

PrefixAgent: An LLM-Powered Design Framework for Efficient Prefix Adder Optimization

DONGSHENG ZUO, The Hong Kong University of Science and Technology (Guangzhou), China

JIADONG ZHU, The Hong Kong University of Science and Technology (Guangzhou), China

YANG LUO, The Hong Kong University of Science and Technology (Guangzhou), China

YUZHENG MA*, The Hong Kong University of Science and Technology (Guangzhou), China

Prefix adders are fundamental arithmetic circuits, but their design space grows exponentially with bit-width, posing significant optimization challenges. Previous works face limitations in performance, generalization, and scalability. To address these challenges, we propose *PrefixAgent*, a large language model (LLM)-powered framework that enables efficient prefix adder optimization. Firstly, we reformulate the problem into two subtasks, namely backbone synthesis and structure refinement, which effectively reduces the search space. The LLM performs these two phases of optimization by invoking tools through function calls, enabling it to iteratively construct the backbone and refine local structures based on reasoning and EDA feedback. Secondly, this new design perspective allows us to systematically collect large-scale, high-quality supervision data. We leverage the rewriting and equality saturation capabilities of e-graphs to comprehensively explore the prefix adder solution space. In addition, the explainability of e-graphs enables us to extract fine-grained rewrite trajectories, which serve as interpretable and effective training data. *PrefixAgent* is then fine-tuned on this dataset, significantly enhancing its optimization ability and reasoning generalization. Experimental results show that *PrefixAgent* synthesizes prefix adders with smaller areas than baseline methods in nearly all configurations, with the advantage growing at larger bit-widths, and it remains effective under a commercial EDA flow.

CCS Concepts: • **Hardware** → **Combinational synthesis**; **Circuit optimization**; **Datapath optimization**.

Additional Key Words and Phrases: Prefix adder, large language models, logic synthesis, e-graph, datapath optimization

ACM Reference Format:

Dongsheng Zuo, Jiadong Zhu, Yang Luo, and Yuzhe Ma. 2026. *PrefixAgent: An LLM-Powered Design Framework for Efficient Prefix Adder Optimization*. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (January 2026), 25 pages. <https://doi.org/10.1145/3843236>

1 Introduction

Adders are fundamental arithmetic components used in digital systems like processors, digital signal processors, and AI chips. Among various adder architectures, the prefix adder is well-known for its parallel structure and efficiency. The prefix adder's internal operation is represented by a prefix graph, where each node computes group propagate and generates signals for rapid parallel carry computation. However, the design space of prefix adders grows exponentially with bit-width,

*Corresponding author: Yuzhe Ma.

Authors' Contact Information: Dongsheng Zuo, The Hong Kong University of Science and Technology (Guangzhou), China; Jiadong Zhu, The Hong Kong University of Science and Technology (Guangzhou), China; Yang Luo, The Hong Kong University of Science and Technology (Guangzhou), China; Yuzhe Ma, The Hong Kong University of Science and Technology (Guangzhou), China.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1557-7309/2026/1-ART

<https://doi.org/10.1145/3843236>

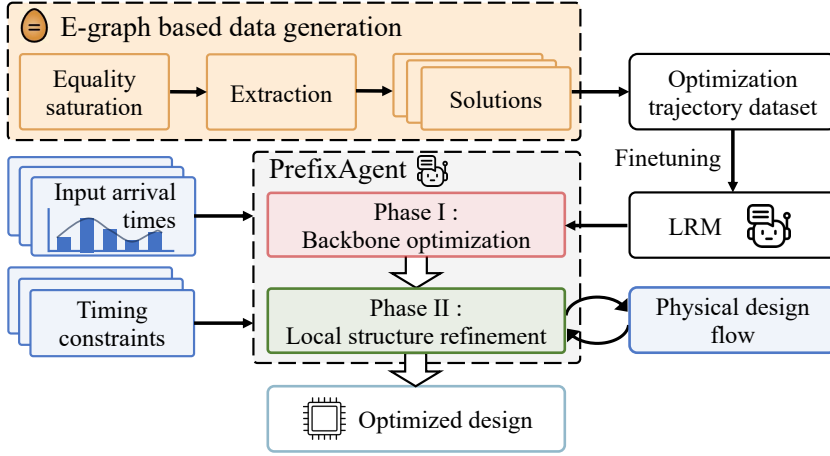


Fig. 1. The two-phase framework in *PrefixAgent*.

making exhaustive exploration impractical. Traditionally, adder designs rely on regular structures or cumbersome manual optimization. Therefore, developing efficient and flexible adder design methodologies is essential.

Previous works on adder design primarily involve regular structures and automated design methods. Regular adder structures such as Kogge–Stone [1], Sklansky [2], Brent–Kung [3], and Han–Carlson [4] have been proposed, focusing on trade-offs among logic level, maximum fanout, and wiring tracks. However, these regular structures may not satisfy diverse requirements. Practical design requires balancing area, delay, and power while considering timing constraints as well as various input arrival time profiles. Consequently, various heuristics and search-based methods are proposed [5–14], which improve the design efficiency to some extent. However, it is difficult to design a generic optimization approach that can cover a rich set of design scenarios. In addition, a non-trivial gap persists between structural-level metrics and physical-implementation metrics. Machine learning (ML) methods have been proposed to bridge this gap. Recent works [15–17] utilized ML models to predict the metrics and optimize designs over pruned adder solution spaces. More recently, reinforcement learning (RL) [18–22], variational autoencoder (VAE) [23], and Monte Carlo tree search (MCTS) [24] methods have demonstrated progress in arithmetic circuit optimization tasks. These methods iteratively modify structures, guided by actual physical metrics. However, these methods suffer from challenges in generalization and scalability due to the extensive training required for each new scenario and the inefficiency of trial-and-error exploration in large action spaces.

Inspired by manual design practices, experienced engineers typically choose a suitable initial structure based on design specifications and refine it using detailed feedback from EDA tools. Although effective, this process requires significant engineering effort. An intelligent agent with strong reasoning and interactive capabilities can automate the design process, addressing generalization challenges and enabling more efficient, scalable optimization. Motivated by this, emerging large language models (LLMs) provide a promising solution to further automate and optimize the prefix adder design process. Recent advancements in large reasoning models (LRMs) [25–29] have demonstrated their powerful reasoning capabilities and potential for solving complex problems. In circuit design tasks, LLMs also have shown potential in analog circuit design [30–32] and RTL code design [33]. However, applying LRMs to the prefix adder design task still faces two main challenges.

The first challenge is the limited capability of large models to efficiently generate and reason about complex graphs with more than 20 nodes [34–36]. This issue is particularly evident in the EDA domain, where existing LLM-based methods have shown promise for small graph circuits, such as analog circuits with fewer than 10 nodes [30, 31]. Recent work [37] attempts to employ LRMs to generate textual representations of prefix graphs; however, it still faces graph reasoning challenges. Specifically, the node count in a prefix graph theoretically scales quadratically with bit-width, and these graphs must adhere to stringent structural constraints, further complicating the generation process. The second challenge is the lack of high-quality data, which is essential for fine-tuning LRMs on the prefix adder design task. Recent studies highlight the importance of high-quality data in improving the reasoning capabilities of LRMs [38–40]. However, simple random data generation or perturbation-based strategies are ineffective in producing meaningful optimization trajectories for training.

In this paper, we propose *PrefixAgent*, an LLM-powered framework that enables efficient prefix adder optimization. The overall workflow of the framework is illustrated in Figure 1. To address LLMs’ limitations on complex graphs, our framework considers the prefix adder design problem from a new perspective, which decomposes the prefix adder synthesis task into two subtasks, namely *backbone generation* and *local structure refinement*. The backbone, representing the critical structural component of the adder, significantly simplifies the overall design complexity. The new perspective also enables the LLM to efficiently generate zero-deficiency [41] or low-deficiency adders. Rather than requiring the LLM to generate fully valid prefix structures with high complexity, *PrefixAgent* adopts the tool-integrated reasoning (TIR) technique [42, 43], which delegates precise local modifications to external tools during the reasoning process. This empowers the LLM’s reasoning capabilities, allowing it to focus on high-level decision-making based on detailed feedback after physical implementation. Furthermore, based on the backbone-centric optimization paradigm for prefix adders, we observed a correspondence between backbone optimization and e-graph-based rewriting. This enables us to gather high-quality data and interpretable reasoning trajectories using e-graphs, successfully addressing the second challenge, i.e., data scarcity, in building LLM agents. By fine-tuning the LLM on these carefully curated datasets, *PrefixAgent* learns to efficiently optimize adders under various scenarios and design requirements. Compared to RL [18], MCTS [24], VAE [23], PrefixGPT [44], and prior LLM methods [37], *PrefixAgent* generates prefix adders with smaller areas in nearly all cases, with a widening advantage at larger bit-widths. Under commercial physical design flows, it achieves superior results to adders generated by commercial tools. The key contributions of this work are summarized as follows:

- We propose *PrefixAgent*, an LLM-powered framework for efficient prefix adder optimization, representing the first LLM-based approach that scales to 64-bit designs and outperforms commercial tools.
- A new design perspective is proposed for the prefix adder, which consists of backbone generation and local refinement. An agent is utilized to perform reasoning for efficient backbone synthesis and targeted optimization guided by detailed EDA feedback.
- Owing to the new design perspective, we utilize the e-graph-based method to systematically generate comprehensive, interpretable, and high-quality optimization trajectories for the backbone optimization task, enabling effective fine-tuning of the agent.
- Experimental results show that *PrefixAgent* synthesizes prefix adders with smaller areas than baseline methods in nearly all configurations, and outperforms the commercial synthesis tool.

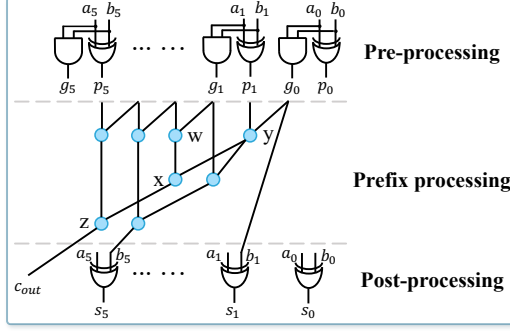


Fig. 2. Structure of a prefix adder.

2 Preliminary

2.1 Prefix Adder

Given two n -bit binary numbers, $A = a_{n-1}a_{n-2} \dots a_0$ and $B = b_{n-1}b_{n-2} \dots b_0$, the prefix adder computes the sum $S = s_{n-1}s_{n-2} \dots s_0$ and the carry-out bit C_{out} . The operation is performed in three stages: pre-processing, prefix computation, and post-processing.

Pre-processing: Each bit position independently computes propagate (p_i) and generate (g_i) signals:

$$p_i = a_i \oplus b_i, \quad g_i = a_i \cdot b_i. \quad (1)$$

Prefix Computation: The propagate and generate signals are combined recursively using the associative prefix operator o defined as:

$$\begin{aligned} (G, P)_{i:j} &= (G, P)_{i:k} \circ (G, P)_{k-1:j} \\ &= \left(G_{i:k} + P_{i:k} \cdot G_{k-1:j}, P_{i:k} \cdot P_{k-1:j} \right). \end{aligned} \quad (2)$$

As the prefix operator o is associative, it allows flexibility in grouping strategies. Each o operation corresponds to a node in the prefix graph, and different groupings lead to different graph topologies. This recursive process continues until each bit position i is assigned an output node to compute $G_{i:0}$, forming the final prefix structure.

Post-processing: The final carry and sum bits are computed using the prefix results:

$$c_i = G_{i:0}, \quad s_i = p_i \oplus c_{i-1} \quad (3)$$

We adopt the notation from [13], where each prefix node (i, j) represents the computation of (i, k) and $(k-1, j)$, with $i \geq k > j$. Here, i and j denote the most significant bit (MSB) and least significant bit (LSB), respectively. We refer to (i, k) as the *trivial fanin* (fi_t) and $(k-1, j)$ as the *non-trivial fanin* (fi_{nt}) of node (i, j) . We further classify the fanouts of each node into *trivial fanout* (fo_t) and *non-trivial fanout* (fo_{nt}): A fanout is labeled as fo_t if it shares the same MSB as its parent, and fo_{nt} if the MSB is larger. For example, node (i, j) is a fo_t of (i, k) and a fo_{nt} of $(k-1, j)$. In Figure 2, node x is a fo_t of node w and a fo_{nt} of node y .

Under the standard (i, j) grid representation, an n -bit prefix graph has at most $O(n^2)$ potential node positions. Given a nodelist, we first map it to this grid and, for each existing node, verify the consistency of its fanin (and optionally fanout) relations in constant time, leading to an $O(n^2)$ worst-case complexity for checking whether the prefix graph is legal.

A zero-deficiency adder aims for a balanced design with minimal cost in both prefix graph size and depth. According to Snir's theorem [41], a prefix adder is defined as zero-deficiency if its size

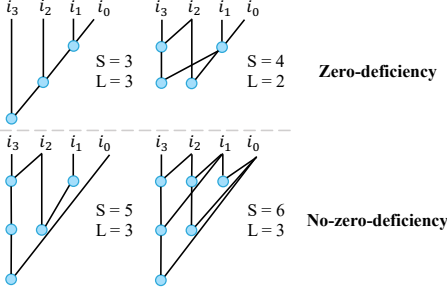


Fig. 3. Comparison of zero and non-zero-deficiency adders.

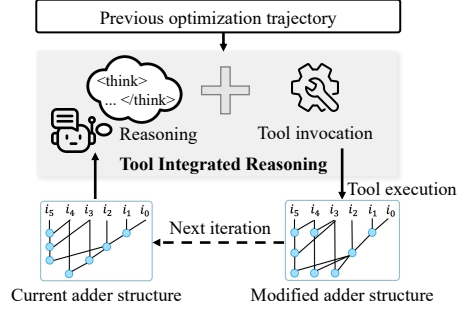


Fig. 4. Tool integrated reasoning in *PrefixAgent*

and depth satisfy the equation:

$$S + L = 2n - 2, \quad (4)$$

where S and L represent the size and logic level of an n -bit prefix adder, respectively. Examples of zero and non-zero deficiency adders are shown in Figure 3.

2.2 Reasoning Models and Function Calling

Recent large language models (LLMs) such as OpenAI’s o1 [25] and o3 [26], DeepSeek-R1 [28], and QwQ [27, 29] have advanced significantly in reasoning capabilities. By incorporating intermediate “thoughts”, enclosed within `<think>...</think>` tags that represent human-like reasoning steps, LLMs are being transformed from simple autoregressive conversational models into more sophisticated cognitive reasoners, referred to as large reasoning models (LRMs). Specifically, LRMs are encouraged to generate deliberate reasoning steps before generating final responses through post-training strategies such as [45]. In addition, supervised fine-tuning (SFT) of reasoning models relies on chain-of-thought (CoT) reasoning formatted within `<think>...</think><answer>...</answer>` tags to ensure consistency and effectiveness. LLMs have implicitly learned domain-specific circuit design knowledge during pretraining and fine-tuning, and LRMs can generalize this knowledge through structured reasoning to handle complex prefix adder synthesis tasks.

2.3 Tool Integrated Reasoning

Function calling enables LLMs to delegate precise tasks to external computational tools through structured calls. In *PrefixAgent*, function calling allows the LLM to invoke external tools for accurate local modifications and evaluations, effectively bridging high-level reasoning with algorithmic precision.

This capability forms the foundation of the *tool integrated reasoning* (TIR) paradigm, where reasoning and tool usage are interleaved throughout the problem-solving process. In a typical TIR loop, the LLM performs symbolic reasoning, emits tool calls based on intermediate conclusions, receives feedback from the tool execution, and updates its internal state accordingly. This paradigm has been shown effective in various domains such as mathematical problem solving, program analysis, and circuit optimization, where purely language-based reasoning often falls short in terms of factual fidelity or structural validity.

As shown in Figure 4, *PrefixAgent* adopts this reasoning paradigm by integrating structured function calling into both backbone optimization and local refinement stages. To perform transformations on prefix structures, the LLM invokes function calls, each of which is accompanied by tool-generated feedback to guide its subsequent reasoning.

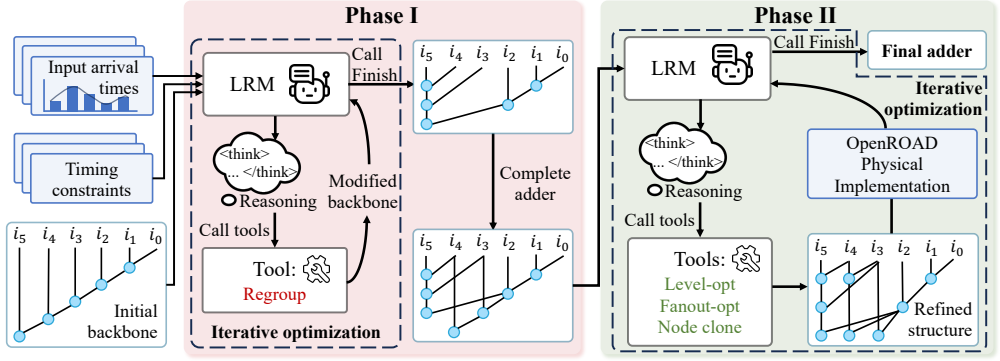


Fig. 5. PrefixAgent framework overview.

2.4 E-graphs

E-graphs [46] are data structures originally developed in the theorem proving community to compactly represent equivalence classes of expressions. An e-graph encodes a set of expressions by grouping nodes—representing variables, constants, or operators—into equivalence classes called *e-classes*. Each e-class contains one or more *e-nodes*, which are individual operator applications with pointers to their child e-classes. In this way, a small number of nodes can represent exponentially many equivalent expressions. E-graphs grow through a process known as *equality saturation* [47], where all applicable rewrites are repeatedly and non-destructively applied to the structure. This process guarantees that rewrites do not interfere with each other, avoiding the classical *phase-ordering problem* in compiler and synthesis optimization. Specifically, rewrites in e-graphs are *constructive*: when a rewrite rule (*lhs*, *rhs*) matches an expression in the e-graph, the right-hand side is simply added to the same e-class without removing the left-hand side. As a result, the e-graph grows monotonically, and all discovered equivalences are preserved throughout the saturation process. The rewrite process eventually reaches a fixed point called *saturation*, where no further rewrites can be applied.

A widely used implementation of e-graphs is the egg library [48], which has enabled extensive research in program optimization and hardware synthesis. In addition to core equality saturation, egg supports several advanced features. Notably, its *e-class analysis* framework allows users to attach semantic attributes to e-classes, enabling cost-guided optimization and program analysis.

E-graphs have been widely adopted in various domains beyond their original use in theorem proving. They have played a critical role in compiler infrastructures for program optimization, where equality saturation enables robust and phase-ordering-free transformations. More recently, e-graphs have gained traction in the EDA community. Specifically, they have been used for RTL-level datapath optimization and equivalence checking of complex arithmetic circuits [49, 50]. For example, the IMpress framework [51] employs e-graphs to explore alternative decomposition strategies for large integer multipliers in high-level synthesis (HLS). At the bit-level, E-Syn [52] applies e-graph-based rewriting for logic-level optimization.

In this work, we leverage the powerful rewriting capabilities of e-graphs in combination with multiple extraction strategies to generate optimized datasets for the backbone of prefix adders. Specifically, we utilize equality saturation and custom rewrite rules to explore the design space, and apply timing-aware extraction to identify efficient backbone candidates. Furthermore, we exploit the egg library’s explanation mechanism to extract detailed, interpretable rewrite trajectories, which are then used to construct high-quality supervision data for fine-tuning the LRM.

Table 1. Summary of tool calls in *PrefixAgent*

	Tool	Description
Phase I	regroup	Modify backbone by regroup 2 nodes.
	finish 1	Complete Phase I.
Phase II	level-opt	Reduce logic level of target node.
	fanout-opt	Reduce fanout of target node.
	node clone	Clone high fanout node.
	finish 2	Complete Phase II.

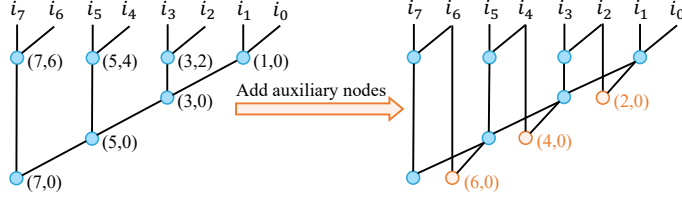


Fig. 6. Illustration of backbone (left) and completed prefix adder (right) with added auxiliary nodes.

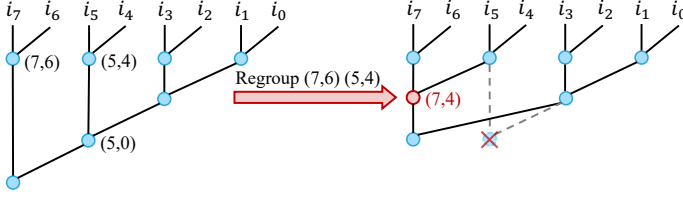


Fig. 7. Backbone regroup example.

3 Proposed PrefixAgent Framework

3.1 Overview

An overview of the *PrefixAgent* framework is shown in Figure 5. The *PrefixAgent* framework operates in two phases. In Phase I, a fine-tuned LRM iteratively refines an initial prefix adder backbone structure to achieve an optimized backbone. Upon completion of Phase I, the framework constructs a full prefix adder based on the optimized backbone, transitioning into Phase II. In Phase II, the LRM further performs local structural refinement, guided by feedback from the physical implementation to optimize timing performance. By decomposing the complex adder design task into these two phases, we simplify the complex prefix design task. The backbone refinement process directly corresponds to e-graph rewrites, enabling efficient generation of high-quality optimization data for fine-tuning the LRM. The detailed methodology for generating data utilizing e-graphs will be elucidated in Section 4.

PrefixAgent is a tool integrated reasoning style framework. Rather than directly generating a fully valid prefix adder structure, *PrefixAgent* employs function calling during reasoning, allowing the LRM to focus more on high-level decision-making, such as where and what type of optimization to perform. The actual execution and evaluation of structural modifications is handled by external tools. The LRM determines the end of each phase by explicitly invoking the corresponding completion tools. The tools employed within each phase are summarized in Table 1.

3.2 Backbone of Prefix Adder

DEFINITION 1 (BACKBONE). *Given an N -bit prefix graph, the backbone is defined as the subgraph consisting of nodes that compute the carry for the most significant bit (MSB), denoted as node $(N - 1, 0)$.*

As illustrated in the left part of Figure 6, the backbone for an 8-bit prefix adder includes nodes specifically structured for computing the MSB carry. The backbone dominates the fundamental architecture of the prefix adder, significantly influencing its overall performance and complexity. In this example, the backbone corresponds to the following grouping structure:

$$G_{7:0} = (i_0 \circ i_1) \circ (i_2 \circ i_3) \circ (i_4 \circ i_5) \circ (i_6 \circ i_7) \quad (5)$$

The backbone can be viewed as a full binary tree with N leaves, where pairs of input nodes from i_0 to i_{N-1} are recursively grouped. It is well known that such a binary tree has exactly $N - 1$ nodes. Therefore, we let S_B denote the number of backbone nodes, and $S_B = N - 1$. To derive the full prefix adder from a given backbone, each bit position must be associated with an output node. For a backbone with level L_B , there already exist L_B output nodes corresponding to the backbone level. Therefore, for the remaining $N - 1 - L_B$ bit positions that lack outputs, we insert auxiliary nodes to complete the prefix graph as shown in Figure 6. The total number of auxiliary nodes is thus $S_A = N - 1 - L_B$. This construction ensures that all intermediate nodes have non-trivial fanout, i.e., each intermediate propagate-generate signal is utilized by subsequent nodes. According to [41], this is a necessary condition for a zero-deficiency adder. By appropriately designing the backbone structure, the completed prefix adder can achieve a level L equal to the backbone level L_B , and the total size and level jointly satisfy Equation (4), corresponding to a zero-deficiency adder. In scenarios involving non-uniform input arrival times, the final level L may exceed L_B . In such cases, subsequent local refinements using the `level_opt` tool in Phase II can yield a low-deficiency adder.

Focusing on the backbone reduces the design complexity while resulting in zero- or low-deficiency adders. For an N -bit prefix adder, the full solution space grows exponentially as a product of Catalan numbers. Restricting optimization to the backbone reduces this to a single Catalan number. For example, in the 16-bit case, the design space drops from approximately 10^{48} to 10^6 .

3.3 Regroup of Backbone

To further improve the backbone structure, *PrefixAgent* employs a regrouping strategy that iteratively modifies the grouping pattern within the binary tree. A regroup operation alters the backbone by reorganizing two existing subtrees. As illustrated in Figure 7, regrouping nodes $(7, 6)$ and $(5, 4)$ introduces a new node $(7, 4)$ and removes node $(5, 0)$, effectively restructuring the backbone. In this example, the grouping structure of the backbone changes from Equation (5) to:

$$G_{7:0} = ((i_0 \circ i_1) \circ (i_2 \circ i_3)) \circ ((i_4 \circ i_5) \circ (i_6 \circ i_7)) \quad (6)$$

The detailed implementation of regroup operations, including how regroup candidates are identified within a backbone, is presented in Section 3.4. To identify valid regrouping opportunities, we impose structural constraints. Specifically, two candidate nodes a and b must satisfy the condition $a.lsb = b.msb - 1$, ensuring they can serve as valid non-trivial and trivial fanins for a new node. Candidate pairs are enumerated by traversing the current set of $N - 1$ backbone nodes, and output nodes are excluded from regrouping.

Notably, each regroup operation can be precisely interpreted as an expression-level rewrite of the MSB carry computation, for example, transforming the expression in Equation (5) into that in Equation (6). This aligns with the equivalence-preserving rewrites in e-graphs. This elegant correspondence enables us to leverage e-graph to generate high-quality backbone structures along with their corresponding regrouping trajectories, which can be used to construct effective fine-tuning data for the LRM, as described in detail in Section 4.

Algorithm 1 Backbone Optimization by LRM

Require: Bit-width N , arrival time $\mathcal{D}$, target delay T , max iteration K

Ensure: Optimized backbone $B^\star$

```
1:  $B \leftarrow \text{InitBackbone}(N)$  ▷ Initialize serial backbone
2: for  $k = 1$  to  $K$  do
3:    $C \leftarrow \text{FindCandidates}(B)$  ▷ Get regroup candidates
4:    $\text{prompt} \leftarrow \text{BuildPrompt}(B, C, \mathcal{D}, T)$ 
5:    $\text{call} \leftarrow \text{LRM}(\text{prompt})$  ▷ LRM decision
6:   if  $\text{call} == \text{finish}$  then ▷ Terminate by LRM
7:     break
8:   else
9:      $(a, b) \leftarrow \text{call}$ 
10:     $B \leftarrow \text{Regroup}(a, b)$  ▷ Apply regroup
11:  end if
12: end for
13: return  $B$  ▷ Return final backbone
14: function  $\text{FINDCANDIDATES}(B)$ 
15:    $\mathcal{P} \leftarrow \emptyset$ 
16:   for  $i = N - 1$  to  $1$  do
17:      $a \leftarrow \text{node in column } i \text{ with } \text{lsb} > 0$ 
18:      $b \leftarrow \text{node in column } a.\text{lsb} - 1 \text{ with } \text{lsb} > 0$ 
19:      $\mathcal{P}.\text{append}((a, b))$ 
20:   end for
21:   return  $\mathcal{P}$ 
22: end function
23: function  $\text{REGROUP}(a, b)$ 
24:   Add node  $(a.\text{msb}, b.\text{lsb})$ 
25:   Remove node  $(b.\text{msb}, 0)$ 
26: end function
```

3.4 PrefixAgent Phase I: Backbone

As shown in Figure 5, given the input arrival time profile and the timing constraint specified by the user, the first phase of *PrefixAgent* focuses on backbone generation. The overall optimization process is outlined in Algorithm 1. The framework first initializes a fully serial backbone structure B , which serves as the starting point for optimization (Algorithm 1). Next, regroup candidates are generated (Algorithm 1) by traversing the backbone and identifying structurally valid node pairs (a, b) that satisfy regrouping constraints (Algorithm 1). The current backbone representation, together with candidate pairs, are then embedded into a prompt (Algorithm 1) and passed to the LRM for decision-making (Algorithm 1). The LRM may respond with a function call to apply a regroup operation (Algorithm 1) or decide to terminate the optimization process (Algorithm 1). Specifically, as detailed in Algorithm 1, the regroup operation introduces a new node to replace one previous node, thereby modifying the grouping pattern within the backbone. This iterative procedure continues until either a termination is triggered by the LRM or the maximum number of iterations K is reached. The resulting backbone is then used to generate a complete prefix adder structure and passed to the next phase.

To enable the LRM to make effective optimization decisions based on both input arrival times and the current structure, we adopt a Timing-Annotated S-Expression format that annotates each backbone node with its signal arrival time. To obtain accurate arrival time annotations, we construct the backbone netlist using interleaved AOI/OAI gates and perform static timing analysis (STA). This process captures accurate signal arrival times for all nodes, including input nodes. An example Timing-Annotated S-Expression for a fully parallel 4-bit backbone is shown below:

TIMING-ANNOTATED S-EXPRESSION

```
(3,0) [arrival=0.1050]
group(
  (3,2) [arrival=0.0600] =
    group(
      input (3,3) [arrival=0.0150],
      input (2,2) [arrival=0.0250]
    ),
  (1,0) [arrival=0.0700] =
    group(
      input (1,1) [arrival=0.0350],
      input (0,0) [arrival=0.0350])
)
```

This representation leverages the inherent hierarchical nature of binary trees and captures the structural information of the backbone clearly. It also allows the LRM to analyze timing behavior along different propagation paths, thereby supporting more informed reasoning.

3.5 PrefixAgent Phase II: Local Structural Refinement

After the backbone is optimized and the prefix adder is completed by adding auxiliary nodes, timing violations may still occur. These violations are primarily caused by three factors: (1) the overall level L of the completed prefix adder may exceed the backbone level L_B ; (2) the addition of auxiliary nodes may increase fan-out at certain nodes; and (3) layout-related effects introduced during physical implementation.

To resolve these timing violations, *PrefixAgent* enters Phase II, which applies local structural refinements to improve timing. In this phase, the LRM identifies timing-critical nodes and invokes one of the three timing optimization tools. The LRM determines the appropriate optimization target and tool type, while the execution of structural modifications is delegated to external tools. As illustrated in Figure 8, the three tools include:

- (1) level-opt, which reduces the target node’s level by inserting an additional node;
- (2) fanout-opt, which inserts a node to alleviate excessive fanout at the target node;
- (3) node clone, which creates a copy of a high-fanout node to share the load.

In our framework, this structural node cloning is complementary to physically aware cloning in downstream implementation tools: cloned nodes can improve placement and routing friendliness as reported in [14], and the node-clone tool can be disabled in *PrefixAgent* if desired. After each optimization step, we invoke a prefix-graph validator to check structural legality using the $O(n^2)$ procedure described in Section 2.1; for bit-widths up to 64 bits, this runs at the millisecond scale and does not dominate the overall runtime.

To support effective reasoning during structural refinement, we introduce the *Enhanced Prefix Representation* (EPR). Preliminary experiments show that explicitly encoding node-level attributes—such as logic level, parent relationships, and fanout—significantly improves the LRM’s ability to analyze and optimize prefix graph structures. The EPR compactly captures both global properties and detailed per-node information of a prefix adder. It includes input and non-input node lists, level and connection relationships. An example EPR for a 4-bit prefix adder is shown below:

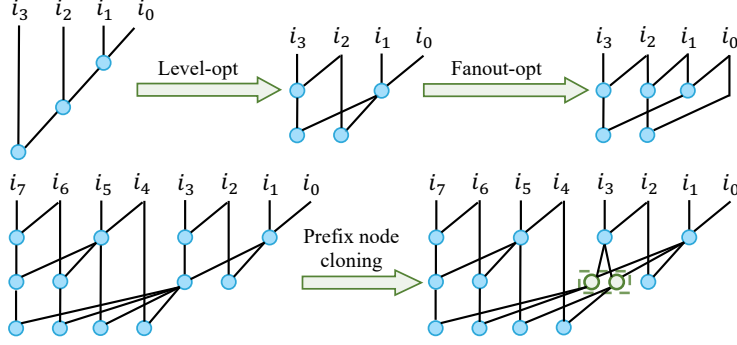


Fig. 8. Illustration of timing optimization tools: level-opt, fanout-opt and node clone.

ENHANCED PREFIX REPRESENTATION (EPR)

```

Bitwidth: 4
Non-input nodes: 3
Max level: 3
Max fanout: 1

Input nodes:
(0,0), fo_t: [], fo_nt: [(1,0)]
(1,1), fo_t: [(1,0)], fo_nt: []
(2,2), fo_t: [(2,0)], fo_nt: []
(3,3), fo_t: [(3,0)], fo_nt: []

Non-input nodes:
(3,0), lvl: 3, fi_t:(3,3), fi_nt:(2,0), fo_t: [], fo_nt: []
(2,0), lvl: 2, fi_t:(2,2), fi_nt:(1,0), fo_t: [], fo_nt: [(3,0)]
(1,0), lvl: 1, fi_t:(1,1), fi_nt:(0,0), fo_t: [], fo_nt: [(2,0)]

```

During timing refinement, critical path information is essential for guiding optimization decisions. Based on the start and end points of the critical path, we reconstruct the full sequence of nodes along the critical path and generate a structured analysis. An example for critical path analysis is as follows:

CRITICAL PATH ANALYSIS

```

Lvl 0: (0,0), [INPUT] fo_t: [], fo_nt: [(1,0)]
Lvl 1: (1,0), fi_t: (1,1), fi_nt: (0,0), fo_t: [], fo_nt: [(2,0)]
Lvl 2: (2,0), fi_t: (2,2), fi_nt: (1,0), fo_t: [], fo_nt: [(3,0)]
Lvl 3: (3,0), fi_t: (3,3), fi_nt: (2,0), fo_t: [], fo_nt: []

- Level efficiency: 2/3 (theoretical min: 2, actual: 3)

```

In each refinement iteration, the *PrefixAgent* framework constructs a prompt using current physical metrics, tool feedback, EPR, and the critical path analysis. In Phase I, the LRM operates on timing-annotated s-expressions of the MSB-carry backbone, which provides a compact tree topology and path delays. In Phase II, it relies on the EPR format to expose full prefix-graph connectivity and node-level attributes, so each phase uses the representation that best matches its optimization objective. Based on this information, the LRM performs reasoning and then either invokes an appropriate optimization tool or issues a termination call to finalize the design and prevent over-design.

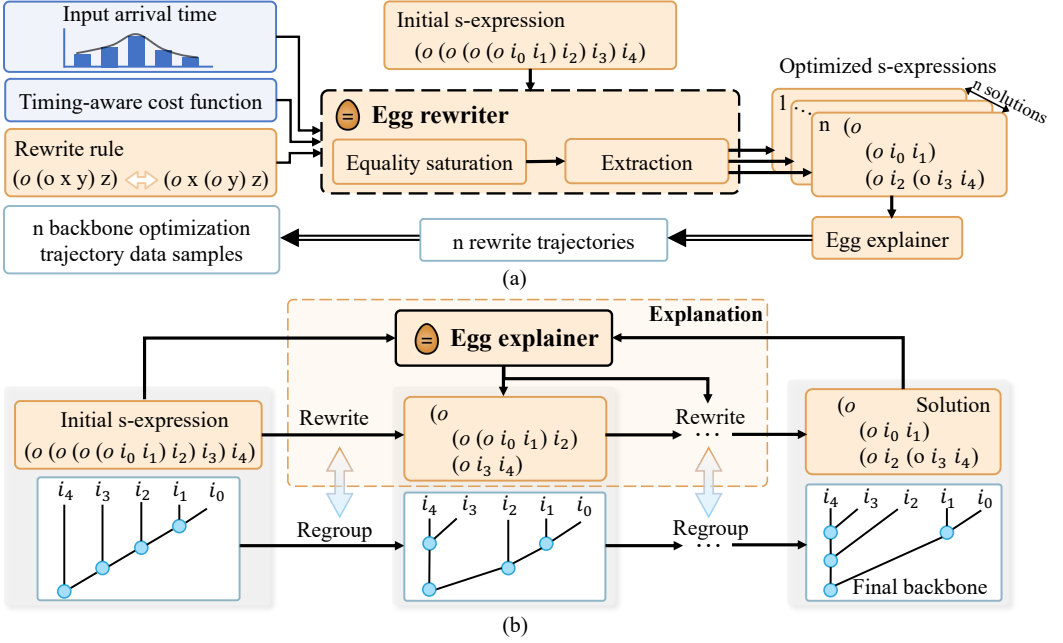


Fig. 9. E-graph based optimization trajectory dataset generation: (a) Given diverse input arrival profiles and extraction strategies, egg extracts a variety of optimized s-expressions. (b) Each optimized s-expression is explained as a rewrite trajectory using egg explainer, then forming interpretable regroup sequences for backbone optimization.

4 E-graph-based Data Generation

Optimizing prefix adder backbones under specific delay constraints and non-uniform input arrival times remains highly complex. For LRMs, this task constitutes an unseen scenario, highlighting the necessity of high-quality optimization trajectory data for effective fine-tuning. To enable effective fine-tuning of the LRM, we propose an e-graph-based data generation framework, as illustrated in Figure 9 and Figure 10. This data generation flow consists of four main components:

- (1) utilizing e-graphs to derive diverse backbone solutions tailored to distinct input arrival profiles and bandwidth configurations;
- (2) leveraging the explanation provided by egg to produce interpretable optimization rewrite (i.e., regroup function callings in *Prefix Agent* Phase I) trajectories derived from the backbone solutions;
- (3) employing commercial LRMs such as GPT-o3 [26] to generate structured `<think>...</think>` reasoning content based on the rewrite trajectories; and
- (4) integrating the *PrefixAgent* prompts with tool feedback, as well as the regroup function callings trajectories, to generate comprehensive training samples for the fine-tuning of LRM.

4.1 Backbone Representation in E-graphs

The egg library represents expressions using nested s-expressions format. An s-expression (short for symbolic expression) is a parenthesis-based notation commonly used to represent nested hierarchical structures, originally popularized in Lisp programming. In the context of e-graphs, s-expressions describe computation trees in a fully parenthesized form, where both operators and operands must be explicitly defined in a domain-specific intermediate language. To describe

prefix adder backbones, we define a domain-specific intermediate language, BackboneLang, to encode backbone structures in a structured and concise form. In BackboneLang, each input node is represented as a Leaf annotated with its input arrival time. The language supports only one single binary operator o , which denotes a grouping operation between two operands—either leaf nodes or previously grouped sub-expressions. For example, grouping inputs i_0 and i_1 is expressed as $(o\ i_0\ i_1)$, where i_0 and i_1 are Leaf nodes. In this way, the backbone structure corresponding to Equation (5) can be represented in BackboneLang as:

BACKBONELANG EXAMPLE1

```
(o
  (o
    (o
      (o i0 i1)
      (o i2 i3))
    (o i4 i5)
    (o i6 i7)))
```

4.2 Backbone Rewrites in E-graphs

We define a single associativity rewrite rule as follows:

$$(o\ (o\ x\ y)\ z) \iff (o\ x\ (o\ y\ z)) \quad (R1)$$

This associativity rewrite rule on S-expressions is semantically equivalent to a regroup operation in the prefix backbone. That is, rewriting an expression according to rule R1 corresponds exactly to restructuring the backbone by modifying the regroup of two subtrees. For example, in Figure 7, the rewrite operation transforms the expression from Equation (5) to Equation (6), resulting in an alteration of the corresponding s-expression to:

BACKBONELANG EXAMPLE2

```
(o
  (o
    (o i0 i1)
    (o i2 i3))
  (o
    (o i4 i5)
    (o i6 i7)))
```

Starting from an initial fully serial backbone expression, we employ egg’s equality saturation [47] to apply the associativity rewrite rule R1. This process compactly encodes the complete solution space of backbone structures within a single e-graph $\mathcal{G}$.

4.3 Timing-aware Cost Function for Extraction

From the generated e-graph $\mathcal{G}$, we extract multiple backbone candidates under given input arrival times and delay constraints. The bottom-up extractor in egg assumes a compositional cost model, where the cost of each enode is computed from its operator and the costs of its children. The built-in costs in egg (AST size or AST depth) are not well aligned with our backbone setting: for a fixed bit-width, the backbone node count is constant, so an AST-size cost becomes degenerate, and a pure depth cost cannot reflect bit-wise non-uniform input arrival times. We therefore employ the egg extractor with a timing-aware cost tailored to the backbone structure.

Table 2. Best critical-path delay (ns) under different extraction cost functions

	Timing-aware	AST-depth	Random
profile 1	0.4862	0.5684	0.6692
profile 2	0.4680	0.5514	0.6851

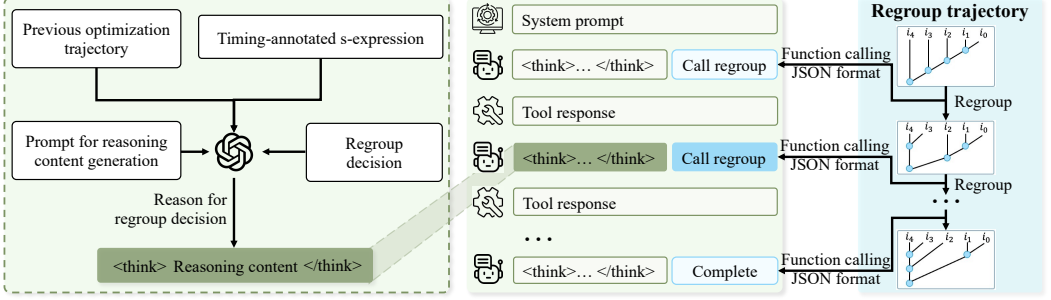


Fig. 10. Fine tuning data synthesis

Prior work [8, 16] has shown that a linear approximation provides estimates of path delays in prefix adders with high fidelity. Given that each backbone node has a fixed fanout count of one, the delay is primarily determined by the longest path in the binary tree structure. Consequently, we define the delay cost as a linear function of the path length, $y = wx + b$, where x denotes the number of nodes on the path. Accordingly, we define the cost function C to guide e-graph extraction under non-uniform input arrival times. This function recursively computes the cost of each internal node based on the arrival times of its inputs and the linear delay model:

$$C(n) = \begin{cases} t_i, & \text{if } n = \text{Leaf}(x_i); \\ \max(C(l), C(r)) + d + \lambda, & \text{if } n = \text{Op}(l, r). \end{cases} \quad (7)$$

Here, t_i denotes the input arrival time at Leaf x_i , d is a constant node delay, and λ is a timing margin used to model physical-implementation uncertainty.

To evaluate the impact of the cost design, we compare our timing-aware cost against two baselines on a 16-bit backbone. For two representative non-uniform arrival profiles, we extract 100 solutions and record the best critical-path delay under three cost functions: the proposed timing-aware cost, an AST-depth cost, and a random baseline. As summarized in Table 2, the timing-aware cost consistently yields backbones with smaller critical-path delays than both baselines, which indicates that it is a suitable and effective extractor-compatible model in our setting.

4.4 Optimization Trajectory Generation

The cost function satisfies decomposability and monotonicity, two essential properties required by egg’s optimal extraction algorithm. We therefore adopt the default bottom-up extractor provided by egg to retrieve the optimal backbone solution. The optimal solution corresponds to the tightest timing constraint scenario. To emulate more general area–delay trade-off conditions, we additionally generate suboptimal solutions by introducing controlled perturbations during the extraction process. This modification allows the extractor to explore a broader set of feasible backbone structures that reflect a wider range of relaxed timing constraints.

Table 3. Trajectory and profile statistics (#Profile = input arrival profiles, #Traj. = trajectories, Length = trajectory length, Level = backbone level, Target delay = target delay range, Steps = total regroup steps).

	#Profile	#Traj.	Length	Level	Target delay (ns)	Steps
8-bit	23	135	3–6	3–6	0.4–0.7	608
12-bit	21	132	4–8	4–7	0.5–0.8	818
16-bit	20	128	12–18	4–9	0.5–0.8	1920
24-bit	18	120	17–26	5–12	0.6–1.0	2580
32-bit	16	110	24–37	5–15	0.7–1.2	3315
48-bit	16	100	41–61	6–18	0.8–1.4	4984
64-bit	16	90	49–84	6–25	1.0–1.7	5760

As illustrated in Figure 9 (a), we first generate numerous backbone solutions (i.e., optimized s-expressions) using egg’s extraction mechanism. Subsequently, for each of these solutions, the egg explainer is utilized to produce an explanation trajectory in the form of a rewrite sequence, describing how the backbone structure is transformed from the initial serial expression. As illustrated in Figure 9 (b), each rewrite of the s-expression within a trajectory directly corresponds to a regroup operation on the backbone. Consequently, these sequences serve as interpretable optimization trajectories, explicitly capturing the chain of regroup operations applied to reach a given backbone structure. Among all backbone solutions encoded in the e-graph, we select training samples that yield minimal increases in the final prefix adder level L relative to the backbone level L_B . Such samples are more likely to correspond to zero-deficiency or low-deficiency adders. A summary of the collected trajectory statistics across bit-widths is given in Table 3. The arrival profiles span seven families (linear, convex, concave, MSB-first, LSB-first, trapezoidal, and random), with multiple instances per family. Equality saturation encodes the full single-Catalan grouping space for each bit-width, and timing-aware extraction samples a structurally diverse subset, as reflected by the backbone-level ranges in Table 3. We then perform an approximately 8:1:1 split, stratified by profile family, at the level of individual profile instances, so that a profile instance used for evaluation never appears in training, even when other instances of the same family do.

4.5 Data Synthesis and Fine Tuning

With the optimization trajectories established, synthesizing complete fine-tuning samples for the LRM necessitates augmenting each regroup decision in the trajectory with explicit Chain-of-Thought (CoT) reasoning, which is annotated within `<think>...</think>` tags. As illustrated on the left side of Figure 10, we synthesize the CoT reasoning content for each regroup decision. This is achieved by prompting a powerful teacher LRM, such as GPT-o3 [26], with the historical optimization trajectory and the specific regroup choice for the current step, instructing it to generate a plausible reasoning chain. To assemble the final fine-tuning data for an agent’s response, the regroup decision is formatted as a JSON-style function call. This call is then concatenated with the corresponding synthesized reasoning content, which is encapsulated within `<think>...</think>` tags. Example data sample for fine-tuning is shown as below:

BACKBONE OPTIMIZATION TRAJECTORY DATA EXAMPLE

```
System prompt: \n
You are an agent for backbone optimization ..... + [input arrival time profile] + [target delay constraint]
↔ + [initial timing-annotated s-expression] + [instructions]

PrefixAgent: \n
```

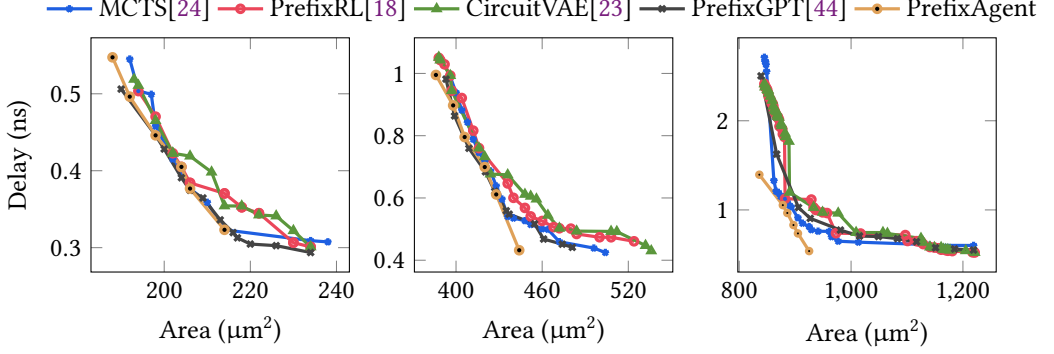


Fig. 11. Pareto-frontiers under uniform arrival time. From left to right: 16-bit; 32-bit; 64-bit.

```

<think>Now we need to optimize the adder backbone to 2ns ... First we look at the input arrival profile ...
↳ It is the most critical part, we need to add parallelism at ... So we need to regroup node (2,2) and
↳ (1,0) </think>regroup tool calls \\

Tool response: \\
Regroup success; Regroup success + [logic level improvement] + [delay improvement] + [current
↳ timing-annotated s-expression]\\

<think>... We still need to speed up at ... </think>regroup tool calls

Tool response: Regroup success + [logic level improvement] + [delay improvement] + [current
↳ timing-annotated s-expression]\\

<think> ... Based on this analysis, I decide to regroup (15,14) and (13,10)</think>regroup tool calls \\

....

Tool response: Regroup success + [logic level improvement] + [delay improvement] + [current s-expression]\\

<think>The current critical path delay already satisfies the timing constraint ... So we can complete
↳ now.</think>Call complete

```

Highly repetitive CoT templates can limit the model’s ability to generalize. In such cases, the model tends to overfit to surface-level textual profiles rather than attend to structural features. This behavior results in reduced sensitivity to function-call accuracy, despite achieving low training loss that does not reflect actual reasoning fidelity. To jointly supervise reasoning generation and structural function calling decision accuracy, we define a composite training objective:

$$L_{\text{total}} = L_{\text{CoT}} + \lambda \cdot L_{\text{function_call}}$$

Here, L_{CoT} supervises the generation of coherent chain-of-thought reasoning, while $L_{\text{function_call}}$ promotes accurate invocation of regroup via function calls.

5 Experimental Results

5.1 Setup

Both fine-tuning and inference are conducted on a single node with 8 NVIDIA H100 (80 GB) GPUs and 2 TB of host memory. The *PrefixAgent* framework adopts the QwQ-32B [29] model as the LRM, which exhibits strong reasoning and function-calling capabilities. We fine-tune it using LoRA with a per-GPU batch size of 2 for three epochs, reaching a peak GPU memory

Table 4. Comparison of non-uniform arrival time profiles

Input arrival profile	Method	16-bit		32-bit		54-bit		64-bit	
		Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)
LSB-first	DP[10]	236	0.6089	461	0.8109	761	0.9540	1130	1.5307
	MCTS[24]	217	0.5810	438	0.8012	707	0.9531	1045	1.5123
	PrefixRL[18]	222	0.5928	445	0.8035	726	0.9486	1078	1.4984
	PrefixGPT[44]	215	0.6014	438	0.8012	712	0.9489	1051	1.5105
	CircuitVAE[23]	219	0.5948	442	0.7946	735	0.9584	1074	1.5078
	PrefixAgent	217	0.5810	426	0.7980	697	0.9461	938	1.4882
Random	DP[10]	220	0.5740	490	0.8607	739	1.1249	1134	1.5610
	MCTS[24]	207	0.5567	463	0.8509	668	0.9963	1075	1.5591
	PrefixRL[18]	212	0.5543	463	0.8509	679	1.0986	1104	1.5643
	PrefixGPT[44]	207	0.5567	449	0.8560	664	1.0587	1094	1.4915
	CircuitVAE[23]	210	0.5478	472	0.8489	672	0.9960	1097	1.5630
	PrefixAgent	205	0.5450	440	0.8373	653	0.9980	953	1.5579

of about 60 GB and completing in roughly 12 hours. We collect CoT content using the GPT-o3 [26] model (API snapshot o3 in a high-entropy, tool-free configuration: temperature = 0.8, top_p = 0.95, max_output_tokens = 16384, seed = 7, tools = [], and tool_choice = "none"). We use NanGate45 Open Cell Library [53] and perform logic synthesis with Yosys [54], and physical implementation with OpenROAD [55]. Functional correctness of all generated adders is verified by equivalence checking using ABC [56]. Our baselines include:

DP [10]: A dynamic-programming method that minimizes area under non-uniform arrival times.

PrefixRL [18]: A RL-based optimization approach, reimplemented following the methodology described in [18].

MCTS [24]: A MCTS-based method, using the open-source codes from [24].

CircuitVAE [23]: A generative approach based on VAE. We re-implement the model following [23], incorporating the optimization strategy proposed in [57]. Under each configuration, randomly sample 1000 initial adders from PrefixRL solutions as initial structures.

PrefixGPT [44]: A transformer-based prefix-adder generation method, extended to 64-bit in our flow.

We also attempted to reproduce PrefixLLM [37]; however, it consistently generated ripple-carry adders or replicated the given initial structures, and failed to produce valid new prefix structures for cases beyond 16 bits. Therefore, we exclude it from comparative evaluation. Detailed discussion of PrefixLLM can be found in Section 5.6.

We evaluate all methods under both uniform and non-uniform arrival time profile conditions. We set various target delays for area-delay trade-off scenarios and conduct physical implementation with the target delays. All approaches are evaluated under the same delay target to ensure a fair comparison. For RL, MCTS, and VAE baselines, we adopt a unified cost formulation to enable consistent timing–area trade-off evaluation. Specifically, the cost formulation penalizes timing violations and rewards area reduction. We further evaluate *PrefixAgent* under a commercial physical design flow.

5.2 Comparing with Baselines

Under the uniform arrival time setting, we evaluate each method across six target delay values per bit-width, representing a range of area–delay trade-off scenarios. For each target delay, *PrefixAgent* generates a single optimized adder structure. In contrast, each baseline (MCTS, RL, VAE, and PrefixGPT) is executed with 5000 samples per delay target, yielding 30,000 designs per method. The resulting Pareto fronts are illustrated in Figure 11. Across all tested bit-widths, *PrefixAgent* consistently achieves high-quality solutions. Notably, in the 64-bit case, it Pareto-dominates all baseline approaches. To show how these trade-offs are realized at the structural level, Figure 12

Table 5. Ablation study on framework components, prefix-graph encoding, and training-data scaling. The full PrefixAgent (bottom row) uses the EPR encoding and the complete training set. All runs use the LSB-first profile under a common delay target.

Method	16-bit		32-bit		64-bit	
	Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)
Phase I only	198	0.6512	407	0.8363	890	1.5215
Phase II w/ grid encoding	223	0.5987	445	0.8106	1046	1.5300
Phase II w/ adjacency-list encoding	227	0.6045	457	0.7940	1067	1.5482
Phase II w/o critical-path prompts	217	0.5810	439	0.7683	960	1.4958
w/o CoT reasoning	245	0.6792	516	0.8245	1176	1.5520
PrefixAgent w/o fine-tuning	225	0.5867	464	0.8106	1106	1.5015
PrefixAgent (1/4 data)	221	0.5914	448	0.8134	1064	1.5150
PrefixAgent (1/2 data)	217	0.5810	437	0.7945	951	1.5069
Phase I + Phase II (PrefixAgent)	217	0.5810	426	0.7980	938	1.4882

presents three representative 16-bit designs sampled along the Pareto frontier. The delay-optimized design adopts more prefix nodes and a shallower, more parallel topology to reduce the logic level. The area-optimized design uses fewer nodes and a more serial structure to save area, and the balanced design lies between them. This illustrates how *PrefixAgent* traverses the area–delay space through interpretable changes in prefix topology.

For the non-uniform arrival time scenario, we evaluate two distinct arrival time profiles: (i) *LSB-first*, where the least significant half of the input bits arrive earlier, and (ii) *Random*, where arrival times are randomly assigned across all bits. For each arrival profile and bit-width configuration, we set a fixed target delay. The specific LSB-first and Random profiles used here are taken from the held-out test partition (Section 4.4); no instance of either appears in the training set. Each learning-based baseline samples 5000 candidate designs under the given delay constraint, while the dynamic-programming baseline is deterministic and yields a single solution per configuration. Among them, we select the adders with the smallest area that meet the timing constraints, allowing at most 1% timing violation. The comparison results are summarized in Table 4.¹ As shown, *PrefixAgent* achieves the lowest area in seven of the eight configurations; the only exception is the 16-bit LSB-first case, where PrefixGPT attains a marginally smaller area. Notably, under the 64-bit configuration, it achieves up to 11.3% area reduction compared to the best-performing baseline. These results indicate that PrefixGPT remains competitive at smaller bit-widths, while the advantage of *PrefixAgent* becomes more pronounced at larger bit-widths and across the broader set of evaluated scenarios. In addition, the 54-bit configuration does not appear in the fine-tuning dataset (Table 3); *PrefixAgent* nonetheless attains the smallest area under both profiles, providing evidence of transfer to this unseen bit-width. Since 54 bits is the only unseen bit-width evaluated and both arrival-profile families appear in the training distribution, these results do not establish broader generalization to arbitrary bit-widths or unseen profile families. Example structures generated by *PrefixAgent* are shown in Figure 13.

We report the average runtime of PrefixAgent and the MCTS, RL, and VAE baselines in Figure 14. Compared to baseline approaches that rely on extensive search within a large design space, *PrefixAgent* performs targeted optimizations guided by LRM decisions. As a result, it achieves over 10× speedup across all evaluated configurations. The significant runtime advantage of *PrefixAgent* stems from its ability to generalize across design cases without retraining. Baseline methods such as RL, MCTS, and VAE require retraining for each new configuration, including variations in bit-width,

¹At smaller bit-widths, different methods can converge to the same prefix structure and therefore report identical post-synthesis area and delay.

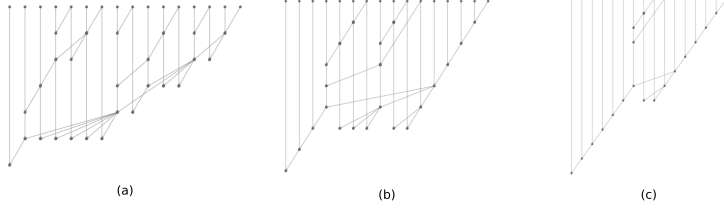


Fig. 12. Representative 16-bit prefix structures along the Pareto frontier: (a) delay-optimized; (b) balanced; (c) area-optimized.

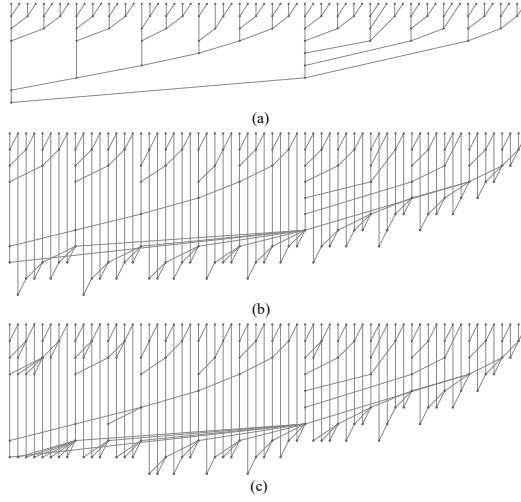


Fig. 13. Example structures generated by *PrefixAgent*: (a) optimized backbone; (b) completed adder after Phase I; (c) final optimized adder after Phase II.

Table 6. Prefix-graph reasoning accuracy (%) under different representations. Each probe consists of 100 questions over randomly sampled prefix graphs.

Probe	EPR	Grid	Adjacency-list
fanout_cone_size	94	59	36
reachable	96	63	47
fanin_cone_size	97	58	33

input arrival time profiles, and target delay constraints. In contrast, *PrefixAgent* is trained once on a carefully curated dataset covering a broad distribution of design scenarios. During inference, the LRM leverages its reasoning capabilities and the structured decomposition of the complex task into a backbone optimization task. This one-time training paradigm, combined with problem space reduction via backbone abstraction, substantially reduces the per-case optimization cost and enhances scalability.

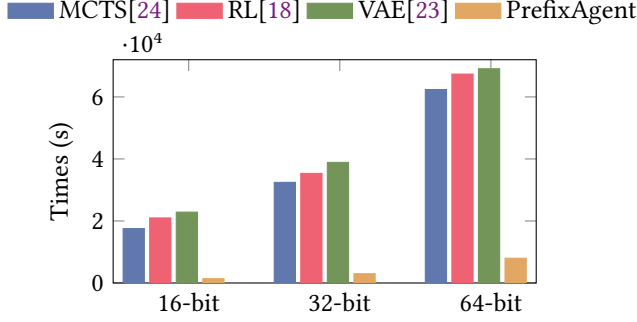


Fig. 14. Average runtime comparison.

Table 7. Ablation study on CoT reasoning

	Model	Area (μm^2)	Delay (ns)	#Tool Calls
16-bit	GPT-o3	217	0.5810	31
	Qwen3	217	0.5976	30
32-bit	GPT-o3	426	0.7980	51
	Qwen3	426	0.7980	51
64-bit	GPT-o3	938	1.4882	73
	Qwen3	954	1.4957	76

5.3 Ablation Studies

We conduct five ablation studies: (i) the contribution of framework components, namely Phase II refinement and the critical-path prompts; (ii) the Phase II prefix-graph encoding, comparing EPR against grid and adjacency-list encodings, for which we additionally report a reasoning-probe analysis; (iii) the effect of CoT reasoning against pure tool-call supervision; (iv) the effect of training-data scale, from no fine-tuning to the full training set; and (v) robustness to the choice of LRM for CoT collection. Items (i)–(iv) are reported in Table 5, the supporting reasoning-probe analysis in Table 6, and item (v) in Table 7. All ablations use the LSB-first arrival profile. Except for the Phase I-only and w/o CoT reasoning settings, the realized delays in Table 5 are close, so the remaining comparisons are read on area.

As shown in Table 5, in Phase I, the complete adders from the backbone may violate timing due to increased logic depth or fanout introduced by the auxiliary nodes. Phase II performs timing-oriented refinements and can recover timing at the cost of area. When we replace EPR with the grid or adjacency-list encoding, the resulting area degrades at every bit-width, indicating that global topology and node-level metadata (levels, fanout, parent/child links) help the LRM make better local edits. To explain EPR’s advantage, we evaluate how well the LRM reasons about a prefix graph under each representation. We construct three probes over randomly sampled prefix graphs, 100 questions each: fanout-cone size, node reachability, and node fanin-cone size. As reported in Table 6, EPR attains at least 94% accuracy on all three, while grid and adjacency-list fall to 58–63% and 33–47%. The LRM therefore recovers global structure far more reliably from EPR’s explicit per-node attributes than from raw adjacency or grid positions alone, which accounts for the area gap in Table 5. Similarly, omitting the critical-path prompts increases area at 32-bit and 64-bit, showing that highlighting the timing-critical region is beneficial.

Table 8. Comparison on commercial physical design flow

Input arrival profile	Method	16-bit		32-bit		64-bit	
		Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	Delay (ns)
LSB-first	CST	371.98	0.9995	728.38	1.0561	1639.09	1.7425
	PrefixAgent	349.19	0.9998	706.01	1.0490	1574.76	1.7453
Random	CST	351.99	0.8781	697.37	1.2027	1767.82	1.5881
	PrefixAgent	333.75	0.8729	663.82	1.1944	1702.25	1.5863

To assess the contribution of CoT reasoning, we evaluate a non-reasoning variant of the LRM. While the full PrefixAgent uses QwQ-32B model, the *w/o CoT reasoning* setting in Table 5 replaces it with its non-reasoning counterpart (Qwen2.5-32B), trained only with tool-call supervision and without CoT content. This variant shows noticeably degraded QoR, confirming that CoT reasoning provides essential guidance beyond pure tool-call signals.

To assess the effect of training-data scale, we compare *PrefixAgent* without fine-tuning against fine-tuning on 1/4 and 1/2 of the trajectories and on the full set, with the subsets sampled by trajectory count. QoR improves consistently as more training data is used, while the marginal gain diminishes as the dataset grows.

Finally, to assess robustness to the choice of LRM for CoT collection, we introduce an open-model variant in Table 7. Besides GPT-o3, we use the open-source Qwen3-235B-A22B-Thinking-2507 model to generate CoT content, keeping the prompt template and hyperparameters identical. We obtain area-delay results and tool-call counts that closely match those of GPT-o3 across all evaluated bit-widths. This indicates that PrefixAgent can operate effectively with different strong LRMs for CoT generation, supporting the robustness of the overall framework. Because the CoT teacher is the only proprietary component and can be replaced by the open-source Qwen3 model with comparable results, the pipeline is reproducible without proprietary models.

5.4 Evaluation under a Commercial Flow and Implementation in Systolic Arrays

To evaluate *PrefixAgent* under a different technology and implementation flow, we conduct experiments using a commercial physical implementation toolchain with a 32 nm technology. We compare our method against adders synthesized by commercial synthesis tools (CST) using $y = a + b$ style Verilog under non-uniform input arrival time profiles. As shown in Table 8, *PrefixAgent* generates adders with a smaller area compared to CST solutions.

To assess benefits in an accelerator setting, we implement a 256-PE weight-stationary systolic array, where each PE uses two PrefixAgent-generated prefix adders: the final carry-propagate adder after the multiplier’s compressor tree and the accumulator adder. As summarized in Table 9, replacing the default synthesis-tool adders with PrefixAgent-generated adders reduces array-level area and also improves critical-path delay under both the OpenROAD flow and a commercial flow, for both 8-bit and 16-bit configurations.

Table 9. Systolic array comparison under OpenROAD and commercial flows. (Area unit: μm^2 . Delay unit: ns).

Flow	Bitwidth	Method	Area	Delay
OpenROAD	8-bit	Default	158069	0.9361
		Ours	156704	0.8716
	16-bit	Default	465961	1.4640
		Ours	452625	1.3160
Commercial	8-bit	Default	401701	1.7120
		Ours	391865	1.6987
	16-bit	Default	1157562	2.3750
		Ours	1138543	2.3648

Table 10. Comparison with a physically-aware Ling-node baseline [58] under the LSB-first non-uniform arrival profile (NanGate45, OpenROAD flow). Area in μm^2 , delay in ns.

Method	16-bit		32-bit		64-bit	
	Area	Delay	Area	Delay	Area	Delay
Ling baseline [58]	247	0.6210	487	0.8095	1106	1.5184
PrefixAgent	217	0.5810	426	0.7980	938	1.4882
PrefixAgent + Ling	226	0.5848	444	0.8033	971	1.4950

5.5 Comparison with Ling-Node and Physically-Aware Synthesis

To address physically-aware prefix co-optimization and Ling-node designs, we compare *PrefixAgent* with a method [58] that is both physically-aware and natively Ling-based, re-implemented in our flow under the LSB-first profile for a consistent comparison. As shown in Table 10, *PrefixAgent* achieves smaller area than this baseline at every bit-width, up to 15.2% at 64 bits, at equal or lower delay. Ling-node strategies are orthogonal to the prefix topology: a Ling adder shares the same prefix-processing network and only reformulates the carry recurrence, so any structure *PrefixAgent* produces can be realized as a Ling adder. Applying Ling’s recurrence to our structures (*PrefixAgent*+Ling) increases area by about 4% while leaving delay essentially unchanged, and *PrefixAgent*+Ling still remains smaller than the baseline, indicating that the prefix structure rather than the Ling reformulation drives the improvement.

5.6 Discussion

In previous LLM-based methods, such as PrefixLLM [37], the language model is tasked with generating the full prefix adder structure directly. However, constructing a valid prefix adder graph requires maintaining intricate structural invariants, such as tree consistency, fan-in/fan-out correctness, and legality of prefix operations. The model is expected to perform analysis, optimization, and validation simultaneously on the full graph, while the complexity of the task exceeds the LLM’s direct reasoning capabilities. In contrast, *PrefixAgent* decomposes the task into two modular phases: backbone generation and local refinement, allowing the LRM to focus its reasoning on manageable subproblems. Equipped with domain-specific knowledge from pre-training and fine-tuning, the LRM benefits from the rich structural context provided by *PrefixAgent*, enabling it to generalize its learned knowledge to novel designs. Through this framework, the LRM is able to make high-level design decisions and invoke external tools to execute complex graph transformations, rather than attempting to manipulate the full structure directly. Moreover,

the use of high-quality optimization trajectories generated via e-graph rewrites enables effective supervised fine-tuning, further aligning the LRM’s reasoning process with domain-specific logic optimization profiles. Search-based baseline methods require retraining for each new scenario and do not scale well with bitwidth due to the exponential design space. As reported in the PrefixRL paper [18], optimizing a 64-bit adder requires over 5×10^5 training steps, taking several weeks for one new scenario. Moreover, we observe that these methods frequently explore high-deficiency structures that lead to excessive area consumption. *PrefixAgent* provides a flexible and efficient alternative across the evaluated design scenarios.

Empirically, we occasionally encounter pathological instances where, under very aggressive timing targets, further Phase II refinements yield only marginal delay improvement at the cost of substantial area growth. In such regimes, *PrefixAgent* will backtrack to Phase I again to propose a new backbone.

6 Conclusion

We propose *PrefixAgent*, a framework for prefix adder optimization. The design process is decomposed into two phases—backbone generation and local refinement. To guide the LRM, we leverage high-quality optimization traces generated through e-graph rewrites. Experimental results show that *PrefixAgent* achieves smaller areas than the evaluated baselines in nearly all tested configurations, with the advantage growing at larger bit-widths.

Moreover, we introduce a novel paradigm for generating high-quality supervision traces, which remains valuable as LRM capabilities continue to advance. Given recent e-graph-based optimizers for multipliers [59], the same data-collection methodology can be reused to learn multiplier optimization policies. In particular, compressor-tree templates (e.g., Wallace/Dadda trees) can be treated as multiplier backbones, and equivalence-preserving compressor rewrites can be used to generate trajectories for LRM fine-tuning. Likewise, e-graph frameworks for general datapath RTL/netlist optimization [50, 52] indicate that this supervision paradigm can extend beyond prefix adders to broader datapath optimization tasks.

References

- [1] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.
- [2] J. Sklansky, “Conditional-sum addition logic,” *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.
- [3] R. P. Brent and H. T. Kung, “A regular layout for parallel adders,” *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.
- [4] T. Han and D. A. Carlson, “Fast area-efficient vlsi adders,” in *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, 1987, pp. 49–56.
- [5] J. Fishburn, “A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between,” in *Proc. DAC*, 1990, pp. 361–364.
- [6] R. Zimmermann, “Non-heuristic optimization and synthesis of parallel-prefix adders,” in *International Workshop on Logic and Architecture Synthesis*, 1996.
- [7] H. Zhu, C.-K. Cheng, and R. Graham, “Constructing zero-deficiency parallel prefix adder of minimum depth,” in *Proc. ASPDAC*, 2005.
- [8] D. Zuo, J. Zhu, C. Li, and Y. Ma, “Ufo-mac: A unified framework for optimization of high-performance multipliers and multiply-accumulators,” in *Proc. ICCAD*, 2024.
- [9] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, “An algorithmic approach for generic parallel adders,” in *Proc. ICCAD*, 2003, pp. 734–740.
- [10] T. Matsunaga and Y. Matsunaga, “Area minimization algorithm for parallel prefix adders under bitwise delay constraints,” in *Proc. GLSVLSI*, 2007.
- [11] S. Lin, B. Jiang, W. Sheng, and E. Young, “Size-optimized depth-constrained large parallel prefix circuits,” in *Proc. DAC*, 2024.

- [12] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," in *Proc. DAC*, 2013, pp. 1–8.
- [13] —, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," *IEEE TCAD*, vol. 33, no. 10, pp. 1517–1530, 2014.
- [14] —, "Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs," *IEEE TCAD*, vol. 35, no. 5, pp. 820–831, 2016.
- [15] S. Roy, Y. Ma, J. Miao, and B. Yu, "A learning bridge from architectural synthesis to physical design for exploring power efficient high-performance adders," in *Proc. ISLPED*, 2017.
- [16] Y. Ma, S. Roy, J. Miao, J. Chen, and B. Yu, "Cross-layer optimization for high speed adders: A pareto driven machine learning approach," *IEEE TCAD*, vol. 38, no. 12, pp. 2298–2311, 2019.
- [17] H. Geng, Y. Ma, Q. Xu, J. Miao, S. Roy, and B. Yu, "High-speed adder design space exploration via graph neural processes," *IEEE TCAD*, vol. 41, no. 8, pp. 2657–2670, 2022.
- [18] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, "Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning," in *Proc. DAC*, 2021, pp. 853–858.
- [19] D. Zuo, Y. Ouyang, and Y. Ma, "RI-mul: Multiplier design optimization with deep reinforcement learning," in *Proc. DAC*, 2023, pp. 1–6.
- [20] D. Zuo, J. Zhu, Y. Ouyang, and Y. Ma, "RI-mul 2.0: Multiplier design optimization with parallel deep reinforcement learning and space reduction," *ACM TODAES*, 2025.
- [21] Z. Wang, J. Wang, D. Zuo, J. Yunjie, X. Xia, Y. Ma, J. Hao, M. Yuan, Y. Zhang, and F. Wu, "A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design," in *Proc. ICML*, 2024.
- [22] Z. Wang, J. Wang, X. Xia, D. Zuo, L. Chen, Y. Ma, J. HAO, M. Yuan, and F. Wu, "Computing circuits optimization via model-based circuit genetic evolution," in *Proc. ICLR*, 2025.
- [23] J. Song *et al.*, "Circuitvae: Efficient and scalable latent circuit optimization," in *Proc. DAC*, 2024.
- [24] Y. Lai, J. Liu, D. Z. Pan, and P. Luo, "Scalable and effective arithmetic tree generation for adder and multiplier designs," in *Proc. NIPS*, vol. 37, 2024, pp. 139 151–139 178.
- [25] OpenAI, "Learning to reason with llms," <https://openai.com/index/learning-to-reason-with-llms/>.
- [26] —, "Openai o3," <https://openai.com/index/introducing-o3-and-o4-mini/>.
- [27] Alibaba, "Qwq-max-preview," <https://qwenlm.github.io/blog/qwq-max-preview/>.
- [28] DeepSeek-AI, D. Guo *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: <https://arxiv.org/abs/2501.12948>
- [29] Alibaba, "Qwq-32b: Embracing the power of reinforcement learning," <https://qwenlm.github.io/blog/qwq-32b/>.
- [30] Y. Lai, S. Lee, G. Chen, S. Poddar, M. Hu, D. Z. Pan, and P. Luo, "Analogcoder: Analog circuit design via training-free code generation," in *Proc. AAAI*, 2024.
- [31] C.-C. Chang *et al.*, "Lamagic: language-model-based topology generation for analog integrated circuits," in *Proc. ICML*, 2024.
- [32] J. Gao, W. Cao, J. Yang, and X. Zhang, "Analoggenie: A generative engine for automatic discovery of analog circuit topologies," in *Proc. ICLR*, 2025.
- [33] K. Chang *et al.*, "Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework," in *Proc. DAC*, 2024.
- [34] N. Chen, Y. Li, J. Tang, and J. Li, "Graphwiz: An instruction-following language model for graph problems," in *Proc. KDD*, 2024.
- [35] X. Dai, H. Qu, Y. Shen, B. Zhang, Q. Wen, W. Fan, D. Li, J. Tang, and C. Shan, "How do large language models understand graph patterns? a benchmark for graph pattern comprehension," in *Proc. ICLR*, 2025.
- [36] J. Tang, Q. Zhang, Y. Li, N. Chen, and J. Li, "Grapharena: Evaluating and exploring large language models on graph computation," in *Proc. ICLR*, 2025.
- [37] W. Xiao, V. S. C. Putrevu, R. V. Hemadri, S. Garg, and R. Karri, "Prefixllm: Llm-aided prefix circuit design," 2024. [Online]. Available: <https://arxiv.org/abs/2412.02594>
- [38] Y. Ye, Z. Huang, Y. Xiao, E. Chern, S. Xia, and P. Liu, "Limo: Less is more for reasoning," 2025. [Online]. Available: <https://arxiv.org/abs/2502.03387>
- [39] X. Li, H. Zou, and P. Liu, "Limr: Less is more for rl scaling," 2025. [Online]. Available: <https://arxiv.org/abs/2502.11886>
- [40] N. Muennighoff *et al.*, "s1: Simple test-time scaling," 2025. [Online]. Available: <https://arxiv.org/abs/2501.19393>
- [41] M. Snir, "Depth-size trade-offs for parallel prefix computation," *Journal of Algorithms*, vol. 7, no. 2, pp. 185–201, 1986.
- [42] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, M. Huang, N. Duan, and W. Chen, "Tora: A tool-integrated reasoning agent for mathematical problem solving," *arXiv preprint arXiv:2309.17452*, 2023.
- [43] X. Li, H. Zou, and P. Liu, "Torl: Scaling tool-integrated rl," *arXiv preprint arXiv:2503.23383*, 2025.
- [44] R. Ding, X. Ning, U. Schlichtmann, and W. Qian, "Prefixgpt: Prefix adder optimization by a generative pre-trained transformer," in *Proc. AAAI*, 2026, pp. 20 808–20 815.

- [45] Z. Shao, P. Wang, Q. Zhu, R. Xu, J.-M. Song, M. Zhang, Y. K. Li, Y. Wu, and D. Guo, “Deepseekmath: Pushing the Limits of Mathematical Reasoning in Open Language Models,” *arXiv.org*, vol. abs/2402.03300, 2024.
- [46] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford University, 1980.
- [47] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: a new approach to optimization,” in *Proc. POPL*, 2009.
- [48] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, “egg: Fast and extensible equality saturation,” *Proc. ACM Program. Lang.*, 2021.
- [49] S. Coward, E. Morini, B. Tan, T. Drane, and G. Constantinides, “Datapath verification via word-level e-graph rewriting,” 2023.
- [50] S. Coward, T. Drane, and G. A. Constantinides, “Constraint-aware e-graph rewriting for hardware performance optimization,” *IEEE TCAD*, vol. 44, no. 4, pp. 1366–1379, 2025.
- [51] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, “Impress: Large integer multiplication expression rewriting for fpga hls,” in *Proc. FCCM*, 2022, pp. 1–10.
- [52] C. Chen, G. Hu, D. Zuo, C. Yu, Y. Ma, and H. Zhang, “E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis,” in *Proc. DAC*, 2024.
- [53] Nangate Inc., “Open Cell Library v2008_10 SP1,” 2008. [Online]. Available: <http://www.nangate.com/openlibrary/>
- [54] C. Wolf, “Yosys open synthesis suite,” <https://yosyshq.net/yosys/>.
- [55] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, “Toward an open-source digital flow: First learnings from the openroad project,” in *Proc. DAC*, 2019, pp. 1–4.
- [56] Berkeley Logic Synthesis and Verification Group, “ABC: A System for Sequential Synthesis and Verification,” <http://www.eecs.berkeley.edu/~alanmi/abc/>.
- [57] A. Tripp *et al.*, “Sample-efficient optimization in the latent space of deep generative models via weighted retraining,” in *Proc. NIPS*, 2020.
- [58] Y. Zhu, J. Liu, H. Zhu, and C.-K. Cheng, “Timing-power optimization for mixed-radix ring adders by integer linear programming,” in *2008 Asia and South Pacific Design Automation Conference*, 2008, pp. 131–137.
- [59] A. Wanna, S. Coward, T. Drane, G. A. Constantinides, and M. D. Ercegovac, “Multiplier optimization via e-graph rewriting,” in *Proc. Asilomar*, 2023, pp. 1528–1533.