BUFFALO: PPA-Configurable, LLM-based Buffer Tree Generation via Group Relative Policy Optimization

Hao-Hsiang Hsiao[1], Yi-Chen Lu[2], Sung Kyu Lim[1], and Haoxing Ren[2]
[1]School of ECE, Georgia Institute of Technology, Atlanta, GA
[2]NVIDIA Research;
{thsiao, limsk}@gatech.edu; {yilu, haoxingr}@nvidia.com;

***Abstract*—Buffer insertion is a critical netlist optimization technique in Physical Design (PD) that balances trade-offs between Power, Performance, and Area (PPA) metrics. Traditional buffering methods rely heavily on local heuristics, which do not scale and often result in globally sub-optimal solutions. Prior Machine Learning (ML) techniques such as BufFormer attempted to alleviate this limitation but remain prohibitively time-consuming (and sub-optimal) due to their incremental nature. In this paper, we introduce BUFFALO, a generative buffer insertion framework that, for the first time in PD, formulates buffer tree generation as a sequence-to-sequence task solved by Large Language Models (LLMs). Particularly, given a design, BUFFALO performs single-shot generation of buffer trees for all fanout-violating nets and INSTA-selected timing critical nets. Furthermore, Group Relative Policy Optimization (GRPO), a Reinforcement Learning (RL) technique, is employed to refine predicted solutions in a PPA-configurable manner. Experimental results on 9 full-chip designs in a 7nm node demonstrate that BUFFALO outperforms an industry-leading commercial PD tool by 71% in Total Negative Slack (TNS), 67.69% in Worst Negative Slack (WNS), and 83x in runtime without incurring additional power consumption.**

I. Introduction

Interconnect buffering is essential for achieving timing closure in modern VLSI design flows [1], [2], [3]. As interconnect delay increasingly dominates gate delay, at advanced technology nodes, buffer cells can account for over 30% of a chip's total cell count [4]. Traditional buffering approaches typically involve a two-stage pipeline: initially constructing either a timing-driven tree [5] or a Steiner minimum tree [6], [7], [8], followed by wire segmentation [9] to determine candidate buffer sites. Subsequently, pseudo-polynomial van Ginneken's dynamic programming (DP) algorithms [10], [11] are employed for buffer sizing and placement. This sequential methodology severely restricts achievable power-performance-area (PPA) trade-offs, creates an explosively large DP state space requiring heuristic pruning [12], [13], and results in multi-second to multi-minute runtimes per net even for moderate complexities [6]. Furthermore, since timing-driven buffer insertion is NP-complete [14], these conventional methods inherently fail to scale efficiently for large nets and full-chip applications.

Although machine learning and generative models [15], [16] have revolutionized numerous electronic design automation (EDA) tasks, interconnect buffering remains notably underexplored. Few attempts have been made [17], [18] to address these challenges, but these efforts suffer significant drawbacks. For instance, [17] advertises "single-shot" buffer insertion but actually fragments the buffering process into multiple sequential stages—including sink clustering, incremental level-by-level tree growth, buffer sizing, and placement—each introducing cumulative errors, latency, and complex hyperparameter tuning. Because [17] is trained exclusively to mimic [6] + [11] solutions, it is inherently bounded by their suboptimal power-performance-area (PPA) results, with practical training constraints further degrading its performance. Additionally, it provides no mechanism for direct PPA optimization or online refinement and is validated only on isolated synthetic nets without full-chip demonstrations to prove scalability or practical deployment. These critical shortcomings highlight an urgent need for an end-to-end, PPA-driven buffering framework capable of scaling to real-world designs.

[†]This work was conducted during an internship at NVIDIA.

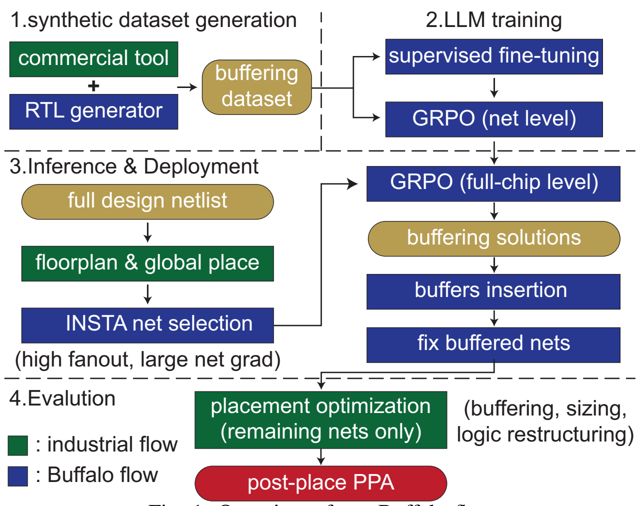

Fig. 1: Overview of our Buffalo flow

To address these limitations, we introduce BUFFALO, the first truly end-to-end buffering framework driven by large language models (LLMs). Unlike fragmented, error-prone multi-stage heuristics, BUFFALO employs a unified full-tree token representation that concurrently predicts buffer-tree topology, sizing, and placement in one seamless inference step, thereby eliminating cumulative errors and significantly enhancing efficiency. To directly optimize multi-objective PPA metrics and surpass traditional PPA ceilings, BUFFALO uniquely employs Group Relative Policy Optimization (GRPO)[19], a state-of-the-art reinforcement learning algorithm proven effective in recent research. To the best of our knowledge, BUFFALO is also the first framework rigorously validated on complex, real-world design benchmarks, demonstrating practical effectiveness and scalability.

Figure 1 illustrates our four-stage BUFFALO methodology. Recognizing that training a robust, generalizable large language model (LLM) fundamentally relies on massive, high-quality data, we begin by developing a net-level netlist generator. This generator synthesizes diverse, realistic buffering scenarios conforming to industry standards. Each generated synthetic net is buffered using a state-of-the-art commercial buffering engine, yielding over 10 million buffered nets that form a substantial dataset for training. In Stage 2, we leverage this rich dataset to fine-tune a pretrained LLM, empowering it to predict comprehensive, industry-grade buffer-tree topologies, sizing, and placements in a single inference pass. To transcend mere imitation and actively discover superior buffering

TABLE I: Key differences: FLUTE+GL [10], [11] vs. BufFormer [17] vs. Buffalo (ours).

Feature	**van Ginneken's style [10], [11]**	**BufFormer [17]**	**Buffalo (ours)**
Dataset size	N/A (non-ML based)	23K nets	10M+ nets
Data source	N/A (non-ML based)	FLUTE+GL synthetic nets	Industrial-grade synthetic nets
Steiner-tree required	Yes	No	No
Sink clustering required	No	Yes	No
Buffer tree construction	Two-stage (Steiner → sizing)	Level-by-level growth + sizing	One-shot tree + sizing generation
Language representation	No	only single tree level token sequence	full-tree token sequence
Model	none (heuristic rules)	6-layer Transformer (trained from scratch)	T5-Large (pretrained LLM)
Refinement	No	No	GRPO multi-objective RL
Optimization objective	Delay-only heuristic	Imitate FLUTE+GL PPA	Direct multi-objective PPA optimization
PPA awareness	No	No	Yes (via GRPO)
Scope	Net-level only	Net-level only	Net + full-chip
Inference speed	Baseline (CPU)	Up to 160× (GPU)	Fastest (one-shot GPU)
Delay evaluation	-	Elmore + LUT	INSTA, commercial tool
Full-chip evaluation	No	No	Yes

† [17] is trained to imitate solutions from [11]; due to irreducible training error, [11] represents an performance upper bound of [17].

strategies, we apply GRPO at the net level to directly optimize multi-objective PPA metrics. In Stage 3, we scale the GRPO to the entire chip, addressing the crucial limitation that localized net-by-net optimizations cannot capture complex cross-net interactions. By employing INSTA.netGrad, we systematically identify the most critical nets and apply full-chip GRPO refinement to derive globally optimal buffering policies. Finally, in Stage 4, we hand off the remaining, less-critical nets to a conventional commercial placement optimization flow and perform an extensive post-placement PPA analysis, thoroughly validating BUFFALO's efficiency and scalability.

Our main contributions are as follows:

- We introduce the first genuinely end-to-end buffering framework utilizing LLMs to concurrently predict complete buffer-tree topology, sizing, and placement in a single inference step.
- We are the first to directly optimize multi-objective PPA metrics using Group Relative Policy Optimization (GRPO).
- We provide the first rigorous full chip validation of ML-driven buffering approach using comprehensive, real-world benchmarks at advanced 7nm technology nodes.

II. Background and Preliminaries

A. Comparative Analysis of Buffering Methods

Interconnect buffering methods broadly fall into two categories: conventional heuristic-based algorithms and emerging machine learning (ML)-based approaches.

Table I contrasts traditional methods, ML-based models, and our BUFFALO framework. Traditional methods (e.g., van Ginneken style [10], [11]) typically involve multi-stage processes—constructing Steiner trees, segmenting wires, and separately sizing and placing buffers via dynamic programming—leading to cumulative errors and limited PPA performance. ML-based methods, such as BufFormer [17], attempt to address these issues using transformer-based models, but still suffer from fragmented representations, indirect imitation-based learning, and insufficient validation limited to isolated nets.

BUFFALO distinguishes itself with three key innovations: (1) A unified full-tree representation enabling one-shot inference of complete buffer-tree solutions. (2) Direct multi-objective PPA optimization through GRPO. (3) Rigorous validation on full-chip benchmarks at advanced 7nm technology nodes, demonstrating superior scalability and practical applicability.

B. Group Relative Policy Optimization (GRPO)

Recently, reinforcement learning (RL) and preference learning has emerged as a promising approach for optimizing complex design tasks [20], [21], [22], [23], [24]. Preference tuning of LLMs traditionally uses methods like Proximal Policy Optimization (PPO) [25], which stabilizes reinforcement learning from human feedback (RLHF)[26] by optimizing a clipped objective with a value-function baseline and KL-divergence penalty. Direct Preference Optimization (DPO) [27] simplifies this further by directly optimizing a classification-style loss, eliminating the reinforcement loop and implicitly controlling policy divergence. However, both PPO and DPO utilize only pairwise or single-sample preferences, underexploiting richer multi-candidate feedback.

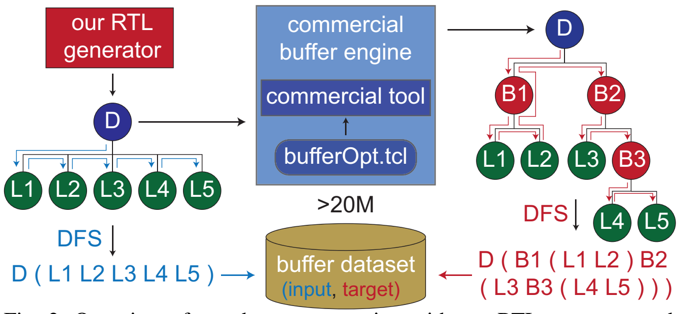

Fig. 2: Overview of our dataset generation with our RTL generator and commercial buffer engine.

Group Relative Policy Optimization (GRPO) [19] addresses this limitation by replacing the traditional value-function critic with a group-level baseline. Specifically, GRPO evaluates multiple candidate outputs simultaneously, computing scalar rewards from a reward model for each candidate. These rewards form group-relative advantages, enabling accurate advantage estimation without needing an explicit value function. GRPO updates the policy using a PPO-inspired clipped objective averaged across candidates. By leveraging richer group feedback, GRPO achieves superior advantage estimation accuracy, increased sample efficiency, and accelerated convergence.

C. INSTA Fast Timing Analysis

GPU acceleration has been increasingly adopted in EDA to handle large-scale design optimization tasks [28], [29], [30], [31]. To support efficient GRPO training, we employ INSTA [28], a GPU-accelerated timing analysis tool. INSTA rapidly computes timing and gradient data, significantly boosting training efficiency and enabling scalable full-chip optimization.

III. Methodologies

A. Large Buffering Dataset Generation

We designed an automated end-to-end pipeline, illustrated in Figure 2, which generates over 20 million paired examples of unbuffered and

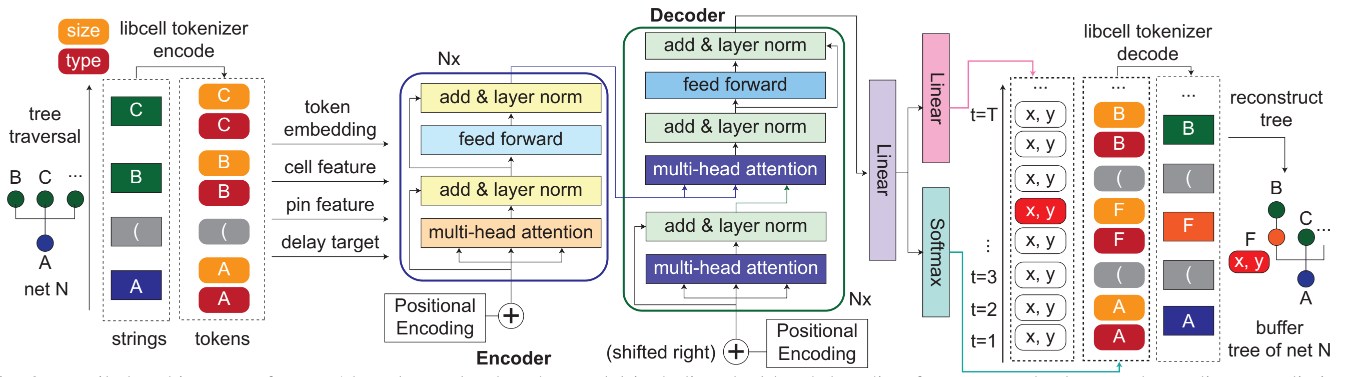

Fig. 3: Detailed architecture of our T5-based encoder-decoder model including dual-head decoding for structured tokens and coordinate prediction.

TABLE II: Parameter sampling ranges used in data generation.

Parameter	Sampling Range
Driver/Sink Type	ASAP7 standard cell library
Driver/Sink Size	ASAP7 standard cell library
Fanout Count	[1, 100]
Delay Target	[100 ps, 1 ns]
Input Transition	$[0, 0.2 \times T_{clk}]$
Input Delay	$[0, 0.2 \times T_{clk}]$
Driver/Sink Placement	[0, Width] × [0, Height]
Leakage/Dynamic Ratio (%)	[0, 100%]

TABLE III: Cell and Pin Features for Model Input

Category	Features
timing arrival	arrival (max/min rise/fall)
capacitance	pin (max/min rise/fall)
slew rate	slew (max/min rise/fall)
slack metrics	slack (max/min rise/fall)
voltage levels	driver pin/ rail voltage (max/min)
fan Metrics	Fan-in; fan-out count; fan-out load
physical Dim.	cell area; bbox; pin count
Library Pin Attributes	Pin cap (max rise/fall); drive resistance (rise/fall); fan-out load

buffered nets. Initially, our RTL net generator synthesizes diverse, high-fanout netlists under realistic constraints outlined in Table II, including variations in driver and sink placements, fanout counts, delay targets, input transitions, and input delays, ensuring comprehensive scenario coverage. Subsequently, we employ a custom `bufferOpt.tcl` script integrated with a commercial physical-design tool to insert buffers while adhering to realistic timing constraints. Each resulting unbuffered and buffered net is serialized into a structured depth-first search (DFS) bracketed sequence (e.g., $D(L_1L_2L_3L_4L_5)$ and $D(B_1(L_1L_2); B_2(L_3B_3(L_4L_5))))$, forming an extensive supervised dataset suitable for training and evaluation.

B. Model Architecture

Building on the T5 encoder–decoder backbone, we enhance the decoder to emit structured tokens and spatial coordinates. The key architectural components are:

1) Transformer Backbone

We adopt the standard T5 architecture, which has stacked multi-head self/cross-attention, and position-wise feed-forward layers.

2) Library-Cell Tokenization

To effectively represent buffer trees for sequence modeling, each buffer tree is serialized into a sequence of tokens following a depth-first search (DFS) traversal order. Instead of assigning a unique token to each distinct gate type and size combination—which would result in a large and sparse vocabulary pool—we adopt a factorized tokenization scheme. Specifically, we split each library-cell name

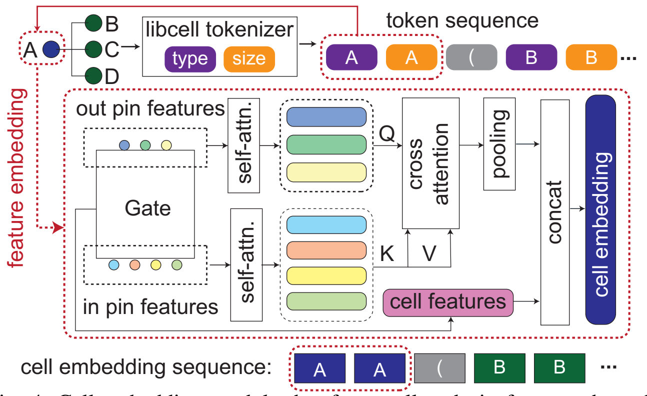

Fig. 4: Cell-embedding module that fuses cell and pin features through self- and cross-attention mechanisms.

into two distinct tokens: one token denoting the gate type and another specifying the gate size (as illustrated in Figure 3). This decomposition naturally exploits the Cartesian structure inherent to standard cell libraries, significantly reducing the total vocabulary size and enabling efficient parameter sharing among cells of similar attributes. Additionally, we introduce special tokens—< (> and <) >—to explicitly encode tree structures within sequences. Standard sequence delimiters such as `<s>`, `</s>`, and padding token `<pad>` are also incorporated to facilitate structured modeling and sequence completion tasks.

3) Input Representation and Embedding

To jointly capture a net's structural topology and detailed per-cell physical attributes, each net targeted for buffering is represented by two parallel, positionally aligned sequences: (1) a bracket-delimited token sequence generated via DFS, and (2) a corresponding cell-embedding sequence. Each token in the DFS-based sequence precisely corresponds, position-by-position, to a cell embedding, ensuring exact alignment between gate tokens and their associated features. For each DFS-ordered cell, a dedicated embedding module (illustrated in Figure 4) integrates intrinsic cell-level features with pin-level attributes. Specifically, to accommodate cells with variable pin counts and accurately model signal propagation through internal cell arcs, input and output pins undergo intra-group self-attention followed by cross-attention from inputs to outputs. A pooling operation then summarizes these pin embeddings into a fixed-size vector, which is subsequently concatenated with intrinsic cell attributes. Finally, these two aligned sequences are fused by the input embedding module (Figure 6), producing a unified representation that enables the LLM to effectively reason about both structural topology and electrical context at each cell.

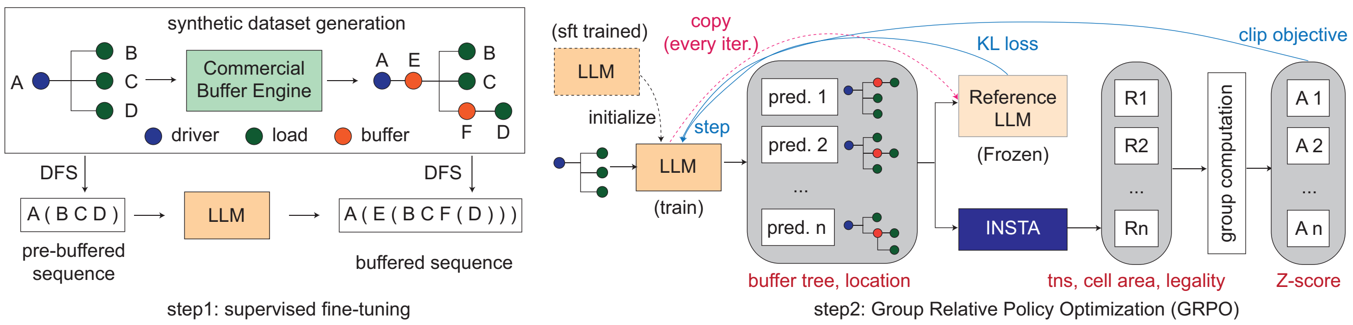

Fig. 5: Overall training workflow illustrating supervised fine-tuning followed by iterative GRPO policy refinement.

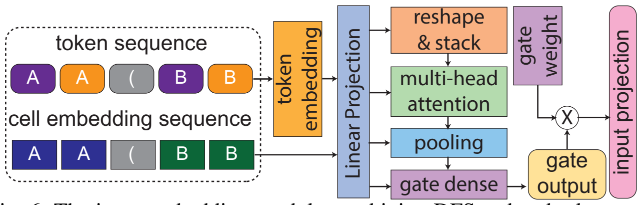

Fig. 6: The input embedding module combining DFS-ordered tokens and cell embeddings to jointly represent net structure and physical context.

4) Dual-Head Decoding for Buffer and Location Prediction

At each decoding step t, the decoder hidden state $\mathbf{h}_t$ is shared by two parallel heads:

- Structural head: selects the next token—either a cell identifier or a structural symbol—from all available tokens.
- Location head: This coordinate prediction is meaningful only at decoding steps corresponding exactly to the buffer's size token (which immediately follows the buffer's type token), specifying the buffer's physical location. Coordinates predicted at all other token positions (e.g., type tokens, drivers, or loads) are ignored.

C. Supervised Fine-Tuning

Our supervised fine-tuning strategy frames buffer-tree synthesis as a sequence-to-sequence task, employing a pretrained T5 encoder–decoder model to translate depth-first search (DFS)-linearized unbuffered net sequences directly into buffered nets in single shot. Specifically, given each input net q, the process (outlined in Algorithm 1) begins by tokenizing library cells into paired tokens (type and size) and extracting associated cell features (lines 6–8). These token sequences and features are then embedded separately, combined with positional encodings, and jointly fed into the T5 encoder to generate context-rich embeddings (lines 11–13). The decoder subsequently outputs a sequence of predictions that include both structural tokens and spatial coordinates for buffer placements (line 15). To effectively train this joint decoding scheme, we introduce a composite loss function comprising two components: a structural classification loss and a masked coordinate regression loss (lines 17–22):

$$\mathcal{L} = \sum_i \Big[-\log p(\text{token}_i) + \lambda\, m_i\, \|\hat{\mathbf{l}}_i - \mathbf{l}_i\|_2^2 \Big], \qquad (1)$$

where the mask m_i activates coordinate loss only at buffer size token positions, ensuring precise learning of meaningful buffer locations (lines 19–21). Here $\hat{\mathbf{l}}_i$ and $\mathbf{l}_i$ denote predicted and ground-truth buffer coordinates, respectively, and the hyperparamete λ balances structural and spatial loss.

D. Group Relative Policy Optimization (GRPO)

While supervised fine-tuning (SFT) effectively trains the policy π_θ to imitate reference solutions generated by commercial buffering tools, it treats all solutions as equally optimal, lacking the ability to balance trade-offs between power, performance, and area (PPA).

Algorithm 1 BUFFALO: Supervised Fine-Tuning

```
Input: initial parameters $\theta_{\text{init}}$; dataset $\mathcal{D} = \{(q,t)\}$; regression weight $\lambda$; batch size $M$; learning rate $\eta$; epochs $E$
Output: fine-tuned parameters $\theta$
1:  $\theta \leftarrow \theta_{\text{init}}$
2:  for $e = 1, \ldots, E$ do
3:      for each mini-batch $\mathcal{D}_b$ of size $M$ do
4:          $\mathcal{L} \leftarrow 0$
5:          for each sample $(q,t) \in \mathcal{D}_b$ do
6:              // 1. Tokenize libcells into (type, size)s and extract features
7:              tokens $\leftarrow$ libcell_tokenizer$(q)$
8:                      ▷ e.g. [A_type, A_size, '(', B_type, B_size, ...]
9:              feats $\leftarrow$ feature_extractor$(q)$
10:             // 2. Embed tokens & features + positional encoding
11:             $E_{\text{tok}} \leftarrow \text{Embed}_{\text{tok}}(\text{tokens})$
12:             $E_{\text{feat}} \leftarrow \text{Embed}_{\text{feat}}(\text{feats})$
13:             $X \leftarrow E_{\text{tok}} + E_{\text{feat}} + \text{PosEnc}$
14:             // 3. Forward through encoder–decoder
15:             $(p_i, \hat{\ell}_i)_{i=1}^{T} \leftarrow \text{Decoder}_\theta(\text{Encoder}_\theta(X))$
16:             // 4. Compute composite loss (Eq. 1)
17:             for $i = 1, \ldots, T$ do
18:                 $\mathcal{L} \mathrel{+}= -\log p_i(t_i)$
19:                 $m_i \leftarrow \begin{cases} 1 & \text{if token}_i \text{ is a size token} \\ & \quad \text{and token}_{i-1} \text{ is a buffer type} \\ 0 & \text{otherwise} \end{cases}$
20:                 $\mathcal{L} \mathrel{+}= \lambda\, m_i\, \|\hat{\ell}_i - \ell_i\|_2^2$
21:         // 5. Gradient update
22:         $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$
```

To explicitly address this limitation, we introduce *Group Relative Policy Optimization* (GRPO), a lightweight, policy-based refinement method designed to directly align buffer-tree predictions with desired PPA objectives. GRPO operates by evaluating multiple candidate buffer trees per net and optimizing the policy based on their relative PPA quality. Specifically, for each net, we generate candidate buffer-tree solutions and quantify their relative performance using Group Relative Advantage Estimation (GRAE):

$$A^{(g)} = w_1 \frac{\Delta\text{TNS}^{(g)} - \mu_{\text{TNS}}}{\sigma_{\text{TNS}}} + w_2 \frac{-\Delta Area^{(g)} - \mu_{\text{Area}}}{\sigma_{\text{Area}}}, \qquad (2)$$

where ΔTNS is the improvement in timing slack and $\Delta Area$ epresents area increase. Each objective is standardized independently to maintain scale invariance and subsequently combined via convex weighting factors w_1, w_2. GRPO employs the mean advantage across candidates within each group as a baseline, eliminating the need for an explicit value-function critic or external ranking models, thereby significantly reducing computational overhead. To efficiently perform the required multi-step evaluations, we integrate INSTA [?], a GPU-

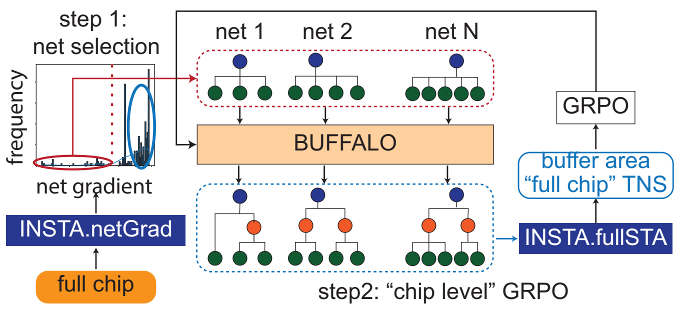

Fig. 7: Illustration of chip-level GRPO strategy identifying and optimizing critical nets for balanced chip-wide timing and area trade-offs.

accelerated timing analysis engine capable of rapid assessment.

In the mean time, we leverage GRPO to explicitly enforce the structural legality of generated buffer trees by penalizing outputs that violate predefined structural requirements. An output is considered *legal* only if it satisfies all of the following conditions:

- Driver match: the first token exactly matches the driver cell's name.
- Balanced parentheses: all parentheses are properly nested and closed only at the end of the sequence.
- Buffer grouping: each opening parenthesis (immediately follows a buffer candidate rather than an input leaf.
- Load-only groups: groups indicated by parentheses contain only leaf sinks and no additional buffers.
- Complete leaf coverage: each original input leaf sink appears exactly once without repetition or omission.

To strongly discourage illegal predictions, any candidate failing these criteria is assigned a fixed z-score advantage of -3 (three standard deviations below the group mean) after GRAE, significantly penalizing outputs that cannot be parsed into valid buffer trees. During inference, we apply token masking to enforce legality in every decoding step.

Algorithm 2 summarizes steps in each GRPO iteration:

1) Set the reference policy π_{ref} to the current policy π_θ, or initialize it from the SFT policy at the first iteration.
2) Sample a mini-batch of nets, each generating G candidate buffer-tree solutions under π_θ.
3) Evaluate each candidate based on timing slack improvement ΔTNS and area change $\Delta Area$.
4) Update π_θ by maximizing a clipped surrogate objective with KL-divergence regularization:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\,\{o_i\}\sim\pi_{\text{ref}}}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min(r_{i,t}\,\hat{A}_{i,t},\; \text{clip}(r_{i,t},1-\epsilon,1+\epsilon)\,\hat{A}_{i,t}) - \beta\, D_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}})\Bigg]. \tag{3}$$

$$\text{where}\quad r_{i,t} = \frac{\pi_\theta(o_{i,t}\mid q, o_{i,<t})}{\pi_{\text{ref}}(o_{i,t}\mid q, o_{i,<t})},\quad q\sim P(Q). \tag{4}$$

and enforce a trust region with a KL-penalty:

$$D_{\text{KL}}[\pi_\theta\|\pi_{\text{ref}}] = \mathbb{E}_{o\sim\pi_{\text{ref}}}\Big[\log\frac{\pi_{\text{ref}}(o)}{\pi_\theta(o)}\Big]. \tag{5}$$

E. Chip level GRPO

Although per-net GRPO iterations successfully optimize local power-performance-area (PPA) trade-offs, independently applying these policies to individual nets can inadvertently degrade overall chip-level performance. For instance, optimizing an upstream net locally might shift critical timing constraints downstream, causing unnecessary power and area overhead. To address this issue, we propose a comprehensive chip-level GRPO strategy that coordinates buffering decisions

Algorithm 2 BUFFALO: Group Relative Policy Optimization (GRPO)

```
Input: initial policy $\pi_{\theta_{\text{init}}}$; INSTA evaluator; dataset $\mathcal{D}$; weights $w_1, w_2$;
    batch size $M$; samples per prompt $G$; inner its $\mu$
Output: optimized policy $\pi_\theta$
 1: $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
 2: for $i = 1,\dots,I$ do
 3:     $\pi_{\text{ref}} \leftarrow \pi_\theta$                          ▷ freeze for KL/trust-region
 4:     for $j = 1,\dots,M$ do
 5:         Sample mini-batch $\mathcal{D}_b \subseteq \mathcal{D}$
 6:         for each $q \in \mathcal{D}_b,\ g = 1,\dots,G$ do
 7:             $o^g \sim \pi_\theta(\cdot \mid q)$                     ▷ generate buffer tree
 8:             $\Delta\text{TNS}^g \leftarrow \text{INSTA}(o^g)$
 9:             $\Delta A^g \leftarrow \text{bufferArea}(o^g)$
10:             $\ell^g \leftarrow \mathbb{I}\{\text{valid}(o^g)\}$              ▷ 1 if valid, else 0
11:         Compute group stats:
12:             $(\mu_1,\sigma_1) \leftarrow \text{mean},\text{std}(\{\Delta\text{TNS}^g\})$
13:             $(\mu_2,\sigma_2) \leftarrow \text{mean},\text{std}(\{\Delta A^g\})$
14:         Compute normalized advantages:
15:             $A^g_{\text{TNS}} \leftarrow (\Delta\text{TNS}^g - \mu_1)/\sigma_1$
16:             $A^g_{\text{Area}} \leftarrow (-\Delta A^g - \mu_2)/\sigma_2$
17:         for $g = 1,\dots,G$ do
18:             if $\ell^g = 0$ then
19:                 $A^g_{\text{TNS}} \leftarrow -3,\ A^g_{\text{Area}} \leftarrow -3$       ▷ penalize illegal
20:             Combine weighted advantage:
21:                 $A^g \leftarrow w_1\,A^g_{\text{TNS}} + w_2\,A^g_{\text{Area}}$
22:         for $k = 1,\dots,\mu$ do
23:             Update $\pi_\theta$ by maximizing clipped GRPO surrogate (with
24:             $\text{KL}(\pi_\theta\|\pi_{\text{ref}})$)
```

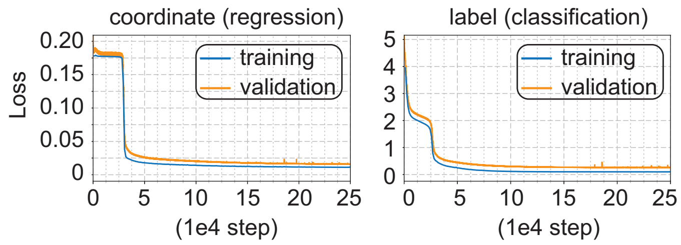

Fig. 8: Training and validation loss curves for coordinate regression (left) and label classification (right) during supervised fine-tuning.

for multiple critical nets in parallel, as detailed in Algorithm 3. Because buffering inevitably increases area and power, we begin by identifying the most timing-critical nets via INSTA.netGrad, which quantifies each net's impact on overall timing delay. As illustrated in Figure 7, we select these top-ranked critical nets and apply our trained buffering policy collectively in a unified GRPO update. During chip-level optimization, the GRPO loss function (Equation 3) incorporates global timing metrics evaluated through INSTA.fullSTA along with the corresponding area overhead. This combined evaluation ensures the updated policy properly balances global timing improvements against area and power penalties. After several fine-tuning iterations, the policy adapts to the overall timing picture, minimizing overhead while enhancing design performance. The resulting optimized buffering configuration is subsequently integrated into the standard commercial design flow for final verification and sign-off.

IV. Experimental Results

This section presents the experimental validation of our proposed buffering methodology, focusing on supervised fine-tuning and Group Relative Policy Optimization (GRPO). Experiments were conducted on a Linux compute cluster with 8 NVIDIA A100 GPUs (each with 96GB HBM2E), and AMD EPYC 7742 64-Core Processor with 2TB of RAM. utilizing Python, PyTorch, and Hugging Face Transformers

Algorithm 3 BUFFALO: Chip-Level GRPO Fine-Tuning

```
Input: net-level checkpoint $\pi_{\theta_{\text{net}}}$; INSTA.netGrad; INSTA.fullSTA; net
    set $\mathcal{N}$; weights $w_1, w_2$; batch size $M$; inner its $\mu$; top-$k$ selector
Output: chip-tuned policy $\pi_\theta$
 1: $\pi_\theta \leftarrow \pi_{\theta_{\text{net}}}$
 2: for $i = 1, \dots, I$ do
 3:     $\pi_{\text{ref}} \leftarrow \pi_\theta$                      ▷ freeze for KL/trust-region
 4:     Sample $\mathcal{N}_b \subseteq \mathcal{N}$ with $|\mathcal{N}_b| = M$
 5:     for each net $n \in \mathcal{N}_b$ do
 6:         $s_n \leftarrow$ INSTA.netGrad$(n)$                      ▷ impact score
 7:     Select high-impact nets $\mathcal{H} \leftarrow$ top$-$k$(\{s_n\})$
 8:     for each $n \in \mathcal{H}$ do
 9:         $o_n \sim \pi_\theta(\cdot \mid n)$                      ▷ generate buffer tree
10:         $\Delta\text{TNS}_n \leftarrow$ INSTA.fullSTA$(o_n)$
11:         $\Delta A_n \leftarrow$ bufferArea$(o_n)$
12:         $\ell_n \leftarrow \mathbb{I}\{\text{valid}(o_n)\}$      ▷ 1 if valid, else 0
13:     Compute group stats:
14:         $(\mu_1, \sigma_1) \leftarrow \text{mean}, \text{std}(\{\Delta\text{TNS}_n\}_{n \in \mathcal{H}})$
15:         $(\mu_2, \sigma_2) \leftarrow \text{mean}, \text{std}(\{\Delta A_n\}_{n \in \mathcal{H}})$
16:     for each $n \in \mathcal{H}$ do
17:         $A_{\text{TNS},n} \leftarrow (\Delta\text{TNS}_n - \mu_1)/\sigma_1$
18:         $A_{\text{Area},n} \leftarrow (-\Delta A_n - \mu_2)/\sigma_2$
19:         if $\ell_n = 0$ then
20:             $A_{\text{TNS},n}, A_{\text{Area},n} \leftarrow -3, -3$
21:         $A_n \leftarrow w_1\, A_{\text{TNS},n} + w_2\, A_{\text{Area},n}$
22:     for $k = 1, \dots, \mu$ do
23:         Update $\pi_\theta$ by maximizing clipped GRPO surrogate with
24:         KL$(\pi_\theta \parallel \pi_{\text{ref}})$
25:     Record Pareto-front solutions from $\{(n, o_n)\}_{n \in \mathcal{H}}$
```

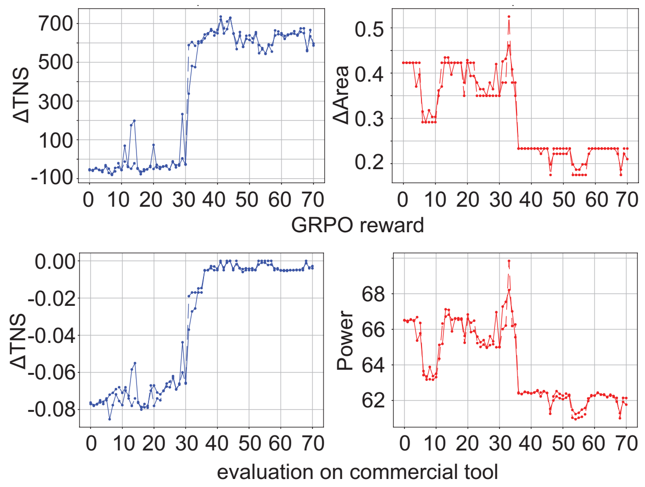

Fig. 9: Validation of GRPO reward proxy vs. actual design objectives.

for model implementation and training.

A. Supervised Fine-Tuning Results

Figure 8 depicts the supervised fine-tuning (SFT) loss curves of our T5-based encoder-decoder model. This model concurrently predicts buffer-tree structures (classification) and buffer coordinates (regression). The training on $20M$ dataset spanned 6 days, demonstrating a consistent decrease in regression loss (mean-squared error for coordinate prediction) from approximately 0.18 to below 0.01, and classification loss (cross-entropy for tree tokens) from roughly 4.5 to about 0.1. The smooth training and validation curves underscores the robustness and stability of our dual-task training procedure.

B. GRPO Rewards Trajectory

For GRPO, we set the learning rate of the policy model as $1e-4$. The KL coefficient is 0.04. For each net, we sample 10 outputs, and

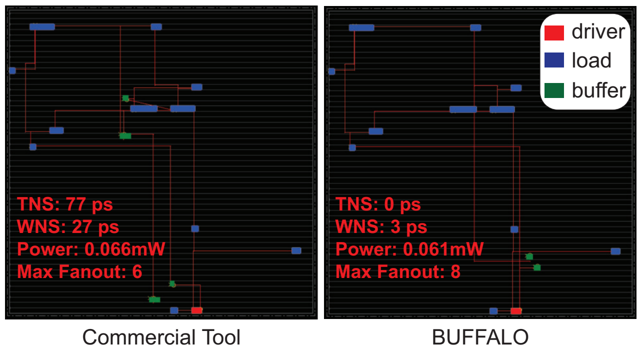

Fig. 10: Commercial tool (left) vs. BUFFALO(right).

the training batch size is 16. With INSTA as a reward proxy, each batch spans $\sim 5\,\text{second}$. Figure 11 illustrates the training curves of our GRPO-based optimization process across six representative nets. Each iteration comprises multiple update steps (M), as detailed in Algorithm 2. Each update step involves sampling groups of buffer-tree candidates, computing their normalized advantages (GRAE), and refining the policy parameters π_θ based on reward signals provided by the INSTA static timing analyzer. We utilize improvements in Total Negative Slack (ΔTNS) (in blue, left axis) and reductions in buffer area overhead (Δarea) (in red, right axis) as reward metrics. Solid lines represent mean values, while dashed lines indicate medians calculated across candidate groups. Horizontal benchmarks correspond to baseline solutions obtained from a commercial tool for direct comparison. Over initial iterations, GRPO consistently achieves marked reductions in buffer area overhead while simultaneously enhancing timing slack by several tens to hundreds of picoseconds. Notably, both area and slack metrics stabilize after approximately 80–100 iterations, demonstrating effective convergence of our GRPO framework. The results clearly indicate that GRPO learns buffer insertion strategies that significantly improve upon baseline approaches, effectively achieving similar or improved timing closure with substantially less buffer area overhead.

C. GRPO Buffer-Tree Trajectory

Figure 12 illustrates the iterative progression of buffer insertion for a representative net during GRPO training. Each snapshot shows the driver (red), sinks (blue), and inserted buffers (green). Metrics for area overhead (ΔArea) and timing-slack change (ΔTNS) are annotated above each subplot. Arrows between panels indicate the metric shifts from iterations 0 to 70. Starting from the supervised baseline (iteraion 0: (Δarea $=$ 0.64), ΔTNS $=$ 2032.67)), GRPO progressively refines buffer-tree topology and placements. By iteration 70, area overhead decreases to 0.36 and slack improves to 2447.82 ps, showcasing GRPO's simultaneous optimization of tree topology and locations of buffers for enhanced area-timing trade-offs.

D. Validation with Real-World Objectives

To validate the alignment of our GRPO proxy rewards with actual design objectives, we export its buffer solutions of a representative net over the GRPO iterations to a commercial physical design flow to measure the actual TNS and power. Figure 9 compares the proxy metrics (top row: ΔTNS and ΔArea) during GRPO training) with actual tool-measured outcomes (bottom row: ΔTNS and ΔArea). The notable increase in GRPO proxy ΔTNS around iteration 30 closely corresponds to an equivalent improvement in actual timing slack, while the steady decrease in proxy area aligns clearly with a reduction in measured power. This strong correlation underscores the reliability of our proxy reward, confirming that GRPO effectively optimizes for genuine timing and power objectives.

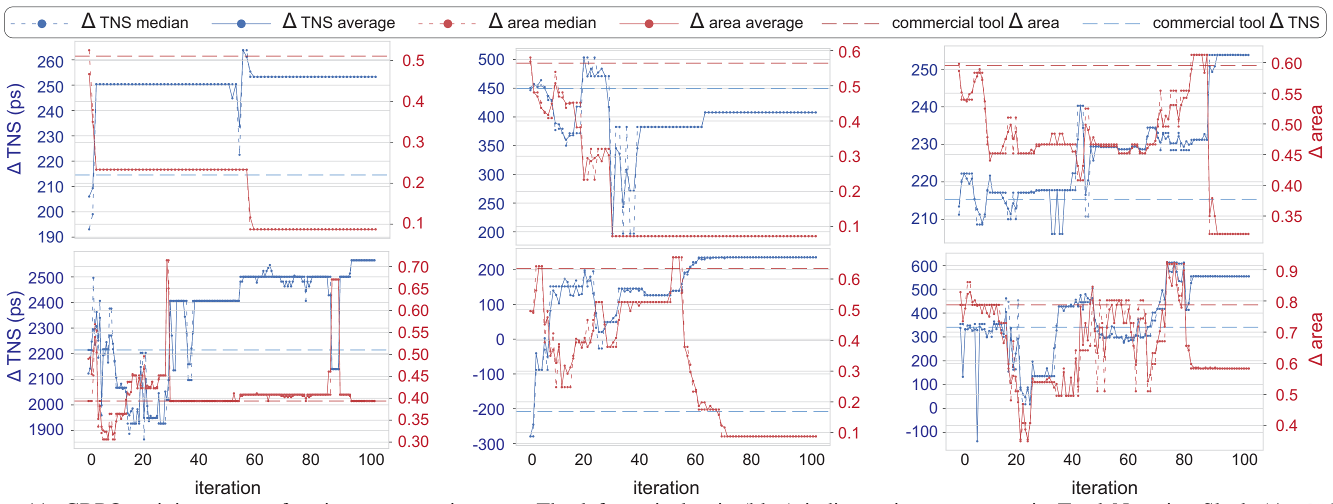

Fig. 11: GRPO training curves for six representative nets. The left vertical axis (blue) indicates improvements in Total Negative Slack (ΔTNS), whereas the right vertical axis (red) shows changes in buffer area overhead ($\Delta Area$). Solid and dashed lines represent mean and median values, respectively, over sampled candidates. Horizontal lines denote baseline metrics from a commercial tool.

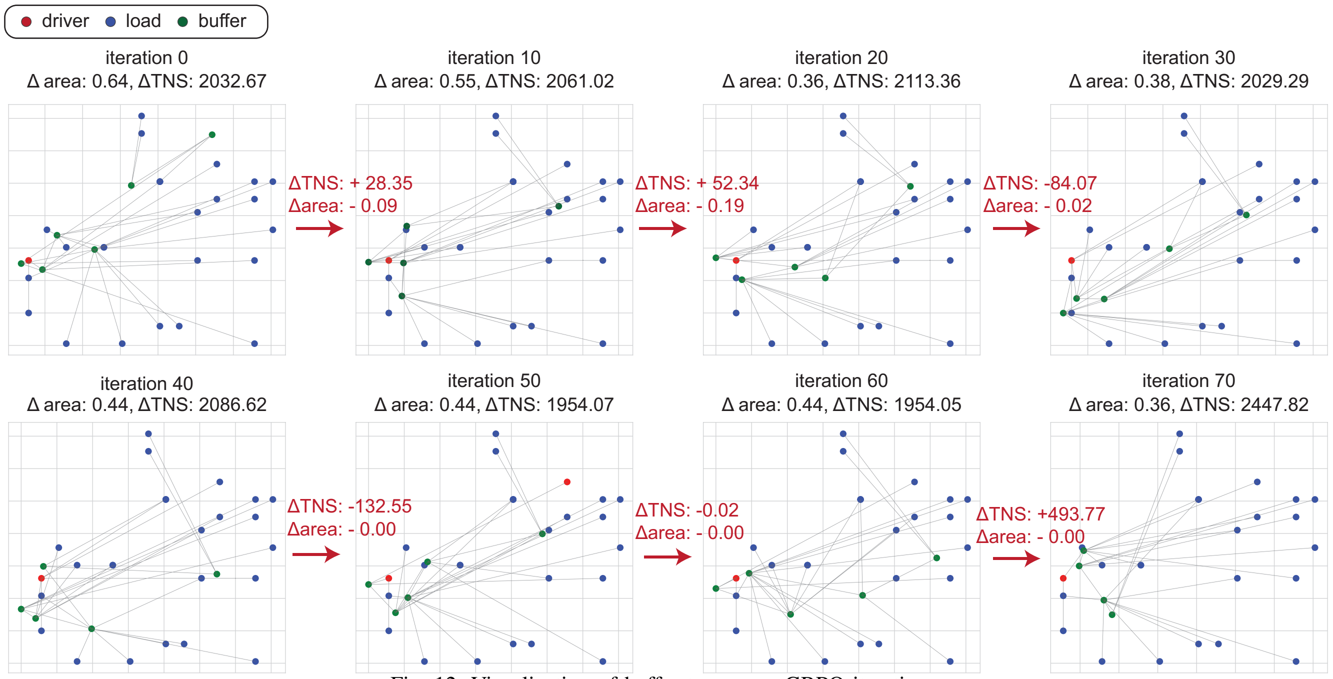

Fig. 12: Visualization of buffer trees over GRPO iterations.

E. Comparison with Commercial Tool

For buffering a single net, our approach typically requires between 0.5 to 1 second, whereas commercial tools commonly need around 30 to 43 seconds for nets with high fanout. Specifically, on a representative net with 50 sinks, BUFFALO achieves up to an 83× speedup compared to commercial solutions. Figure 10 contrasts buffer insertion outcomes for a representative net between a commercial tool (left) and our BUFFALO GRPO (right). The commercial solution yields a Total Negative Slack (TNS) of 77 ps, Worst Negative Slack (WNS) of 27 ps, power consumption of 0.066 mW, and a maximum fanout of 6. In comparison, Buffalo achieves zero TNS, significantly reduces WNS to just 3 ps, lowers power consumption to 0.061 mW, and accommodates a higher maximum fanout of 8—well within the design's max fanout constraint (16). This direct comparison demonstrates that our LLM-driven, GRPO-refined buffering approach surpasses commercial methods by delivering superior timing closure and lower power consumption.

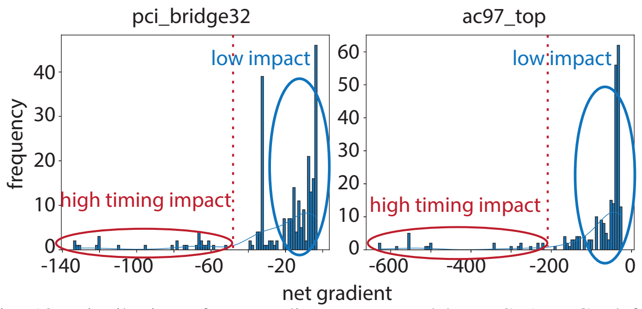

Fig. 13: Distribution of net gradients computed by INSTA.netGrad for the `pci_bridge32` and `ac97_top` designs.

TABLE IV: TNS, Power and WNS across benchmarks. Percentage improvements relative to the commercial baseline are in parentheses.

Benchmark	Commercial Tool			[6]+[11]/ [17]			BUFFALO SFT (ours)			BUFFALO GRPO (ours)		
	TNS (ns)	Power (mW)	WNS (ns)	TNS (ns)	Power (mW)	WNS (ns)	TNS (ns)	Power (mW)	WNS (ns)	TNS (ns)	Power (mW)	WNS (ns)
ac97_top	-139.27	59.79	-0.27	-154.70	59.60	-0.15	-135.03	59.65	-0.19	-101.27 (−27.29%)	60.66	-0.18 (−33.58%)
aes	-45.75	7.92	-0.18	-65.85	8.00	-0.24	-43.24	8.13	-0.19	-23.14 (−49.42%)	7.99	-0.15 (−18.13%)
des	-4.29	4.70	-0.13	-10.81	4.35	-0.25	-2.67	4.47	-0.11	-1.24 (−71.10%)	4.31	-0.04 (−67.69%)
i2c_master	-4.79	2.86	-0.15	-5.01	2.89	-0.15	-4.79	2.92	-0.14	-4.66 (−2.71%)	2.84	-0.10 (−33.99%)
mc_top	-125.37	11.41	-0.40	-140.03	11.53	-0.33	-125.72	11.57	-0.37	-107.66 (−14.13%)	11.62	-0.36 (−9.60%)
pci_bridge32	-321.57	37.82	-0.23	-449.17	38.29	-0.35	-294.60	38.24	-0.21	-188.99 (−41.23%)	38.29	-0.28 (+21.15%)
tv80s	-41.86	3.01	-0.19	-62.24	3.02	-0.26	-39.36	3.01	-0.20	-25.10 (−40.04%)	2.99	-0.16 (−17.37%)
arianne133	-653.82	497.09	-0.28	-829.81	496.05	-0.52	-674.76	496.80	-0.24	-210.87 (−67.77%)	496.60	-0.20 (−30.63%)
arianne136	-277.40	503.54	-0.13	-534.94	504.18	-0.28	-351.27	502.19	-0.13	-175.30 (−36.80%)	503.63	-0.09 (−33.33%)

† [17] is trained to imitate solutions from [11]; due to irreducible training error, [6]+[11] represents an upper bound and a much stronger baseline.

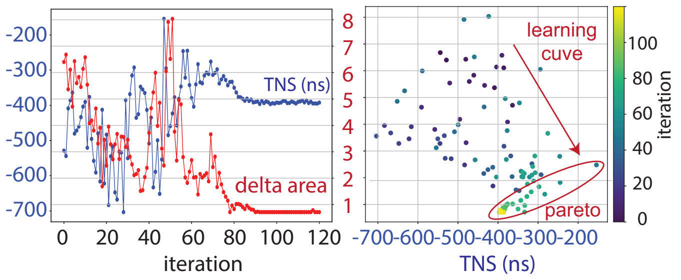

Fig. 14: Visualization of Total Negative Slack vs. ΔArea over 120 iterations of full-chip GRPO optimization.

F. Net Selection and Chip-Level GRPO

The net gradient computed by INSTA.netGrad quantitatively measures the sensitivity of the TNS to perturbations in individual net delays. Figure 13 illustrates the distribution of these net gradients for the `pci_bridge32` and `ac97_top` designs. The majority of nets exhibit gradients near zero, indicating minimal influence on timing closure. However, a distinct subset of nets demonstrates significantly negative gradients, forming a pronounced negative tail in the distribution. We define the "knee point" as the gradient threshold at which the distribution sharply transitions from low-impact to high-impact. To efficiently and effectively enhance overall PPA metrics, we target our buffering effort by selecting nets exceeding this threshold and violating the maximum fanout constraint for GRPO.

Figure 14 illustrates the full-chip GRPO optimization trajectory for the `pci_bridge32` design. The left panel plots the evolution of TNS (in blue) and buffering area overhead (Δarea, in red) across 120 iterations. Initially, both metrics exhibit fluctuations as the policy explores diverse buffering configurations but subsequently stabilize and converge toward significantly improved TNS and reduced area. The right panel depicts the trade-off between TNS and Δarea, with a color gradient representing iteration progression. Early iterations generate scattered solutions in suboptimal regions (top-left). As iterations proceed, solutions systematically transition toward the Pareto frontier with reduced area and slack (bottom-right). This optimization trajectory demonstrates that the full-chip GRPO effectively guides buffering toward superior PPA trade-offs.

V. Full-Chip Evaluation Results

We export the buffering solution generated by the full-chip GRPO stage to the real design and run it through subsequent optimization and post-placement PPA evaluation, as described in Flow 1. We compare our results against a commercial tool and state-of-the-art (SOTA) methods [11], [17]. Since [17] is trained to learn the buffering solutions from [11], and given the irreducible training error, the method of [11] represents an upper bound and thus a stronger baseline. We therefore use the latter directly as a stronger baseline for comparison. For the commercial tool, we use its default optimization engine. For [11], we adopt the open-source implementation [32]. Note that [17], [11] do not inherently include a net-selection mechanism and operate only at the net level rather than the chip level. However, for a fair comparison, we buffer the exact same nets as BUFFALO, with the nets selected by INSTA, and run them through the identical downstream optimization flow for PPA evaluation. The results are summarized in Table I.

VI. Full-Chip Evaluation Results

We evaluate the BUFFALO framework on nine benchmarks under the ASAP7 $7\,\mathrm{nm}$ technology node, following the workflow illustrated in Figure 1. Specifically, we deploy buffering solutions generated by our full-chip GRPO approach, subsequently apply placement optimization on remaining nets, and conduct detailed post-placement Power-Performance-Area (PPA) analysis.

To establish rigorous evaluation baselines, we compare BUFFALO GRPO against a leading commercial buffer tool as well as SOTA academic buffering methods [11], [17]. For the commercial baseline, we employ its built-in optimization engine with default settings. Among the academic baselines, [17] is specifically trained to imitate the buffering solutions from [11]. Given the inherent irreducible errors associated with learning, the method of [11] provides a theoretically stronger baseline. We implement this using the open-source OpenPhySyn framework[32]. It is noteworthy that both academic baselines [17], [11] operate at a per-net granularity, lacking scalability for full-chip buffering or intrinsic mechanisms for efficient chip-level net selection. Therefore, to ensure an equitable comparison, we enhance their methodologies by applying buffering solely to the exact subset of nets identified as timing-critical by our INSTA-driven net-selection stage. Subsequently, all evaluated methodologies undergo an identical downstream optimization and evaluation process.

Table I demonstrates the substantial performance advantage of BUFFALO GRPO, achieving improvements of up to 77.7% in TNS and 67.7% in WNS compared to the commercial baseline, with minimal or negligible additional power overhead. These significant gains confirm that our BUFFALO framework effectively generates buffer-tree solutions that optimally balance timing closure improvements with power efficiency, thereby enhancing overall PPA outcomes.

VII. Conclusion

In this paper, we introduce BUFFALO, the first LLM-based buffering framework capable of predicting complete buffer trees in a single shot and directly optimizing multi-objective PPA via GRPO. Comprehensive full-chip validation on nine benchmarks under a $7nm$ node demonstrated that BUFFALO outperforms an industry-leading commercial PD tool by 77.7% in TNS, and 67.7% in WNS, without incurring additional power consumption, demonstrating BUFFALO's practicality for advanced physical design flows.

References

[1] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, *Handbook of algorithms for physical design automation*. CRC press, 2008.

[2] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*, vol. 312. Springer, 2011.

[3] N. A. Sherwani, *Algorithms for VLSI physical design automation*. Springer Science & Business Media, 2012.

[4] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, "Repeater scaling and its impact on cad," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 451–463, 2004.

[5] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh, "Prim-dijkstra revisited: Achieving superior timing-driven routing trees," in *Proceedings of the 2018 International Symposium on Physical Design*, pp. 10–17, 2018.

[6] C. Chu and Y.-C. Wong, "Flute: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2007.

[7] C. J. Alpert, G. Gandham, J. Hu, J. Neves, S. T. Quay, and S. S. Sapatnekar, "Steiner tree optimization for buffers, blockages, and bays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 4, pp. 556–562, 2001.

[8] C. J. Alpert, J. Hu, S. S. Sapatnekar, and C. Sze, "Accurate estimation of global buffer delay within a floorplan," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1140–1145, 2006.

[9] C. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proceedings of the 34th Annual Design Automation Conference*, pp. 588–593, 1997.

[10] L. P. Van Ginneken, "Buffer placement in distributed rc-tree networks for minimal elmore delay," in *1990 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 865–868, IEEE, 1990.

[11] J. Lillis, C.-K. Cheng, and T.-T. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, 1996.

[12] R. Chen and H. Zhou, "Efficient algorithms for buffer insertion in general circuits based on network flow," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pp. 322–326, IEEE, 2005.

[13] Z. Jiang and W. Shi, "Circuit-wise buffer insertion and gate sizing algorithm with scalability," in *Proceedings of the 45th annual Design Automation Conference*, pp. 708–713, 2008.

[14] W. Shi, Z. Li, and C. J. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*, pp. 609–614, IEEE, 2004.

[15] Y.-C. Lu, H. Ren, H.-H. Hsiao, and S. K. Lim, "Dream-gan: Advancing dreamplace towards commercial-quality using generative adversarial learning," in *Proceedings of the 2023 International Symposium on Physical Design*, ISPD '23, (New York, NY, USA), p. 141–148, Association for Computing Machinery, 2023.

[16] Y.-C. Lu, H. Ren, H.-H. Hsiao, and S. K. Lim, "GAN-Place: Advancing Open Source Placers to Commercial-quality Using Generative Adversarial Networks and Transfer Learning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 2, pp. 1–17, 2024.

[17] R. Liang, S. Nath, A. Rajaram, J. Hu, and H. Ren, "Bufformer: A generative ml framework for scalable buffering," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pp. 264–270, 2023.

[18] H. Wu, Z. Huang, X. Li, and W. Zhu, "Aito: Simultaneous gate sizing and buffer insertion for timing optimization with gnns and rl," *Integration*, vol. 98, p. 102211, 2024.

[19] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[20] H.-H. Hsiao, Y.-C. Lu, P. Vanna-Iampikul, and S. K. Lim, "Fasttuner: Transferable physical design parameter optimization using fast reinforcement learning," in *Proceedings of the 2024 International Symposium on Physical Design*, pp. 93–101, 2024.

[21] H.-H. Hsiao, P. Vanna-Iampikul, Y.-C. Lu, and S. K. Lim, "Ml-based physical design parameter optimization for 3d ics: From parameter selection to optimization," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.

[22] H.-H. Hsiao, S. Kundu, W. Zeng, W.-T. J. Chan, D. Guo, and S. K. Lim, "Insightalign: A transferable physical design recipe recommender based on design insights," in *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025.

[23] Y.-C. Lu, S. Nath, V. Khandelwal, and S. K. Lim, "Rl-sizer: Vlsi gate sizing for timing optimization using deep reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 733–738, 2021.

[24] Y.-C. Lu, W.-T. Chan, D. Guo, S. Kundu, V. Khandelwal, and S. K. Lim, "Rl-ccd: Concurrent clock and data optimization using attention-based self-supervised reinforcement learning," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[26] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[27] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, pp. 53728–53741, 2023.

[28] Y.-C. Lu, Z. Guo, K. Kunal, R. Liang, and H. Ren, "Insta: An ultra-fast, differentiable, statistical static timing analysis engine for industrial physical design applications," in *2025 62th ACM/IEEE Design Automation Conference (DAC)*, 2025.

[29] H.-H. Hsiao, Y.-C. Lu, P. Vanna-iampikul, A. Agnesina, R. Liang, Y.-H. Lu, H. Ren, and S. K. Lim, "Dco-3d: Differentiable congestion optimization in 3d ics," in *2025 62th ACM/IEEE Design Automation Conference (DAC)*, 2025.

[30] Y.-H. Chung, S. Jiang, W.-L. Lee, Y. Zhang, H. Ren, T.-Y. Ho, and T.-W. Huang, "Simpart: A simple yet effective replication-aided partitioning algorithm for logic simulation on gpu," in *Proceedings of the European Conference on Parallel Processing*, 2025.

[31] Y.-H. Chung, N. Oh, M. G. Balabhadra Naga Venkata, A. Shiledar, S. Kundu, V. Khandelwal, and T.-W. Huang, "Accelerating gate sizing using gpu," in *Proceedings of the European Conference on Parallel Processing*, 2025.

[32] A. Agiza and S. Reda, "Openphysyn: An open-source physical synthesis optimization toolkit," in *2020 Workshop on Open-Source EDA Technology (WOSET)*, 2020.