms on a neutral-atom quantum computer," *Nature*, vol. 604, no. 7906, pp. 457–462, Apr 2022. [Online]. Available: https://doi.org/10.1038/s41586-022-04603-6

[18] M. B. Hastings, J. Haah, and R. O'Donnell, "Fiber bundle codes: breaking the n 1/2 polylog (n) barrier for quantum ldpc codes," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, 2021, pp. 1276–1288.

[19] O. Higgott and C. Gidney, "Sparse blossom: correcting a million errors per core second with minimum-weight matching," *arXiv preprint arXiv:2303.15933*, 2023.

[20] S. Huang, K. R. Brown, and M. Cetina, "Comparing shor and steane error correction using the bacon-shor code," *Science Advances*, vol. 10, no. 45, p. eadp2008, 2024.

[21] T. Itoko, R. Raymond, T. Imamichi, and A. Matsuo, "Optimization of quantum circuit mapping using gate transformation and commutation," *Integration*, vol. 70, pp. 43–50, 2020.

[22] E. Knill, "Quantum computing with very noisy devices," *arXiv preprint quant-ph/0410199*, 2007.

[23] T. LeBlond, R. S. Bennink, J. G. Lietz, and C. M. Seck, "Tiscc: A surface code compiler and resource estimator for trapped-ion processors," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1426–1435.

[24] H. J. Manetsch, G. Nomura, E. Bataille, K. H. Leung, X. Lv, and M. Endres, "A tweezer array with 6100 highly coherent atomic qubits," *arXiv preprint arXiv:2403.12021*, 2024.

[25] A. Molavi, A. Xu, S. Tannu, and A. Albarghouthi, "Compilation for surface code quantum computers," *arXiv preprint arXiv:2311.18042*, 2023.

[26] P. Panteleev and G. Kalachev, "Asymptotically good quantum and locally testable classical ldpc codes," in *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, 2022, pp. 375–388.

[27] T. Patel, D. Silver, and D. Tiwari, "Geyser: a compilation framework for quantum computing with neutral atoms," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 383–395.

[28] C. A. Pattison, A. Krishna, and J. Preskill, "Hierarchical memories: Simulating quantum ldpc codes with local gates," *arXiv preprint arXiv:2303.04798*, 2023.

[29] B. Tan, D. Bluvstein, D. M. Lukin, and J. Cong, "Qubit mapping for reconfigurable atom arrays," *ICCAD*, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3508352.3549331

[30] D. B. Tan, W.-H. Lin, and J. Cong, "Compilation for dynamically field-programmable qubit arrays with efficient and provably near-optimal scheduling," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 921–929.

[31] H. Wang, P. Liu, D. B. Tan, Y. Liu, J. Gu, D. Z. Pan, J. Cong, U. A. Acar, and S. Han, "Atomique: A quantum compiler for reconfigurable neutral atom arrays," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 293–309.

[32] H. Wang, B. Tan, P. Liu, Y. Liu, J. Gu, J. Cong, and S. Han, "Q-pilot: field programmable quantum array compilation with flying ancillas," *arXiv preprint arXiv:2311.16190*, 2023.

[33] G. Watkins, H. M. Nguyen, K. Watkins, S. Pearce, H.-K. Lau, and A. Paler, "A high performance compiler for very large scale surface code computations," *Quantum*, vol. 8, p. 1354, 2024.

[34] Q. Xu, J. P. Bonilla Ataides, C. A. Pattison, N. Raveendran, D. Bluvstein, J. Wurtz, B. Vasić, M. D. Lukin, L. Jiang, and H. Zhou, "Constant-overhead fault-tolerant quantum computation with reconfigurable atom arrays," *Nature Physics*, pp. 1–7, 2024.

[35] H. Yamasaki and M. Koashi, "Time-efficient constant-space-overhead fault-tolerant quantum computation," *Nature Physics*, pp. 1–7, 2024.

[36] S. Yoshida, S. Tamiya, and H. Yamasaki, "Concatenate codes, save qubits," *arXiv preprint arXiv:2402.09606*, 2024.

[37] H. Zhou, C. Zhao, M. Cain, D. Bluvstein, C. Duckering, H.-Y. Hu, S.-T. Wang, A. Kubica, and M. D. Lukin, "Algorithmic fault tolerance for fast quantum computing," *arXiv preprint arXiv:2406.17653*, 2024.