TREAD: A TRANSFERABLE RTL OPTIMIZER VIA REWRITE TRAJECTORIES AND SYNTHESIS-GROUNDED SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Improving RTL before synthesis is an important way to reduce hardware cost, but effective rewrites depend on both the design structure and the downstream synthesis flow. Manually exploring this space or rediscovering useful transformations for each design is expensive, motivating methods that can learn and transfer optimization experience across RTL designs. We develop TREAD, a transferable RTL optimizer that learns such experience from rewrite trajectories. Imitation learning first provides an executable policy over legal rule–location actions, while offline reinforcement learning captures long-horizon rewrite effects from complete trajectories and delayed returns. To overcome the limited action coverage of observed trajectories, we generate alternative legal rewrites from the same source states and evaluate these unselected actions under a common Design Compiler flow, providing synthesis-grounded preferences for value and policy learning. At inference, TREAD performs bounded residual search beyond the imitation endpoint and accepts a learned residual only when its predicted utility exceeds a positive margin. Across seven RTL benchmark categories under a common technology library and synthesis flow, TREAD achieves the best QoR among the evaluated RTL optimizers under the reported synthesis flow, including representative e-graph-based and LLM-based systems. These results suggest that combining executable behavior, long-horizon returns, and same-state synthesis feedback enables optimization experience to transfer across RTL designs and improve RTL optimization.

1 INTRODUCTION

RTL optimization plays a critical role in modern hardware design flows, as early-stage optimization decisions can significantly affect the quality of synthesized implementations. Small changes to an arithmetic expression, operator arrangement, or datapath structure can alter the netlist selected by the synthesis tool and consequently change area, delay, and their trade-off. Rewrite effects are nevertheless highly design-dependent. A transformation that is useful in one datapath may be neutral in another, and a rewrite that produces little immediate improvement may expose a more favorable sequence of later transformations. An effective RTL optimizer therefore needs to reason about both the structured syntax of the current design and the behavior of the downstream synthesis flow.

Existing RTL optimizers can be broadly divided into two categories. The first uses e-graphs to expand and explore spaces of equivalent RTL implementations. Equality-saturation methods retain multiple datapath expressions and defer implementation selection until extraction, while subsequent work extends this paradigm with mixed-precision rewriting, abstract interpretation, range and constraint analysis, technology-aware cost models, and formal verification Coward et al. (2022; 2024b; 2023b;a; 2024a); Wanna et al. (2023); Coward et al. (2023c). However, efficiently identifying high-quality implementations in this expanded space remains challenging. Static or decomposable cost models make rewrite search efficient, but cannot fully capture the interactions between RTL rewrites and downstream technology-mapped synthesis, creating a gap between estimated and realized QoR. Direct synthesis provides more accurate QoR feedback, but evaluating many e-graph candidates incurs substantial runtime overhead. As a result, the optimization knowledge obtained from synthesis-guided exploration is often used only within individual optimization runs rather than being distilled into a generalizable strategy for other RTL designs.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The second category uses learned models or language models to guide synthesis and RTL optimization. ABC-RL learns Boolean synthesis recipes and uses retrieval to adapt a pretrained policy to unfamiliar circuits Basak Chowdhury et al. (2024). Other methods learn technology-aware quality predictors from e-graphs or mapped circuit representations, improving implementation-cost estimation from higher-level structures Chen et al. (2024); Moravej et al. (2024); Liu et al. (2025b). LLM-based systems provide another form of guidance through retrieved rules, generated edits, equality saturation, symbolic reasoning, or synthesis feedback Yao et al. (2024); Zhang et al. (2025); Wang et al. (2026); Thorat et al. (2025). These methods demonstrate the value of learned optimization behavior, but they have two limitations. First, LLM-based methods operate in an open-ended transformation space, so many generated edits may be incorrect or functionally inequivalent and are filtered out during formal verification, reducing RTL optimization efficiency and code usability. They also primarily learn individual transformations rather than long-horizon rewrite decisions and compositions. Second, RL-based methods model sequential decisions more explicitly, but online or synthesis-in-the-loop learning still requires repeated synthesis to evaluate explored actions Dong et al. (2026). The resulting synthesis experience is typically tied to the current design search rather than distilled into a reusable policy for fine-grained rewrite decisions across new RTL designs.

Taken together, these limitations suggest that the key challenge is not simply to search for better RTL implementations, but to learn optimization decisions that can be reused across designs while remaining grounded in downstream synthesis outcomes. We develop TREAD, a transferable RTL optimizer that learns this experience from rewrite trajectories and synthesis-grounded supervision. RTL rewrites are structured rule–location actions, so an unconstrained learner would spend much of its search budget on illegal or uninformative transformations. TREAD first applies imitation learning to synthesis-validated trajectories whose final designs improve QoR. This gives the policy executable action support and a stable prior over the rewrite space. We further recognized that the outcome of an individual rewrite cannot be judged from the current state alone: its effect also depends on how the optimization trajectory unfolds afterward. TREAD models this long-horizon dependence through trajectory-based value refinement. Complete rewrite trajectories provide suffix returns, while successor-value bootstrapping and observed-return supervision train action-value and state-value functions over the trajectory distribution. Their advantages guide policy updates: actions on successful optimization paths are reinforced, negative-return actions are discouraged through a separate policy objective, and successful behavior is replayed to retain useful preparatory rewrites. This refinement captures delayed effects rather than relying only on immediate rewrite gains. The learned values also let the search rank alternative legal rewrite paths beyond the demonstrated sequence and identify paths with potential for further synthesis gains.

As we examined the rewrite trajectories, we found that each visited state usually records only the selected rule–location action, while other legal actions at the same state remain unobserved. This makes it difficult to compare alternative rewrites using trajectory returns alone. TREAD therefore augments the trajectories with additional legal rewrites generated from the same source states and evaluated under a common Design Compiler flow. The resulting synthesis measurements provide direct, state-conditional feedback about local rewrite choices and complement the long-horizon information in the trajectories. We further use this learned information in a residual optimization procedure. TREAD first follows the IL policy and then searches for improvements beyond its endpoint. A positive acceptance margin is required before the residual trajectory replaces the IL result, allowing the synthesis-aware policy to extend successful behavior while falling back to the IL endpoint when the predicted residual improvement is insufficient. Together, these components turn synthesis-validated rewrite processes into reusable optimization knowledge for new RTL designs.

The contributions of this work are threefold. First, we introduce a trajectory-level formulation of RTL optimization as transferable sequential decision making over structured rule–location rewrites, and present TREAD, which converts synthesis-validated rewrite trajectories into reusable optimization knowledge across designs. Second, we develop a staged learning method that combines imitation-based action support with long-horizon value refinement. Trajectory returns capture delayed rewrite effects, while same-state legal rewrites provide synthesis-grounded preferences for local action comparison. These complementary signals guide value and policy refinement and a gated residual search beyond the demonstrated trajectories. Third, we evaluate TREAD on RTL designs held out from training and validation. Across the evaluated benchmark categories, it achieves the strongest post-synthesis QoR among the representative RTL optimizers and reduces end-to-end optimization time by more than 65% relative to the evaluated synthesis-based e-graph flow.

2 Related Work

2.1 RTL Rewriting and Equality Saturation

Equality saturation retains multiple equivalent RTL expressions in an e-graph and defers implementation selection until extraction. Early work applies this paradigm to arithmetic architectures whose costs depend on operand widths and rewrite interactions Coward et al. (2022). ROVER extends it to mixed-precision RTL with width-sensitive rewrites, rule validation, verification certificates, and ILP-based extraction using an area cost model Coward et al. (2024b); we therefore use ROVER as an e-graph-based baseline. Subsequent work enriches e-graph optimization with abstract interpretation, input-domain constraints, range and delay analysis, multiplier and gating transformations, and word-level verification Coward et al. (2023b;a; 2024a); Wanna et al. (2023); Coward et al. (2024c; 2023c). These extensions improve rewrite expressiveness, cost estimation, or correctness, but candidate extraction still depends on a cost model. Static cost models make extraction efficient but may diverge from the QoR reported by industrial synthesis tools such as Synopsys Design Compiler (DC). Direct DC evaluation is more faithful but substantially more expensive, and the resulting synthesis experience is usually not distilled into a reusable strategy that transfers across designs.

2.2 Learning-Based Synthesis and Quality Estimation

Learning-based synthesis treats optimization as a sequential decision problem whose outcome depends on preceding transformations. ABC-RL learns policies over Boolean synthesis heuristics and uses retrieval to adapt a pretrained policy to unfamiliar circuits Basak Chowdhury et al. (2024). This work highlights that transferring optimization behavior requires both useful policy knowledge and a mechanism for determining when that knowledge should guide exploration. A complementary line learns implementation quality rather than rewrite actions. E-Syn uses technology-aware area and delay predictors for e-graph extraction Chen et al. (2024), while VeriDistill and StructRTL transfer information from mapped circuit representations to HDL- or graph-level quality models Moravej et al. (2024); Liu et al. (2025b). These methods improve cost estimation from higher-level representations, but their targets are implementation-quality predictions rather than reusable policies over fine-grained RTL rewrite actions.

2.3 LLM-Based RTL Generation and Optimization

Language models have been applied to HDL generation through domain-specific training, code augmentation, compiler feedback, and RTL representation learning Thakur et al. (2024); Liu et al. (2024); Zhao et al. (2025); Cui et al. (2024); Liu et al. (2025a;c). Benchmarks such as VerilogEval and RTLLM evaluate generated HDL for syntax, functional correctness, and design quality Liu et al. (2023); Lu et al. (2024). These settings differ from optimizing existing RTL, where functional equivalence must be preserved while improving the synthesized implementation. For this task, RTL-Rewriter combines program analysis, retrieved guidance, cost-aware search, and verification Yao et al. (2024); ASPEN combines language-model reasoning with equality saturation, generated rules, and synthesis feedback Zhang et al. (2025). SymRTLO and LLM-VeriPPA further use retrieved rules, symbolic reasoning, and staged correctness/QoR optimization Wang et al. (2026); Thorat et al. (2025). We use RTLRewriter and ASPEN as representative LLM-based baselines. These methods provide useful design-specific guidance, but their open-ended edits or generated rules may require additional equivalence checking and synthesis-in-the-loop search. The resulting optimization experience is generally tied to each design, rather than distilled into a reusable policy over fine-grained rule–location actions.

3 Methodology

3.1 Overview

RTL rewriting requires legal rule–location actions and must account for synthesis effects that may emerge after several rewrites. TREAD combines trajectory and synthesis supervision to address these complementary needs. It first constructs synthesis-aware trajectories and trains an imitation policy, then refines the policy with trajectory-based value learning to capture delayed effects. Addi-

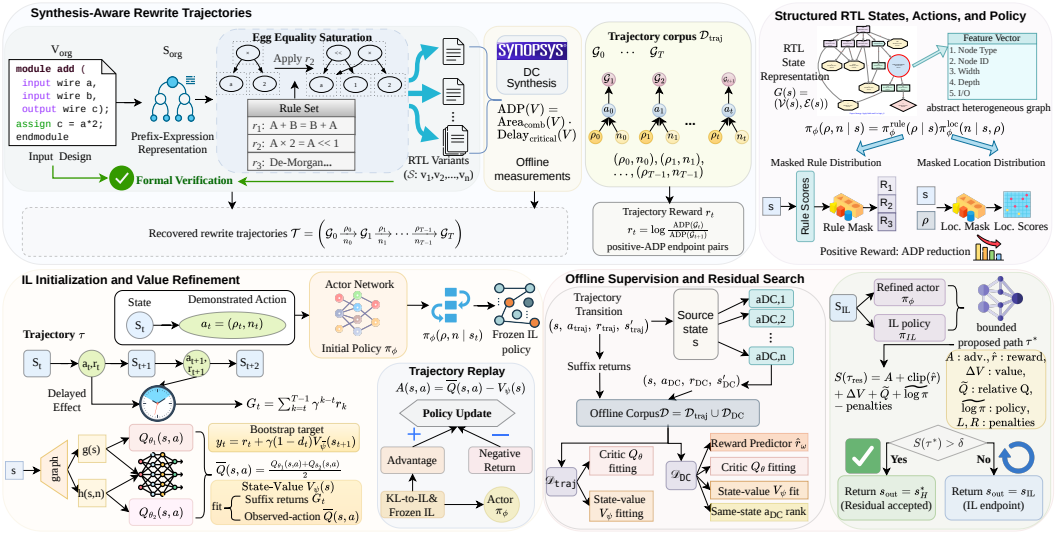


Figure 1: Overview of TREAD. TREAD learns transferable RTL rewrite policies from synthesis-aware trajectories, combining imitation learning, long-horizon value refinement, and same-state Design Compiler supervision. At deployment, bounded residual search extends the IL endpoint under a positive-margin acceptance gate.

tional legal rewrites generated from the same states are synthesized offline under a common Design Compiler flow, providing local preferences for alternative actions. At deployment, TREAD follows the IL policy to an endpoint and performs bounded residual search using the learned policy and value estimates; a residual is accepted only if its predicted utility exceeds a positive margin.

3.2 TREAD Framework

Synthesis-aware rewrite trajectories. Given an input RTL design V_{org}, TREAD first translates it into a prefix-expression representation S_{org} and processes it with the `egg` equality-saturation engine Willsey et al. (2021). The rewrite library contains algebraic identities, structural transformations, Boolean simplifications, constant folding, strength reduction, and arithmetic-shift equivalences. Applying these rules to matching subexpressions produces an e-graph containing equivalent implementations. For an illustrative example of width-preserving e-graph rewriting, please refer to Appendix C. We extract a set of structurally diverse RTL variants

$$\mathcal{S} = \{V_1, V_2, \ldots, V_N\}. \tag{1}$$

Each extracted variant is converted into synthesizable Verilog and checked for functional equivalence with the original RTL using Yosys Wolf et al. (2013). Only verified variants are retained for the offline synthesis measurements. All variants associated with one design are evaluated under the same technology library, timing constraints, and compilation settings. For a valid synthesized variant V, we record its combinational area and critical-path delay and compute

$$\mathrm{ADP}(V) = \mathrm{Area}_{\mathrm{comb}}(V) \cdot \mathrm{Delay}_{\mathrm{critical}}(V). \tag{2}$$

Using `egg`'s equivalence explanations, we recover rewrite sequences connecting the original expression to the evaluated variants:

$$\mathcal{T} = \left(\mathcal{G}_0 \xrightarrow[n_0]{\rho_0} \mathcal{G}_1 \xrightarrow[n_1]{\rho_1} \cdots \xrightarrow[n_{T-1}]{\rho_{T-1}} \mathcal{G}_T \right), \tag{3}$$

where $\mathcal{G}_t$ denotes the RTL state at step t, ρ_t is the applied rewrite rule, and n_t is its target location. Each state is later converted into the graph representation described below, while (ρ_t, n_t) serves as the action label. The trajectory corpus retains improving, neutral, degrading, and delayed-gain steps. This preserves cases in which a rewrite has little immediate effect but enables a favorable

continuation. When valid synthesis measurements are available for two consecutive states, we define the relative reward as

$$r_t = \log \frac{\mathrm{ADP}(\mathcal{G}_t)}{\mathrm{ADP}(\mathcal{G}_{t+1})}. \tag{4}$$

The reward is defined only for valid endpoint pairs with positive ADP. The resulting corpus $\mathcal{D}_{\mathrm{traj}}$ contains the trajectory evidence used for long-horizon credit assignment.

Structured RTL states, actions, and policy. We represent RTL state s as a heterogeneous graph

$$G(s) = (\mathcal{V}(s), \mathcal{E}(s)). \tag{5}$$

Nodes represent operators, signals, constants, and structural RTL constructs. Typed edges encode expression and data dependencies. Each logic node uses a five-dimensional feature vector containing node type, node identifier, bit width, expression depth, and I/O attribute. These node features are combined with typed edges that encode expression and data dependencies. An action is a rule–location pair

$$a = (\rho, n), \tag{6}$$

where ρ is an equivalence-preserving rewrite rule and n is a logic node. Applying action produces

$$s' = f(s, a). \tag{7}$$

The policy factorizes action selection into a rule decision and a rule-conditioned location decision:

$$\pi_\phi(\rho, n \mid s) = \pi_\phi^{\mathrm{rule}}(\rho \mid s)\pi_\phi^{\mathrm{loc}}(n \mid s, \rho). \tag{8}$$

Rule and location masks remove transformations that are illegal for the current graph before the corresponding distributions are normalized. For a detailed description of rewrite-state construction and action-validity checks, please refer to Appendix B. For any valid synthesized state, the transition reward is

$$r(s, a) = \log \frac{\mathrm{ADP}(s)}{\mathrm{ADP}(s')}, \qquad s' = f(s, a), \tag{9}$$

whenever both states have valid positive ADP. A positive reward indicates an ADP reduction.

Imitation learning as an executable prior. The structured action space requires the policy to produce legal rewrites before synthesis outcomes can be used effectively. TREAD therefore initializes the actor from synthesis-validated rewrite trajectories. Given a demonstrated action (ρ_t, n_t) at state s_t, the imitation objective is

$$\mathcal{L}_{\mathrm{IL}} = -\log \pi_\phi^{\mathrm{rule}}(\rho_t \mid s_t) - \log \pi_\phi^{\mathrm{loc}}(n_t \mid s_t, \rho_t). \tag{10}$$

The resulting IL policy provides executable action support and a stable prior over the rewrite space. It also defines the endpoint from which residual optimization is measured. This initialization keeps subsequent learning and search within the legal structured action space.

Trajectory-based value refinement. A rewrite cannot always be judged by its immediate reward because its value may depend on subsequent transformations. For a trajectory

$$\tau = (s_0, a_0, r_0, \ldots, a_{T-1}, r_{T-1}, s_T), \tag{11}$$

we compute the suffix return

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k. \tag{12}$$

This allows an early rewrite to receive credit for improvements that appear only after later transformations. The graph encoder produces a state embedding $g(s)$ and a contextual location embedding $h(s, n)$. TREAD learns two independently parameterized action-value functions Q_{θ_1} and Q_{θ_2}, a state-value function V_ψ, and a transition-reward predictor $\widehat{r}_\omega$. For a transition $(s_t, a_t, r_t, s_{t+1}, d_t)$, the bootstrap target is

$$y_t = r_t + \gamma(1 - d_t)V_{\bar{\psi}}(s_{t+1}), \tag{13}$$

where $V_{\bar{\psi}}$ is a slowly updated target value function. The two action-value functions are trained against this target, and their mean is

$$\overline{Q}(s,a) = \frac{Q_{\theta_1}(s,a) + Q_{\theta_2}(s,a)}{2}. \tag{14}$$

In addition to the bootstrap target, $\overline{Q}$ is anchored to recorded suffix returns and constrained to preserve useful ordering between actions from the same source state. The state-value function is fitted to observed returns and to the value distribution induced by the observed actions. A conservative candidate-set penalty discourages unsupported high-value estimates for legal alternatives that are insufficiently represented in the offline data. The actor is refined using the advantage

$$A(s,a) = \overline{Q}(s,a) - V_{\psi}(s). \tag{15}$$

Advantage-weighted updates reinforce actions supported by favorable long-horizon evidence, while negative-return actions are discouraged through a separate policy objective. Successful trajectory behavior is replayed to retain useful preparatory rewrites, and a KL penalty to the frozen IL policy limits unnecessary drift from the executable prior. This value refinement does not assume that the refined policy must outperform IL on every design. Its role is to estimate delayed consequences and provide additional guidance for residual search beyond the IL endpoint.

Same-state synthesis supervision. The trajectory corpus records the selected action at each visited state, but does not generally contain the measured outcomes of other legal actions available at that state. This creates a coverage gap for learning how alternative rewrites from the same RTL state affect synthesis quality. TREAD addresses this gap by generating additional legal rewrites offline from source states sampled from the trajectory corpus. We enumerate legal rule–location pairs, apply the corresponding rewrites, and remove duplicate successor expressions. Structural statistics such as changes in expression size, depth, and operator composition allocate the candidate budget; they do not define the measured reward. For each retained candidate, matched synthesis measurements are obtained for the source and rewritten RTL under the same Design Compiler configuration. A candidate pair is retained only when both endpoints have valid positive ADP. Its immediate reward is computed using Eq. equation 9 and stored as a one-step transition record (s, a, r, s') without an observed continuation. Thus, the record ends after the measured rewrite for training purposes; this does not imply that s' is a terminal RTL state. The successor may still admit further rewrites, but their future value is not observed in this DC record. The complete offline corpus is

$$\mathcal{D} = \mathcal{D}_{\mathrm{traj}} \cup \mathcal{D}_{\mathrm{DC}}. \tag{16}$$

Trajectory samples provide successor information and suffix returns for long-horizon value learning, whereas same-state DC samples provide measured immediate rewards and action comparisons for alternative rewrites from the same source RTL. The two sources therefore provide complementary supervision: trajectory data describe how rewrite consequences unfold, while DC measurements supply local synthesis evidence for actions that were not selected in the recorded trajectory. The transition-reward predictor estimates the immediate synthesis effect:

$$\widehat{r}_{\omega} = \widehat{r}_{\omega}\big(g(s), g(s'), h(s,n), e_{\rho}\big), \tag{17}$$

where e_{ρ} is a learned rule embedding. In contrast to the reward head, which predicts the immediate effect of one rewrite, Q_{θ_i} estimates the value of taking that rewrite and continuing from the successor. This separation allows an action with weak immediate reward to remain useful when its continuation is favorable. Trajectory successors supervise long-horizon continuation value, while same-state DC candidates provide counterfactual supervision for immediate synthesis improvement and local candidate ordering. DC-only candidates are therefore used for reward prediction and same-state ranking rather than as complete Bellman transitions. Their use in policy refinement is controlled by the corresponding policy-learning masks. The DC measurements are collected offline and provide synthesis-grounded supervision during training, rather than serving as an online evaluator for every candidate during deployment. For a detailed description of the training objectives and the roles of trajectory and same-state DC supervision, please refer to Appendix D.

Residual search and conservative deployment. For a held-out design, TREAD first follows the IL policy and obtains an endpoint s_{IL}. Residual search starts from this endpoint with score zero

and considers proposals from both the refined actor and the IL policy, retaining useful demonstrated actions as candidates. For a residual trajectory $\tau_{\mathrm{res}} = (s_0, a_0, \ldots, s_H)$ with $s_0 = s_{\mathrm{IL}}$, TREAD uses the following score to rank candidate paths:

$$S(\tau_{\mathrm{res}}) = \sum_{t=0}^{H-1} \eta^t \big[\lambda_A A(s_t, a_t) + \lambda_r \kappa^t \,\mathrm{clip}(\widehat{r}_t) + \lambda_V \big(V(s_{t+1}) - V(s_t)\big) + \lambda_Q \widetilde{Q}_t + \lambda_\pi \widetilde{\log \pi}_t - \lambda_L - \lambda_R R_t\big]. \tag{18}$$

The relative Q and centered log-policy terms compare proposals at the same state, while reward clipping and depth decay limit the effect of prediction errors. The search also filters unchanged rewrites, short cycles, duplicate states, and excessive runs of neutral rules. Search limits and scoring coefficients follow the experimental configuration. Let τ^* be the highest-scoring residual path. TREAD returns

$$s_{\mathrm{out}} = \begin{cases} s_H^*, & S(\tau^*) > \delta, \\ s_{\mathrm{IL}}, & \text{otherwise}, \end{cases} \qquad \delta > 0. \tag{19}$$

This gate retains the IL endpoint when the best residual path lacks sufficient predicted utility. It is a model-based acceptance criterion; the final hardware quality is measured independently under the evaluation synthesis flow. For the complete residual-search budget and inference configuration, please refer to Appendix G.

4 Experiments

4.1 Experimental Settings

We evaluate TREAD on seven unseen RTL categories from the public DeepCircuitX dataset Li et al. (2025), covering arithmetic, logic, shifting, filtering, and processor designs. All evaluation designs are held out from training and validation to measure zero-shot transfer. Key configurations follow. **Training.** The batch size is 8, with random seed 42, discount factor $\gamma = 1.0$, learning rate 10^{-4}, and actor learning-rate scale 0.1. We use five value-warmup epochs followed by 20 RL epochs. The dataset split is design-disjoint, with 10% of trajectory groups reserved for validation. For additional training curves and convergence diagnostics, please refer to Appendix 4. **Inference.** IL performs at most 8 rewrite steps. The RL residual search performs at most 7 additional steps with beam width 5. At each state, it considers the top 4 rewrite rules and top 4 candidate locations. A residual path is accepted only when its predicted residual score exceeds the margin $\delta = 0.05$. **Synthesis and evaluation.** All candidates are synthesized using Synopsys Design Compiler with a TSMC 28 nm standard-cell technology library and the `compile_ultra` command. For sequential designs, we impose a 20 ns clock constraint with zero input/output delays and output load 1.0, while designs without a clock use a 20 ns input-to-output maximum-delay constraint. A candidate is valid only if it successfully parses, elaborates, links, and synthesizes, and produces positive area, delay, and ADP. For a detailed description of the synthesis flow and validity criteria, please refer to Appendix H. **Runtime accounting.** Reported runtime includes test-time inference, rewrite generation, RTL conversion, and final synthesis. Trajectory collection, counterfactual Design Compiler labeling, and model training are excluded as offline costs. For the complete dataset and training configuration, please refer to Appendix F.

4.2 Experimental Results

4.2.1 Optimization Quality

All QoR results are aggregated over valid held-out test designs whose benchmark categories are disjoint from the training and validation sets. For category-level aggregation, each design is normalized by its original ADP and the ratios are aggregated using a geometric mean. Lower normalized ADP and higher ΔADP indicate better QoR. For the normalization and category-level aggregation procedure, please refer to Appendix I.1. Table 1 shows that TREAD reduces post-synthesis ADP across all evaluated categories, with the largest gains on MCM, shift register, MUX, and ALU. The observed reductions are largely associated with delay reductions, while area changes are comparatively smaller. TREAD also improves on the external baselines in categories where effective optimization involves sequences of dependent transformations. ROVER and ASPEN select

Table 1: Post-synthesis QoR comparison across seven held-out RTL benchmark categories under a common TSMC 28 nm Design Compiler flow. Area, delay, and ADP are reported in $\mu\mathrm{m}^2$, ns, and $\mu\mathrm{m}^2 \cdot \mathrm{ns}$, respectively.

(a) Comparison with the e-graph-based RTL optimizers ROVER and ASPEN.

Benchmark Category	Original			ROVER				ASPEN			
	Area	Delay	ADP	Area	Delay	ADP	ΔADP	Area	Delay	ADP	ΔADP
FMA	4282.61	4.28	18329.59	4269.13	4.28	18271.88	0.32%	4324.07	4.06	17555.71	4.22%
MCM	6828.70	5.06	34553.20	6650.03	4.56	30324.13	12.24%	6828.70	5.06	34553.20	0.00%
Shift Register	905.44	0.71	642.86	905.44	0.71	642.86	0.00%	905.44	0.71	642.86	0.00%
MUX	4416.93	3.63	16033.46	4553.01	2.65	12065.48	24.75%	4553.01	2.65	12065.48	24.75%
ALU	1267.31	2.23	2826.10	1269.45	2.11	2678.54	5.22%	1186.67	2.13	2527.60	10.56%
FIR_Filter	36953.91	1.80	66517.04	36945.85	1.77	65394.15	1.69%	36956.05	1.80	66520.89	-0.01%
MIPS_CPU	267004.32	6.51	1738198.19	266661.61	6.47	1725300.61	0.74%	266407.09	6.51	1734310.15	0.22%

(b) Comparison with RTLRewriter and the proposed TREAD framework.

Benchmark Category	Original			RTLRewriter				TREAD			
	Area	Delay	ADP	Area	Delay	ADP	ΔADP	Area	Delay	ADP	ΔADP
FMA	4282.61	4.28	18329.59	4269.13	4.28	18271.88	0.32%	4317.14	4.06	17527.58	**4.38%**
MCM	6828.70	5.06	34553.20	6839.78	2.68	18330.62	46.95%	6657.46	2.68	17841.99	**48.37%**
Shift Register	905.44	0.71	642.86	905.31	0.71	642.77	0.01%	907.83	0.38	344.98	**46.34%**
MUX	4416.93	3.63	16033.46	4563.85	2.57	11729.08	26.86%	4564.85	2.48	11320.84	**29.39%**
ALU	1267.31	2.23	2826.10	1267.31	2.23	2826.10	0.00%	1184.78	2.08	2464.34	**12.80%**
FIR_Filter	36953.91	1.80	66517.04	36945.85	1.77	65394.15	1.69%	36943.33	1.71	63173.09	**5.03%**
MIPS_CPU	267004.32	6.51	1738198.19	266407.09	6.51	1734310.15	0.22%	265338.39	6.47	1716738.78	**1.23%**

Table 2: Test-time runtime comparison between bounded e-graph search and TREAD across seven RTL benchmark categories. Offline trajectory collection, counterfactual DC labeling, and model training are excluded. Runtime reduction is relative to the bounded e-graph flow.

Benchmark Category	Bounded E-Graph Optimization Time (s)					TREAD Optimization Time (s)							Runtime Reduction
	$V_{\mathrm{org}} \to S_{\mathrm{org}}$	Rule Rewrite	Candidate Generation	Candidate Synthesis	E-Graph Total	$V_{\mathrm{org}} \to S_{\mathrm{org}}$	Graph Encoding	IL Inference and Trajectory	RL Inference and Trajectory	$S_{\mathrm{TREAD}} \to V_{\mathrm{TREAD}}$	Final Synthesis	TREAD Total	
ALU	1.33	1.92	19.42	218.72	241.39	0.35	10.67	0.31	20.72	0.43	15.08	47.56	**80.30%**↓
FMA	1.12	1.16	16.50	675.08	693.86	0.32	8.64	1.60	35.94	0.40	37.96	84.86	**87.77%**↓
Shifter Register	1.44	1.99	22.01	373.51	398.95	0.18	6.10	1.14	88.36	0.32	11.06	107.16	**73.14%**↓
MCM	1.15	1.49	22.19	1513.88	1538.71	0.15	5.67	2.51	170.07	0.44	64.56	243.40	**84.18%**↓
MUX	0.91	1.35	16.66	634.48	653.40	0.15	6.39	2.21	141.78	0.36	35.26	186.15	**71.51%**↓
FIR Filter	1.34	21.63	24.95	5118.77	5166.69	0.37	87.44	32.82	820.72	21.39	335.31	1298.05	**74.88%**↓
MIPS CPU	1.05	37.27	50.22	7993.75	8082.29	0.62	364.87	54.78	1624.58	34.24	632.20	2711.29	**66.45%**↓

equivalent expressions through design-specific e-graph exploration Coward et al. (2024b); Zhang et al. (2025), while RTLRewriter searches generated edits for the current design Yao et al. (2024). TREAD instead combines an IL policy for legal action selection, trajectory-trained values for continuation-aware credit assignment, and same-state DC supervision for local action preferences. Its residual search is designed to extend the IL endpoint when later transformations are predicted to be beneficial, rather than selecting only the best immediate rewrite. The stronger gains on MUX and MCM are consistent with the hypothesis that continuation-aware search is particularly useful when effective improvements require dependent rewrite sequences. The smaller improvements on FMA and FIR may reflect the limited residual optimization headroom after synthesis has already optimized their original structures. Similarly, the more modest gain on MIPS may reflect the influence of control logic, sequential state, and the behavioral SRAM model on its synthesized QoR. Since the same trained policy is applied without design-specific retraining to categories excluded from training and validation, these results provide evidence of transfer across unseen RTL structures.

4.2.2 Optimization Efficiency

Runtime is measured on the same held-out benchmark designs, without per-design retraining. Table 2 compares the end-to-end test-time runtime of TREAD with bounded e-graph search. The bounded e-graph flow includes rewrite exploration, extraction, synthesis of 20 candidate variants, and selection of the best result. This 20-variant budget provides a practical trade-off between coverage of the equivalence space and synthesis cost, and serves as the per-design synthesis budget. TREAD includes graph construction, IL and residual policy inference, RTL reconstruction, and one final synthesis. TREAD's efficiency advantage over the conventional e-graph-and-synthesis optimization flow comes primarily from avoiding repeated synthesis of intermediate candidates for QoR evaluation. TREAD reduces total optimization time by 66.45%–87.77% across all seven benchmark

Table 3: Per-category ablation of TREAD components. Each entry reports the geometric mean of ADP normalized to the corresponding original RTL and ADP reduction. The Geo. Mean is a macro-average over seven benchmark categories, with each category weighted equally.

Benchmark Category	IL		IL + Residual		IL + RL		IL + DC		TREAD	
	Norm. ADP	ΔADP	Norm. ADP	ΔADP	Norm. ADP	ΔADP	Norm. ADP	ΔADP	Norm. ADP	ΔADP
FMA	0.9968	0.32%	0.9852	1.48%	0.9610	3.90%	0.9780	2.20%	**0.9562**	**4.38%**
MCM	0.9520	4.80%	0.8840	11.60%	0.5420	45.80%	0.6120	38.80%	**0.5163**	**48.37%**
Shift Register	0.9999	0.01%	0.9820	1.80%	0.5610	43.90%	0.8250	17.50%	**0.5366**	**46.34%**
MUX	0.7525	24.75%	0.7320	26.80%	0.7180	28.20%	0.7250	27.50%	**0.7061**	**29.39%**
ALU	0.9820	1.80%	0.9478	5.22%	0.8950	10.50%	0.9120	8.80%	**0.8720**	**12.80%**
FIR Filter	0.9983	0.17%	0.9831	1.69%	0.9580	4.20%	0.9670	3.30%	**0.9497**	**5.03%**
MIPS CPU	0.9978	0.22%	0.9940	0.60%	0.9890	1.10%	0.9920	0.80%	**0.9877**	**1.23%**
Geo. Mean	**0.9515**	**4.85%**	**0.9272**	**7.28%**	**0.7891**	**21.09%**	**0.8443**	**15.57%**	**0.7750**	**22.50%**

categories. Graph construction and model inference introduce additional overhead, but remain substantially cheaper than repeated synthesis, which dominates the e-graph runtime. The advantage is also maintained for larger designs such as FIR Filter and MIPS CPU, where candidate synthesis is especially expensive. The trajectory-generation, counterfactual DC-labeling, and model-training costs are incurred offline and amortized across designs. Once trained, the same models are reused for new RTL inputs without per-design retraining. TREAD therefore improves optimization efficiency while improving RTL quality.

4.2.3 Component Ablation

Contribution of each TREAD component across the seven held-out benchmark categories. For the definitions of the controlled component variants, please refer to Appendix I. IL denotes the imitation-learned rewrite sequence, while the other configurations incrementally add residual search, RL value guidance, and same-state DC supervision. For each category and configuration, only designs that successfully parse, elaborate, link, and synthesize with positive ADP are included; missing or invalid results are excluded. The number of valid designs is 21 for ALU, FIR, MCM, shifter, and MUX, and 20 for FMA. For MIPS, IL has 13 valid designs, while the other component variants each have 10. We report category-level normalized ADP and the corresponding ADP reduction; lower normalized ADP and higher positive ΔADP indicate better QoR. For the normalization and aggregation procedure, please refer to Appendix I.1.

The ablation results show that the different components provide complementary contributions to the final QoR. IL already provides useful improvements, while residual search further extends the IL endpoint when additional rewrites remain beneficial. RL provides substantial additional gains on MCM, shift registers, ALU, and FIR, where effective improvements may require longer transformation sequences and delayed reward signals. Same-state DC supervision provides complementary local synthesis-aware guidance, but does not replace trajectory-based search. Combining all components, TREAD achieves the lowest Norm. ADP and the largest ADP reduction relative to the baseline in every category. The gains are largest for MCM, shift registers, MUX, and ALU, where source-level structural changes have a larger impact on mapped QoR, while they are smaller for FMA, FIR, and MIPS, where solid baseline synthesis quality and timing-dominated QoR leave less room for further optimization. These results support the complementary roles of policy initialization, residual exploration, long-horizon value estimation, and synthesis-grounded supervision.

5 Conclusion

We presented TREAD, a transferable RTL optimizer that distills synthesis-aware rewrite trajectories into reusable optimization policies. By combining imitation-based policy initialization, long-horizon value refinement, same-state Design Compiler supervision, and gated residual search, TREAD captures both delayed rewrite effects and local synthesis preferences while remaining applicable across designs. On seven held-out RTL benchmark categories, TREAD consistently improves post-synthesis ADP and reduces test-time optimization cost by 66.45%–87.77% relative to bounded e-graph search. These results demonstrate that synthesis-grounded trajectory learning can transfer effective RTL optimization experience to unseen designs.

AI USE STATEMENT

We used generative AI tools to assist with drafting and editing the text of this manuscript, improve readability, format LaTeX, prepare the bibliography, and assist with software development, including code review and the identification of potential implementation bugs. We also used generative AI tools to help identify potentially relevant related work; all such references were subsequently read and verified by the authors against their primary sources.

We did not use generative AI tools to formulate the research idea, design the TREAD optimizer, define the rewrite trajectories, design the training stages or learning objectives, generate training or evaluation data, or analyze or interpret the experimental results. Every number reported in this paper comes from our own experiments and evaluation pipeline.

The authors reviewed all AI-assisted work. The technical descriptions, algorithmic definitions, experimental protocols, figures, tables, and implementation changes were checked by the authors for correctness and consistency. AI-assisted code was inspected, tested, and verified against the reported synthesis and evaluation procedures. We take full responsibility for the final content of this paper, including all text, claims, results, references, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The main paper specifies the TREAD formulation, structured RTL states and rule-location actions, imitation-learning objective, trajectory-based value refinement, synthesis-grounded supervision, residual search procedure, and evaluation metrics. Appendix B describes the rewrite rules and their validity conditions. Appendix C provides the model architecture and training objectives, including the training schedule, optimization settings, discount factor, random seed, and learning rates. Appendix D gives the inference and residual-search configuration, including the search depths, beam width, candidate-selection limits, and acceptance margin. Appendix E documents the dataset construction and the design-disjoint training, validation, and test organization. Appendix F reports additional training and ablation analyses. Appendix G specifies the synthesis and evaluation protocol, including the Design Compiler flow, technology library, timing constraints, validity checks, ADP computation, aggregation procedure, and runtime accounting.

The supplementary materials contain the implementation, configuration files, rewrite rules, held-out benchmark organization, training checkpoints, and scripts for rollout and Design Compiler evaluation. Together, these materials provide the information and artifacts needed to reproduce the reported optimization, synthesis, ablation, and runtime results.

REFERENCES

Animesh Basak Chowdhury, Marco Romanelli, Benjamin Tan, Ramesh Karri, and Siddharth Garg. Retrieval-guided reinforcement learning for boolean circuit minimization. In *International Conference on Learning Representations*, volume 2024, pp. 35039–35060, 2024.

Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.

Samuel Coward, George A Constantinides, and Theo Drane. Automatic datapath optimization using e-graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pp. 43–50. IEEE, 2022.

Samuel Coward, George A Constantinides, and Theo Drane. Automating constraint-aware datapath optimization using e-graphs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2023a.

Samuel Coward, George A Constantinides, and Theo Drane. Combining e-graphs with abstract interpretation. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pp. 1–7, 2023b.

Samuel Coward, Emiliano Morini, Bryan Tan, Theo Drane, and George A Constantinides. Datapath verification via word-level e-graph rewriting. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 92–100. IEEE, 2023c.

Samuel Coward, Theo Drane, and George A Constantinides. Constraint-aware e-graph rewriting for hardware performance optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1366–1379, 2024a.

Samuel Coward, Theo Drane, and George A Constantinides. Rover: Rtl optimization via verified e-graph rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(12):4687–4700, 2024b.

Samuel Coward, Theo Drane, Emiliano Morini, and George A Constantinides. Combining power and arithmetic optimization via datapath rewriting. In *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*, pp. 24–31. IEEE, 2024c.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024.

Guande Dong, Hongtao Cheng, Jianwang Zhai, Xiao Yang, Chuan Shi, and Kang Zhao. Orl-lo: Offline reinforcement learning for pretraining and finetuning in logic optimization. *ACM Transactions on Design Automation of Electronic Systems*, 31(6):1–24, 2026.

Zeju Li, Changran Xu, Zhengyuan Shi, Zedong Peng, Yi Liu, Yunhao Zhou, Lingfeng Zhou, Chengyu Ma, Jianyuan Zhong, Xi Wang, et al. Deepcircuitx: A comprehensive repository-level dataset for rtl code understanding, generation, and ppa analysis. *arXiv preprint arXiv:2502.18297*, 2025.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461, 2024.

Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. Deeprtl: Bridging verilog understanding and generation with a unified representation model. *arXiv preprint arXiv:2502.15832*, 2025a.

Yi Liu, Hongji Zhang, Yiwen Wang, Dimitris Tsaras, Lei Chen, Mingxuan Yuan, and Qiang Xu. Beyond tokens: Enhancing rtl quality estimation via structural graph learning. *arXiv preprint arXiv:2508.18730*, 2025b.

Yi Liu, Hongji Zhang, Yunhao Zhou, Zhengyuan Shi, Changran Xu, and Qiang Xu. Deeprtl2: A versatile model for rtl-related tasks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6485–6500, 2025c.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727. IEEE, 2024.

Reza Moravej, Saurabh Bodhe, Zhanguang Zhang, Didier Chetelat, Dimitrios Tsaras, Yingxue Zhang, Hui-Ling Zhen, Jianye Hao, and Mingxuan Yuan. The graph's apprentice: Teaching an llm low level knowledge for circuit quality estimation. *arXiv preprint arXiv:2411.00843*, 2024.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024.

Kiran Thorat, Jiahui Zhao, Yaotian Liu, Amit Hasan, Hongwu Peng, Xi Xie, Bin Lei, and Caiwen Ding. Llm-verippa: Power, performance, and area optimization aware verilog code generation with large language models. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, pp. 1–7. IEEE, 2025.

Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Bowei Tian, Shwai He, Guoheng Sun, Zheyu Shen, Sihan Chen, et al. Symrtlo: Enhancing rtl code optimization with llms and neuron-inspired symbolic reasoning. *Advances in Neural Information Processing Systems*, 38: 50093–50118, 2026.

Andy Wanna, Samuel Coward, Theo Drane, George A Constantinides, and Miloš D Ercegovac. Multiplier optimization via e-graph rewriting. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*, pp. 1528–1533. IEEE, 2023.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145/3434304.

Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, pp. 1–6, 2013.

Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. Rtlrewriter: Methodologies for large models aided rtl code optimization. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–7, 2024.

Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. Aspen: Llm-guided e-graph rewriting for rtl datapath optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, pp. 1–9. IEEE, 2025.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, et al. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

A Appendix

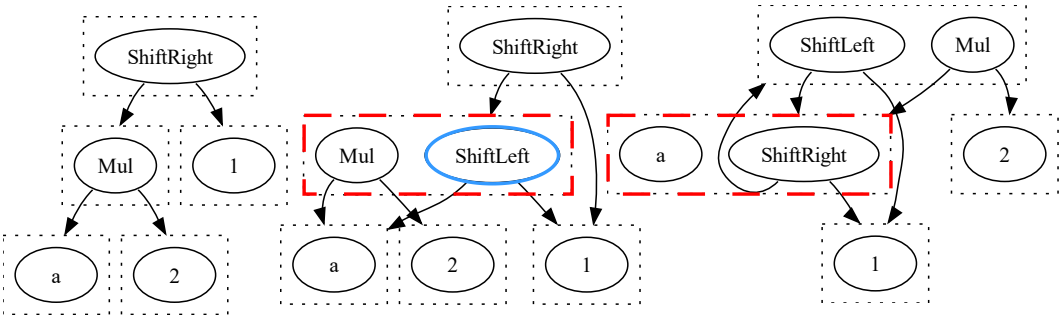
B Rewrite States and Action Validity

Each input RTL design is parsed into a typed expression representation before rewrite search. A state stores both the expression and the provenance of the rewrite that produced it. An action is a pair (ρ, n), where ρ is a rule and n is a node selected from the current expression. We first mask nodes whose operator type, operand order, width, or local parent structure is incompatible with ρ. The rule logits are masked by the set of rules having at least one legal node, and the node logits are masked again after a rule is selected. Thus every action passed to the rewrite engine is legal by construction. We reject an action if it produces an unchanged expression, a previously visited state, or an invalid expression. This validity filtering is applied identically during trajectory construction and deployment. Data splits are formed before training and are kept disjoint at the design level.

C Rewrite Mechanism

E-graph rewriting represents equivalent RTL expressions by e-nodes grouped into e-classes. A rewrite inserts a new e-node into the e-graph and merges it with the e-class of the matched expression when the rule preconditions hold. This representation preserves multiple structurally distinct expressions until extraction, allowing the optimizer to defer the final implementation choice while retaining a bounded rewrite space.

Figure 2 illustrates this process under explicit width and signedness conditions. Let a be an unsigned w-bit value, and assume that the intermediate multiplication and shift expressions use at least $w+1$ bits so that the left shift does not discard any significant bit. Figure 2a shows the initial expression $(a \times 2) \gg 1$. In Fig. 2b, the width-preserving strength-reduction rule $a \times 2 \rightarrow a \ll 1$ introduces an alternative representation into the same e-class. In Fig. 2c, the rule $(a \ll 1) \gg 1 \rightarrow a$ merges the corresponding e-classes because the widened intermediate representation prevents information

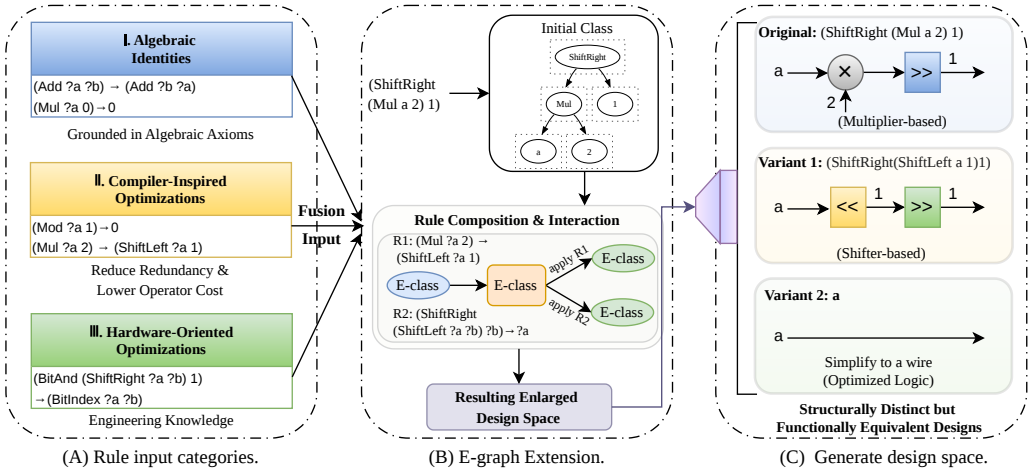


(a) Initial e-graph contains (ShiftRight (Mul a 2) 1)

(b) Apply (Mul ?a 2) → (ShiftLeft ?a 1)

(c) Apply (ShiftRight (ShiftLeft ?a ?b) ?b) → ?a

Figure 2: E-graph construction for a width-preserving strength reduction and subsequent cancellation. E-graph rewriting until equality saturation. Dashed boxes indicate equivalence classes. Blue nodes are newly added terms, red dashed boxes denote modified e-classes. Equality saturation progressively merges rewrite trajectories and expression nodes into shared e-classes.



(A) Rule input categories.

(B) E-graph Extension.

(C) Generate design space.

Figure 3: Rewriting-driven design space expansion via rule–e-graph synergy, illustrated through the example $(ShiftRight\ (Mul\ a\ 2)\ 1)$. The expression corresponds to $(a \times 2) \gg 1$, where $ShiftRight$ and $ShiftLeft$ denote right and left shift operations ($\gg$ and $\ll$), respectively, and ? represents any subgraph.

loss. The expression a is consequently represented in the same e-class as the original expression and can be selected during extraction. Compared with open-ended code generation, e-graph rewriting explores a constrained expression space through predefined rewrite rules. Width- or sign-sensitive rules are enabled only when their preconditions are satisfied, providing a reliable mechanism for preserving RTL functionality.

Figure 3 illustrates the interaction of multiple rules using $(\mathsf{ShiftRight}\ (\mathsf{Mul}\ a\ 2)\ 1)$. Under the required width and signedness conditions, strength reduction first produces $(\mathsf{ShiftRight}\ (\mathsf{ShiftLeft}\ a\ 1)\ 1)$, after which a cancellation rule may recover a. The e-graph retains these expressions within the represented equivalence space, from which TREAD extracts structurally distinct candidates under its bounded sampling budget. Candidates that pass the evaluation-flow checks are then admitted to synthesis and quality-based selection.

The complete set of rules we applied during the rewriting process is shown in Table 4.

Table 4: Rewrite rules.

Class	Rewrite Rule	Instruction
Logical Operations	(And ?a (Or ?a ?b)) → ?a	$A \wedge (A \vee B) = A$
	(And ?b (Or ?a ?b)) → ?b	$B \wedge (A \vee B) = B$
	(Or ?a (And ?a ?b)) → ?a	$A \vee (A \wedge B) = A$
	(Or ?b (And ?a ?b)) → ?b	$B \vee (A \wedge B) = B$
	(Or ?a ?a) → ?a	$A \vee A = A$
	(And ?a ?a) → ?a	$A \wedge A = A$
	(Not (Not ?a)) → ?a	$\neg(\neg A) = A$
	(Or ?a ?b) → (Or ?b ?a)	$A \vee B = B \vee A$
	(And ?a ?b) → (And ?b ?a)	$A \wedge B = B \wedge A$
	(Or ?a (Not ?a)) → true	$A \vee \neg A = true$
	(Or ?a true) → true	$A \vee true = true$
	(Or ?a false) → ?a	$A \vee false = A$
	(And ?a true) → ?a	$A \wedge true = A$
	(And ?a false) → false	$A \wedge false = false$
	(And (Not ?a) ?a) → false	$\neg A \wedge A = false$
Arithmetic Operations	(Add ?a ?b) → (Add ?b ?a)	$A + B = B + A$
	(Add ?a (Add ?b ?c)) → (Add (Add ?a ?c) ?b)	$A + (B + C) = (A + C) + B$
	(Mul ?a ?b) → (Mul ?b ?a)	$A \times B = B \times A$
	(Mul ?a (Mul ?b ?c)) → (Mul (Mul ?a ?b) ?c)	$A \times (B \times C) = (A \times B) \times C$
	(Add (Mul ?a ?b) (Mul ?a ?c)) → (Mul ?a (Add ?b ?c))	$A \times B + A \times C = A \times (B + C)$
	(Add (Mul ?a ?c) (Mul ?b ?c)) → (Mul ?c (Add ?a ?b))	$A \times C + B \times C = C \times (A + B)$
	(Add ?a (Sub (Atom (Int 0)) ?b)) → (Sub ?a ?b)	$A + (0 - B) = A - B$
	(Div (Div ?a ?b) ?c) → (Div (Div ?a ?c) ?b)	$(A \div B) \div C = (A \div C) \div B$
	(Div ?a ?b) → (Mul ?a (Div (Atom (Int 1)) ?b))	$A \div B = A \times (1 \div B)$
	(Div ?a (Mul ?b ?c)) → (Div (Div ?a ?b) ?c)	$A \div (B \times C) = (A \div B) \div C$
	(Mod (Add ?a ?b) ?c) → (Mod (Add (Mod ?a ?c) (Mod ?b ?c)) ?c)	$(A + B) \mod C = ((A \mod C) + (B \mod C)) \mod C$
	(Mod (Sub ?a ?b) ?c) → (Mod (Sub (Mod ?a ?c) (Mod ?b ?c)) ?c)	$(A - B) \mod C = ((A \mod C) - (B \mod C)) \mod C$
	(Mod (Mul ?a ?b) ?c) → (Mod (Mul (Mod ?a ?c) (Mod ?b ?c)) ?c)	$(A \times B) \mod C = ((A \mod C) \times (B \mod C)) \mod C$
	(Mod (Power ?a ?b) ?c) → (Mod (Power (Mod ?a ?c) (Mod ?b ?c)) ?c)	$(A^B) \mod C = ((A \mod C)^{B \mod C}) \mod C$
	(Mod (Add (Mod (Add ?a ?b) ?c) ?d) ?c) → (Mod (Add ?a (Mod (Add ?b ?d) ?c) ?c))	$(((A + B) \mod C) + D) \mod C = (A + ((B + D) \mod C)) \mod C$
	(Mod (Mul (Mod (Mul ?a ?b) ?c) ?d) ?c) → (Mod (Mul ?a (Mod (Mul ?b ?d) ?c) ?c))	$(((A \times B) \mod C) \times D) \mod C = (A \times ((B \times D) \mod C)) \mod C$
	(Mod (Mul (Mod (Add ?a ?b) ?c) ?d) ?c) → (Mod (Add (Mod (Mul ?a ?d) ?c) (Mod (Mul ?b ?d) ?c) ?c))	$((A + B) \mod C \times D) \mod C = ((A \times D \mod C) + (B \times D \mod C)) \mod C$
	(Add ?a ?a) → (Mul ?a (Atom (Int 2)))	$A + A = A \times 2$
	(Mul ?a ?a) → (Power ?a (Atom (Int 2)))	$A \times A = A^2$
	(Add ?a (Atom (Int 0))) → ?a	$A + 0 = A$
	(Sub ?a (Atom (Int 0))) → ?a	$A - 0 = A$
	(Sub (Atom (Int 0)) ?a) → (Subone ?a)	$0 - A = -A$
	(Mul ?a (Atom (Int 0))) → (Atom (Int 0))	$A \times 0 = 0$
	(Mul ?a (Atom (Int 1))) → ?a	$A \times 1 = A$
	(Div (Atom (Int 0)) ?a) → (Atom (Int 0))	$0 \div A = 0$
	(Div ?a (Atom (Int 1))) → ?a	$A \div 1 = A$
	(Mod ?a (Atom (Int 1))) → (Atom (Int 0))	$A \mod 1 = 0$
	(Power ?a (Atom (Int 0))) → (Atom (Int 1))	$A^0 = 1$
	(Power ?a (Atom (Int 1))) → ?a	$A^1 = A$
Bitwise Operations	(BitAnd ?a (BitOr ?a ?b)) → ?a	$A \& (A \mid B) = A$
	(BitAnd ?b (BitOr ?a ?b)) → ?b	$B \& (A \mid B) = B$
	(BitOr ?a (BitAnd ?a ?b)) → ?a	$A \mid (A \& B) = A$
	(BitOr ?b (BitAnd ?a ?b)) → ?b	$B \mid (A \& B) = B$
	(BitOr ?a ?a) → ?a	$A \mid A = A$
	(BitAnd ?a ?a) → ?a	$A \& A = A$
	(BitOr ?a ?b) → (BitOr ?b ?a)	$A \mid B = B \mid A$
	(BitAnd ?a ?b) → (BitAnd ?b ?a)	$A \& B = B \& A$
	(BitXor ?a ?b) → (BitXor ?b ?a)	$A \oplus B = B \oplus A$
Associative/ Distributive	(BitOr (BitOr ?a ?b) ?c) → (BitOr ?a (BitOr ?b ?c))	$(A \mid B) \mid C = A \mid (B \mid C)$
	(BitAnd (BitAnd ?a ?b) ?c) → (BitAnd ?a (BitAnd ?b ?c))	$(A \& B) \& C = A \& (B \& C)$
	(BitAnd ?a (BitOr ?b ?c)) → (BitOr (BitAnd ?a ?b) (BitAnd ?a ?c))	$A \& (B \mid C) = (A \& B) \mid (A \& C)$
	(BitOr ?a (BitAnd ?b ?c)) → (BitAnd (BitOr ?a ?b) (BitOr ?a ?c))	$A \mid (B \& C) = (A \mid B) \& (A \mid C)$
Shift Operations	(Mul (Power (Atom (Int 2)) ?b) ?a) → (ShiftLeft ?a ?b)	$2^B * A = A \ll B$
	(ShiftLeft ?a (Add ?b ?c)) → (ShiftLeft (ShiftLeft ?a ?b) ?c)	$A \ll (B + C) = (A \ll B) \ll C$
	(Mul ?a 2) → (shiftLeft ?a (Atom (Int 1)))	$A * 2 = A \ll 1$
	(ShiftLeft (Mul ?a ?b) ?c) → (Mul (ShiftLeft ?a ?c) ?b)	$(A \times B) \ll C = (A \ll C) \times B$
	(AshiftLeft (Mul ?a ?b) ?c) → (Mul (AshiftLeft ?a ?c) ?b)	$(A \times B) \lll C = (A \lll C) \times B$
	(Mul (Power (Atom (Int 2)) ?b) ?a) → (AshiftLeft ?a ?b)	$A \times 2^B = A \lll B$
	(AshiftLeft ?a (Add ?b ?c)) → (AshiftLeft (AshiftLeft ?a ?b) ?c)	$A \lll (B + C) = (A \lll B) \lll C$
	(Mul ?a (Atom (Int 2))) → (AshiftLeft ?a (Atom (Int 1)))	$A \times 2 = A \lll 1$
	(Mul (ShiftLeft ?a ?c) ?b) → (ShiftLeft (Mul ?a ?b) ?c)	$(A \ll C) \times B = (A \times B) \ll C$
	(Mul (AshiftLeft ?a ?c) ?b) → (AshiftLeft (Mul ?a ?b) ?c)	$(A \lll C) \times B = (A \times B) \lll C$
	(Div ?a (Power 2 ?b)) → (ShiftRight ?a ?b)	$A \div 2^B = A \gg B$
	(ShiftRight (ShiftRight ?a ?b) ?c) → (ShiftRight ?a (Add ?b ?c))	$(A \gg B) \gg C = A \gg (B + C)$
	(Mul ?a (Power 2 ?b)) → (ShiftLeft ?a ?b)	$A \times 2^B = A \ll B$
	(AshiftRight ?a (Add ?b ?c)) → (AshiftRight (AshiftRight ?a ?b) ?c)	$(A \ggg B) \ggg C = A \ggg (B + C)$
	(Div ?a (Atom (Int 2))) → (ShiftRight ?a (Atom (Int 1)))	$A \div 2 = A \gg 1$
	(BitAnd (ShiftRight ?a ?b) (Atom (Int 1))) → (BitIndex ?a ?b)	$(A \gg B) \& 1 = A[B]$
	(Mul (ShiftRight ?a ?c) ?b) → (ShiftRight (Mul ?a ?b) ?c)	$(A \gg C) \times B = (A \times B) \gg C$
	(Mul (AshiftRight ?a ?c) ?b) → (AshiftRight (Mul ?a ?b) ?c)	$(A \ggg C) \times B = (A \times B) \ggg C$

D TRAINING OBJECTIVES

Training proceeds in stages. The imitation policy is first optimized using rule and location prediction losses. The value heads are then refined with trajectory returns, successor-value targets, and within-

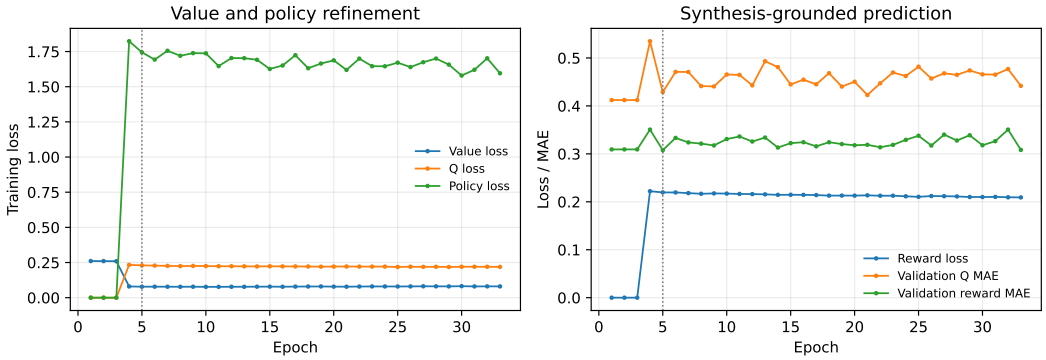


Figure 4: Training dynamics of TREAD. The left panel shows the value, action-value, and policy losses during value warm-up and trajectory-guided refinement. The right panel shows the synthesis-grounded reward-prediction loss, together with the validation Q MAE and reward MAE. The vertical marker indicates the end of the five-epoch value warm-up stage. These curves provide training diagnostics for the value and reward models and are separate from the post-synthesis QoR evaluation.

state action-ranking terms. The synthesis-grounded reward predictor is trained separately to regress transition-level synthesis improvement targets. For a valid transition, we use a robust regression loss

$$\mathcal{L}_{\text{reward}} = \text{SmoothL1}\big(\widehat{r}_\omega(s,a), r(s,a)\big). \tag{20}$$

For an independently formed within-state preference pair, the corresponding ranking head uses the logistic pairwise loss

$$\mathcal{L}_{\text{rank}} = \log\left(1 + \exp\left[-\big(D(s,a^+) - D(s,a^-)\big)\right]\right), \tag{21}$$

where a^+ has lower measured ADP than a^-. The ranking head and the transition-reward predictor therefore supervise different quantities. Trajectory samples provide successor-based targets for long-horizon value learning, whereas same-state DC candidates supervise immediate synthesis improvement and within-state candidate ordering. In particular, DC-only candidates are used as counterfactual supervision at the current state rather than as complete Bellman transitions.

E Training Dynamics and Convergence

Figure 4 summarizes the optimization dynamics recorded during value warm-up and trajectory-guided refinement. The first five epochs warm up the value estimates, after which the value and policy objectives are refined jointly using the complete trajectory targets. The curves are plotted on a common epoch axis; the vertical marker denotes the end of value warm-up.

F Dataset and Training Configuration

Unless stated otherwise, training uses batch size 8, random seed 42, discount factor $\gamma = 1.0$, and Adam with learning rate 10^{-4}. The actor learning rate is scaled by 0.1 during value refinement. The value module is warmed up for five epochs and then refined for 20 epochs using trajectory returns, successor-value targets, and policy supervision. The synthesis-grounded reward predictor is trained independently with AdamW from within-state candidate comparisons. The long-horizon critic is initialized from the imitation checkpoint, and its two action-value heads are retained for residual value estimation.

G Inference Search Configuration

The deployment configuration is summarized in Table 5. IL first generates a baseline endpoint. Residual search then expands legal actions from this endpoint using a bounded beam. Candidate states are scored by the learned residual estimator, and the best residual path replaces the IL endpoint only when its score exceeds the fixed acceptance margin.

Table 5: Default inference configuration.

Parameter	Value
IL maximum depth	8
Residual maximum depth	7
Total maximum depth	15
Beam width	5
Top rules per state	4
Top legal locations per rule	4
Residual acceptance margin	0.05

Table 6: Definitions of the component variants used in the ablation study.

ID	Configuration	Enabled components
A0	IL	Imitation policy only
A1	IL + Residual	Imitation policy and residual search
A2	IL + RL	IL endpoint, residual search, and RL value guidance
A3	IL + DC	IL endpoint, residual search, and same-state DC supervision
A4	TREAD	IL, residual search, RL guidance, and DC supervision

H Synthesis and Evaluation Protocol

All original and rewritten RTL variants are evaluated with the same Design Compiler flow, technology library, timing constraints, and report parser. Sequential designs use a 20 ns clock constraint, while designs without a clock use a 20 ns input-to-output maximum-delay constraint. We compute

$$\mathrm{ADP} = \mathrm{Area} \times \mathrm{Delay}\,.$$

A result is considered valid only when the RTL parses, elaborates, links, and synthesizes successfully and produces finite, positive area and delay.

For synthesis-grounded supervision, each candidate record stores its source state, rewrite rule, target node, predicted score, and measured area, delay, and ADP. Candidates generated from the same source state are compared using their measured ADP, and only complete synthesis records are used to construct the corresponding preference data.

I Additional Ablation and Analysis

Table 6 defines the controlled component variants used in the ablation study. All variants use the same input designs, rewrite rules, synthesis flow, validity checks, and aggregation procedure. Residual search always starts from the IL endpoint; the variants progressively enable residual exploration, long-horizon RL guidance, and same-state DC supervision.

I.1 Metric Computation

For each valid design d, the normalized ADP for method m is

$$\mathrm{NormADP}(d, m) = \frac{\mathrm{ADP}(d, m)}{\mathrm{ADP}(d, \mathrm{Original})}.$$

For each category, we report the geometric mean of this ratio over the valid designs:

$$\mathrm{NormADP}(m) = \left(\prod_{d \in \mathcal{V}_m} \mathrm{NormADP}(d, m) \right)^{1/|\mathcal{V}_m|},$$

where $\mathcal{V}_m$ is the set of valid designs for method m. The reported ADP reduction is

$$\Delta\,\mathrm{ADP} = (1 - \mathrm{NormADP}(m)) \times 100\%.$$

Consequently, lower normalized ADP and larger positive ADP reduction indicate better QoR. Missing candidates, synthesis failures, and non-positive ADP values are excluded before aggregation.