POET: Power-Oriented Evolutionary Tuning for LLM-Based RTL PPA Optimization

Heng Ping
hping@usc.edu
University of Southern California
United States

Peiyu Zhang
pzhang65@usc.edu
University of Southern California
United States

Zhenkun Wang
zhenkunw@usc.edu
University of Southern California
United States

Shixuan Li
sli97750@usc.edu
University of Southern California
United States

Anzhe Cheng
anzheche@usc.edu
University of Southern California
United States

Wei Yang
wyang930@usc.edu
University of Southern California
United States

Paul Bogdan*
pbogdan@usc.edu
University of Southern California
United States

Shahin Nazarian*
shahin.nazarian@usc.edu
University of Southern California
United States

Abstract

Applying large language models (LLMs) to RTL code optimization for improved power, performance, and area (PPA) faces two key challenges: ensuring functional correctness of optimized designs despite LLM hallucination, and systematically prioritizing power reduction within the multi-objective PPA trade-off space. We propose POET (Power-Oriented Evolutionary Tuning), a framework that addresses both challenges. For functional correctness, POET introduces a differential-testing-based testbench generation pipeline that treats the original design as a functional oracle, using deterministic simulation to produce golden references and eliminating LLM hallucination from the verification process. For PPA optimization, POET employs an LLM-driven evolutionary mechanism with non-dominated sorting, power-first intra-level ranking, and proportional survivor selection to steer the search toward the low-power region of the Pareto front without manual weight tuning. Evaluated on the RTL-OPT benchmark across 40 diverse RTL designs, POET achieves 100% functional correctness, the best power on all 40 designs, and competitive area and delay improvements.

CCS Concepts

• **Hardware** → **High-level and register-transfer level synthesis**; • **Computing methodologies** → *Natural language processing*.

Keywords

RTL optimization, large language models, evolutionary algorithm

The rapid advancement of large language models (LLMs) [10, 28, 31] is reshaping electronic design automation (EDA), enabling new capabilities across design space exploration, verification, and optimization [16, 26, 34]. In particular, the strong code generation abilities of LLMs have inspired growing interest in RTL design [27], and benchmarks such as VerilogEval [11, 17] and RTLLM [12, 13] have further accelerated this area. However, existing efforts [18, 19] predominantly target *spec-to-RTL generation*, i.e., producing functionally correct Verilog from specification, while paying limited

*Corresponding author.

attention to PPA quality. In practice, initial RTL implementations are seldom PPA-optimal regardless of whether they are written by humans or generated by LLMs [9]. *RTL-to-RTL optimization*, which transforms a functionally correct design into a PPA-superior equivalent, therefore constitutes an important yet underexplored direction [6]. Meanwhile, the proliferation of IoT devices, mobile computing, and embedded systems has made low-power optimization an increasingly essential objective in modern chip design [2].

Applying LLMs to power-centric RTL PPA optimization faces two fundamental challenges. The first concerns the *functional correctness of optimized designs*. Due to inherent hallucination tendencies, LLMs frequently produce functional errors when generating PPA-optimized variants [1], with experiments on RTL-OPT [14] showing that even DeepSeek-R1 introduces errors in over 25% of attempts. Since functional incorrectness renders any PPA improvement meaningless, robust verification is essential. Existing approaches employ diverse verification strategies: VeriOpt [21] and LLM-VeriPPA [22] rely on manually crafted testbenches, which are labor-intensive and difficult to scale; SymRTLO [23] directs the LLM to generate testbenches from the input RTL, yet the results often suffer from low reliability; although ASPEN [35] employs equivalence checking, its simplified formulation with sparse constraints cannot fully guarantee functional equivalence. These limitations call for a more reliable and automated verification method.

The second challenge lies in the *directionality and effectiveness of power-prioritized PPA optimization*. PPA metrics exhibit fundamental trade-offs, forming a Pareto front where improving one objective necessarily degrades another [8]. Existing methods handle this multi-objective nature inadequately. In-context learning approaches such as VeriOpt [21] and LLM-VeriPPA [22] embed optimization techniques into prompts but lack systematic design space exploration. Evolutionary approaches like REvolution [15] and VFlow [25] collapse the multi-objective problem into a single weighted fitness score (e.g., $F = \alpha \cdot \Delta P + \beta \cdot \Delta A + \gamma \cdot \Delta T$), which requires manual weight tuning and cannot reach non-convex Pareto regions. None of these approaches can systematically prioritize power while maintaining Pareto optimality.

To address both challenges, we propose **POET** (Power-Oriented Evolutionary Tuning), a framework that integrates reliable functional verification with power-oriented multi-objective RTL PPA optimization. For functional correctness, POET employs a *differential-testing-based testbench generation pipeline.* The key insight is that the input design, while PPA-suboptimal, is functionally correct and can serve as a functional oracle [14]. POET directs the LLM to extract a functional specification and generate diverse test stimuli, which are then simulated through the original design to produce golden output signals. The resulting input-output pairs are assembled into a checking testbench, ensuring expected outputs come from deterministic simulation rather than LLM reasoning. For PPA optimization, POET adopts an evolutionary algorithm [20] that iteratively generates, evaluates, and selects RTL design variants through LLM-driven mutation and crossover. Designs are ranked via power-oriented non-dominated sorting, which first partitions the population into Pareto levels, then sorts each level by power in ascending order so that lower-power designs are always preferred among Pareto-equivalent candidates. A proportional survivor selection strategy then allocates more slots to higher-priority levels while preserving diversity from lower levels, steering the search toward the low-power region of the Pareto front without manually tuned weight parameters.

Our key contributions:

- **Differential-Testing-Based Testbench Generation**: We propose an automated pipeline that treats the functionally correct input design as an oracle. Golden input-output pairs are obtained by simulating LLM-generated test stimuli through the original design, from which reliable checking testbenches are assembled, avoiding manual testbench creation or LLM-generated testbenches.
- **Power-Oriented Multi-Objective Evolutionary Optimization**: We introduce an LLM-driven evolutionary mechanism employing non-dominated sorting, power-first intra-level ranking, and proportional survivor selection to systematically prioritize power reduction while preserving Pareto optimality, without manual weight tuning.
- **Superior Performance**: Evaluation on the RTL-OPT benchmark across 40 diverse RTL designs demonstrates that POET achieves state-of-the-art power reduction with competitive area and delay improvements. Ablation studies further confirm the effectiveness of each component.

1 Background

1.1 LLM-Based RTL Optimization

With the rapid progress of LLMs in complex tasks [3, 4, 7, 29, 30, 33, 36], a key prerequisite for RTL optimization is high-quality benchmarks. Unlike RTL code generation, where benchmarks are well established, dedicated datasets for RTL PPA optimization have only recently emerged. PfP [8] constructs a low-power RTL benchmark by pairing baseline modules with manually crafted low-power variants, though limited to small modules. RTL-OPT [14] provides a larger-scale benchmark of 40 expert-crafted designs covering diverse categories. However, the limited scale of optimization data constrains training-based approaches such as PPA-RTL [37].

Given these limitations, most recent works adopt inference-time agentic frameworks combining LLMs with external tool feedback. In-context learning approaches such as VeriOpt [21] and LLM-VeriPPA [22] incorporate PPA feedback into iterative prompting but lack systematic design space exploration. Structured rewriting approaches take a different path: RTLRewriter [32] partitions RTL designs and employs cost-aware MCTS for rewriting; ASPEN [35] combines LLM-guided rewrite rules with e-graph-based equality saturation; SymRTLO [23] integrates AST templates for datapath optimization with symbolic scripts for FSM minimization. These methods focus on localized transformations without a global multi-objective search strategy. A third line of work, including REvolution [15], EvolVE [9], and VFlow [25], applies evolutionary algorithms to search more broadly by evolving populations of RTL candidates. However, as detailed in Section 1.2, these approaches uniformly rely on weighted-sum fitness or proxy metrics, which cannot express power-centric optimization preferences without manual hyperparameter tuning.

1.2 Evolutionary Approaches for RTL Design

Recent work has demonstrated the effectiveness of combining evolutionary computation (EC) with LLMs for RTL design [9, 15, 25]. In this paradigm, a population of design candidates is maintained and iteratively refined through an evolutionary loop of offspring generation, evaluation, and survivor selection. Each individual represents a design candidate containing the RTL implementation and its evaluation results. New offspring are produced by prompting the LLM with selected parents and an evolutionary operator such as mutation (improving a single parent) or crossover (combining two parents). Each offspring is evaluated for functional correctness and PPA quality, and the fittest ones survive into the next generation.

A critical design choice is how to define fitness for PPA optimization. REvolution [15] computes a weighted sum $F = \alpha \cdot \Delta P + \beta \cdot \Delta A + \gamma \cdot \Delta T$ with manually set weights. EvolVE [9] uses the area-delay product as a proxy metric. VFlow [25] decomposes the search into specialized populations (functionality-first, area-first, timing-first, and balanced), but each still optimizes a single scalar objective. Despite their broader design space exploration, these approaches all rely on weighted-sum scalarization or proxy metrics, preventing them from expressing a systematic preference for power reduction. In contrast, POET adopts multi-objective non-dominated sorting to preserve true Pareto optimality while injecting power preference through the selection mechanism, as detailed in Section 2.

2 Method

Figure 1 illustrates the overall architecture of POET, which comprises two core components: a differential-testing-based testbench generation pipeline that treats the original functionally correct design as a golden oracle to produce a reliable testbench, and a power-oriented evolutionary optimization engine that iteratively generates, evaluates, and selects RTL design variants.

2.1 Problem Formulation

Given a functionally correct RTL design $\mathcal{D}_{\text{orig}}$ with PPA metrics $\mathbf{m}_{\text{orig}} = (P_{\text{orig}}, A_{\text{orig}}, D_{\text{orig}})$ for power, area, and critical path delay, the goal is to find a functionally equivalent design $\mathcal{D}^*$ with superior
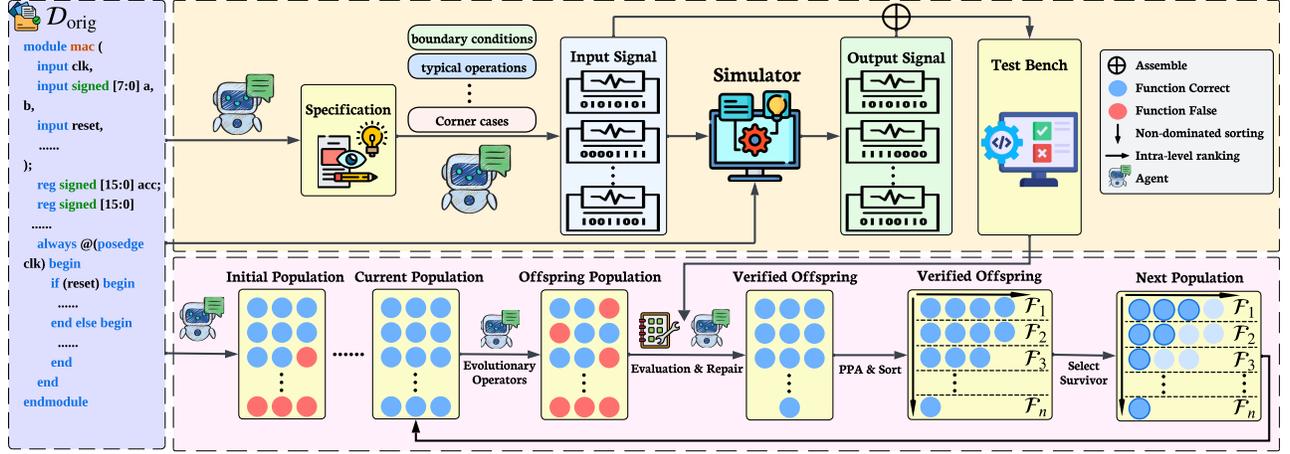
Figure 1: POET. Top: differential-testing-based testbench generation. Bottom: power-oriented evolutionary optimization.

PPA quality:

$$\mathcal{D}^* = \arg\min_{\mathcal{D}} \ \mathbf{m}(\mathcal{D}) = (P, A, D) \quad \text{s.t.} \quad \mathcal{D} \equiv_f \mathcal{D}_{\text{orig}} \qquad (1)$$

Since PPA objectives are generally conflicting [8], we adopt Pareto dominance to compare designs:

$$\mathcal{D}_a \succ \mathcal{D}_b \iff \mathbf{m}(\mathcal{D}_a) \leq \mathbf{m}(\mathcal{D}_b) \ \wedge \ \mathbf{m}(\mathcal{D}_a) \neq \mathbf{m}(\mathcal{D}_b) \quad (2)$$

A design is Pareto optimal if no other feasible design dominates it. To systematically prefer power reduction without collapsing into a single scalar, POET ranks designs on the same Pareto level (i.e., neither dominates the other) by power in ascending order:

$$\mathcal{D}_a \succ_P \mathcal{D}_b \iff P_a < P_b \qquad (3)$$

This steers the search toward the low-power region of the Pareto front while preserving multi-objective trade-off information, unlike weighted-sum formulations [9, 15] that require manual tuning.

2.2 Differential-Testing-Based Testbench Generation

Enforcing the functional equivalence constraint $\mathcal{D} \equiv_f \mathcal{D}_{\text{orig}}$ (Eq. 1) throughout the evolutionary process requires a reliable testbench for every input design. Existing approaches either rely on simplified equivalence checking [35], which cannot fully guarantee functional equivalence, or assign testbench generation entirely to an LLM [23], where hallucination-induced errors undermine verification reliability. POET addresses this through a differential testing pipeline that leverages LLMs for tasks where they excel, such as specification comprehension and test scenario design, while delegating output computation to deterministic simulation.

The key insight is to decompose testbench generation into input-side tasks, well-suited for LLMs, and output-side tasks, which require cycle-accurate reasoning that LLMs handle poorly for sequential circuits. By simulating $\mathcal{D}_{\text{orig}}$ with LLM-generated input stimuli, we obtain golden output signals without relying on LLM reasoning. The pipeline comprises four steps:

Step 1: Specification Extraction. The LLM analyzes $\mathcal{D}_{\text{orig}}$ and produces a structured specification $\mathcal{S}$ following a predefined template,

including module interface, sequential/combinational classification, reset behavior, and functional description:

$$\mathcal{S} = \text{LLM}_{\text{spec}}(\mathcal{D}_{\text{orig}}) \qquad (4)$$

Extracting $\mathcal{S}$ as an intermediate representation, rather than generating test vectors directly from raw RTL, provides a concise functional summary that facilitates more accurate test scenario identification.
Step 2: Test Vector Generation. Based on $\mathcal{S}$, the LLM identifies test scenarios covering boundary conditions, typical operations, and corner cases, and generates the corresponding input signal sequences:

$$\mathbf{V} = \{v_1, v_2, \ldots, v_n\} = \text{LLM}_{\text{vec}}(\mathcal{S}) \qquad (5)$$

where each v_i is a time-indexed input vector for one test scenario.
Step 3: Golden Output Capture. Rather than asking the LLM to reason about expected outputs, we construct a stimulus testbench from a predefined template that applies $\mathbf{V}$ to $\mathcal{D}_{\text{orig}}$ and records all output signals via an RTL simulator:

$$\mathbf{O} = \{o_1, o_2, \ldots, o_n\} = \text{Sim}(\mathcal{D}_{\text{orig}}, \mathbf{V}) \qquad (6)$$

Since $\mathcal{D}_{\text{orig}}$ is functionally correct, $\mathbf{O}$ constitutes a reliable golden reference. This avoids potential hallucination that would arise from relying on LLM reasoning for output computation.
Step 4: Checking Testbench Assembly and Validation. The input-output pairs $(\mathbf{V}, \mathbf{O})$ are assembled into a checking testbench $\mathcal{T}$ through a predefined template:

$$\mathcal{T} = \text{Assemble}(\mathbf{V}, \mathbf{O}) \qquad (7)$$

A validation step verifies that $\mathcal{D}_{\text{orig}}$ passes all assertions in $\mathcal{T}$; the testbench is accepted only upon successful validation. The validated $\mathcal{T}$ then serves as the functional verification oracle for all candidate designs, enforcing:

$$\mathcal{D} \equiv_f \mathcal{D}_{\text{orig}} \Longleftarrow \mathcal{T}(\mathcal{D}) = \text{PASS} \qquad (8)$$

where $\mathcal{T}(\mathcal{D})$ denotes executing testbench on candidate design.

Algorithm 1 Power-Oriented Evolutionary Optimization

Require: $\mathcal{D}_{\text{orig}}, \mathcal{T}, N, \lambda, G, R$
Ensure: Pareto-optimal design set
1: $\mathcal{P}_0 \leftarrow \textsc{InitPopulation}(\mathcal{D}_{\text{orig}}, N)$
2: **for** $t = 1$ **to** G **do**
3:    $O \leftarrow \emptyset$
4:    **for** $j = 1$ **to** λ **do**
5:       Select operator a_j via UCB (Eq. 9)
6:       Select parent(s) from $\mathcal{P}_{t-1}$ with $p_i \propto 1/r_i$
7:       $\mathcal{D}' \leftarrow \text{LLM}(\text{parent(s)}, a_j, \Delta\mathbf{m})$
8:       **for** $r = 1$ **to** R **do**
9:          **if** $\mathcal{T}(\mathcal{D}') = \text{PASS}$ **then**
10:            **break**
11:          **end if**
12:          $\mathcal{D}' \leftarrow \text{LLM}_{\text{repair}}(\mathcal{D}', \text{error\_log})$
13:       **end for**
14:       **if** $\mathcal{T}(\mathcal{D}') = \text{PASS}$ **then**
15:          $\mathbf{m}' \leftarrow \text{Synthesize}(\mathcal{D}'); O \leftarrow O \cup \{(\mathcal{D}', \mathbf{m}')\}$
16:          Update UCB reward for a_j
17:       **end if**
18:    **end for**
19:    $\mathcal{P}_t \leftarrow \textsc{PowerOrientedSelect}(\mathcal{P}_{t-1} \cup O, N)$
20: **end for**
21: **return** Pareto front of $\mathcal{P}_G$

2.3 Power-Oriented Evolutionary Optimization

RTL-to-RTL optimization requires selecting and combining design-specific rewriting strategies across a large transformation space that demands broad exploration, while synthesis feedback from each attempt should be exploited to guide subsequent efforts. Evolutionary algorithms are naturally suited for this task, as they maintain diverse candidates for exploration while using fitness-based selection for exploitation [15].

POET formulates the evolutionary process over a population of RTL design candidates. Each individual is defined as $\mathcal{I} = (\mathcal{D}, \mathbf{m})$, where $\mathcal{D}$ is the RTL implementation and $\mathbf{m} = (P, A, D)$ is its PPA metric. The population at generation t is denoted $\mathcal{P}_t = \{\mathcal{I}_1, \ldots, \mathcal{I}_N\}$ with size N. The initial population $\mathcal{P}_0$ is seeded using diverse strategies (power-focused, area-focused, timing-focused, balanced, architectural exploration, and simplification) to ensure broad initial coverage. The evolutionary loop iterates for G generations, each comprising offspring generation, evaluation, and survivor selection. Algorithm 1 summarizes the complete procedure.

2.3.1 Offspring Generation. Each generation produces λ offspring by applying evolutionary operators to parents sampled from $\mathcal{P}_{t-1}$ with probability $p_i \propto 1/r_i$, where r_i is the global rank of individual $\mathcal{I}_i$ after power-oriented sorting (Section 2.3.3). POET defines six evolutionary operators: five *mutation* operators that each transform a single parent (**Improve**, **Refactor**, **Explore**, **Simplify**, and **Fusion**) and one *crossover* operator (**Crossover**) that combines two parents. All operator prompts include the parent's PPA change relative to $\mathcal{D}_{\text{orig}}$ as a percentage delta $\Delta\mathbf{m}$ to provide quantitative optimization context. Each operator embeds domain-specific circuit optimization techniques, summarized as follows:

Improve targets the parent's weakest PPA metric from synthesis feedback. *Power*: e.g., clock gating, operand isolation, register update suppression; *Area*: e.g., resource sharing, bit-width reduction, precomputation with LUT replacement; *Delay*: e.g., operator strength reduction, carry optimization, critical path restructuring.
Refactor keeps the parent's optimization strategy but re-implements it structurally. *Power*: e.g., FSM re-encoding to reduce switching activity; *Area*: e.g., flat logic to parameterized blocks; *Delay & Area*: e.g., behavioral to structural arithmetic such as Wallace trees.
Explore generates a maximally different architectural approach. *Power & Area*: e.g., different encoding schemes or pipeline configurations; *Delay*: e.g., alternative arithmetic structures; *All*: fundamentally different algorithmic alternatives.
Simplify reduces implementation complexity. *Power*: e.g., eliminating redundant computations; *Area*: e.g., merging redundant logic and reducing bit widths; *Delay*: e.g., simplifying boolean expressions to reduce logic depth.
Fusion augments a single parent with complementary techniques. *Power + Area*: e.g., clock gating with resource sharing; *Power + Delay*: e.g., operand isolation with strength reduction; *Area + Power*: e.g., bit-width reduction with logic consolidation. Conflicting combinations are explicitly discouraged.
Crossover combines two parents by inheriting each parent's PPA-superior techniques. *Power*: inherited from the parent with better power; *Area*: from better area; *Delay*: from better delay. Potential conflicts are flagged.

To balance exploration and exploitation across operators, POET employs UCB-based adaptive strategy selection [20]:

$$\text{UCB}(a_i) = \underbrace{\frac{R_i}{n_i}}_{\text{exploitation}} + c \underbrace{\sqrt{\frac{\ln T}{n_i}}}_{\text{exploration}} \tag{9}$$

where R_i and n_i are the cumulative reward and selection count of operator a_i, T is the total selection count, and c is an exploration coefficient. The exploitation term favors operators with higher average reward, while the exploration term grows for less frequently selected operators. A reward of +1 is granted when the offspring achieves lower power than $\mathcal{D}_{\text{orig}}$.

2.3.2 Evaluation and Repair. Each offspring $\mathcal{D}'$ is verified against the checking testbench $\mathcal{T}$. If $\mathcal{T}(\mathcal{D}') \neq \text{PASS}$, the error log is fed back to the LLM for repair, repeating for up to R attempts; offspring that still fail are discarded. This maintains an *all-correct population invariant*: every individual in $\mathcal{P}_t$ satisfies $\mathcal{T}(\mathcal{I}) = \text{PASS}$. Verified offspring are then evaluated for PPA through logic synthesis.

2.3.3 Power-Oriented Survivor Selection. After evaluation, the combined pool $\mathcal{P}_{t-1} \cup O$ is reduced to N survivors through three steps built upon NSGA-II [5]:
(1) Non-dominated sorting. The pool is partitioned into Pareto levels $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_L$ via the dominance relation $\succ$ (Eq. 2), where $\mathcal{F}_1$ is the Pareto front and subsequent levels represent lower quality.
(2) Intra-level ranking. Within each level $\mathcal{F}_k$, individuals are sorted by power in ascending order via $\succ_P$ (Eq. 3), ensuring lower-power designs rank higher among Pareto-equivalent candidates.
(3) Proportional slot allocation. Unlike standard NSGA-II sequential fill, which may completely eliminate lower Pareto levels,

Table 1: PPA comparison on RTL-OPT. Area in μm^2, CPD in ns, power in μW. Green = best result; ✗ = functionally incorrect.

Design	Original			I/O			CoT			REvolution			POET		
	Area	CPD	Power	Area	CPD	Power	Area	CPD	Power	Area	CPD	Power	Area	CPD	Power
add_sub	201.89	0.58	161.0	201.89	0.58	161.0	201.89	0.58	161.0	135.93	0.64	114.0	153.75	0.55	99.20
adder	458.05	1.15	393.0	✗	✗	✗	✗	✗	✗	409.37	1.26	363.0	272.65	1.03	195.0
adder_carry	52.14	0.50	32.50	48.94	0.33	29.10	48.94	0.34	29.10	47.61	0.32	28.30	47.61	0.32	28.30
adder_select	432.25	1.14	305.0	383.04	1.13	253.0	432.25	1.14	305.0	318.40	1.08	249.0	191.79	1.13	120.0
addr_calcu	124.75	0.59	196.0	124.75	0.59	196.0	124.75	0.59	196.0	124.75	0.59	196.0	118.37	0.46	138.0
alu_8bit	169.44	0.44	111.0	167.85	0.47	109.0	169.44	0.44	111.0	163.06	0.48	101.0	164.65	0.39	96.40
alu_64bit	1483.22	3.47	1040	1392.51	2.43	983.0	1392.51	2.43	983.0	1291.96	2.77	976.0	1321.75	2.55	944.0
calculation	902.80	2.46	1850	✗	✗	✗	763.69	2.47	1680	697.72	2.48	1470	657.82	2.41	1140
comparator	42.03	0.30	19.10	42.03	0.30	19.10	42.03	0.30	19.10	42.03	0.30	19.10	36.18	0.37	16.50
comparator_2bit	10.11	0.16	4.73	9.31	0.08	3.73	7.98	0.09	3.47	7.98	0.08	3.47	7.98	0.08	3.47
comparator_4bit	21.55	0.16	9.99	17.02	0.12	8.19	17.82	0.14	8.32	12.77	0.16	4.87	12.77	0.16	4.87
comparator_8bit	47.88	0.24	20.90	30.32	0.22	14.60	34.85	0.20	16.10	30.32	0.22	14.60	30.32	0.22	14.60
comparator_16bit	104.80	0.29	44.50	65.17	0.29	32.80	67.83	0.28	31.20	36.71	0.22	14.30	36.71	0.22	14.30
decoder_6bit	99.22	0.21	19.50	99.22	0.21	19.50	99.22	0.21	19.50	78.20	0.11	19.50	78.20	0.11	19.50
decoder_8bit	410.70	0.48	57.30	423.21	0.63	58.10	422.41	1.14	102.0	250.04	0.18	46.60	250.04	0.18	46.60
divider_4bit	40.96	0.34	24.60	40.96	0.34	24.60	40.96	0.34	24.60	35.91	0.31	21.20	24.21	0.35	11.10
divider_8bit	193.65	1.42	508.0	✗	✗	✗	✗	✗	✗	178.75	1.49	407.0	178.75	1.49	407.0
divider_16bit	1142.47	6.35	102000	1142.47	6.35	102000	752.51	5.44	26700	754.11	5.61	24700	748.52	5.42	21400
divider_32bit	3196.79	18.28	275000	✗	✗	✗	3196.79	18.28	275000	3028.14	21.25	241000	3028.14	21.25	241000
fsm	111.45	0.24	123.0	✗	✗	✗	160.66	0.48	50.70	77.67	0.22	71.60	113.32	0.33	26.80
fsm_encode	332.23	0.62	1150	✗	✗	✗	✗	✗	✗	✗	✗	✗	381.18	0.65	226.0
gray	99.22	0.32	384.0	119.43	0.23	167.0	155.08	0.38	242.0	161.20	0.37	117.0	261.21	0.86	102.0
mac	1020.11	0.99	2800	✗	✗	✗	✗	✗	✗	975.69	0.92	1940	975.69	0.92	1940
mul	496.89	0.87	1280	✗	✗	✗	✗	✗	✗	486.25	0.80	1080	491.04	0.78	1060
mul_const	69.69	0.26	47.00	52.40	0.21	30.30	52.40	0.21	30.30	52.40	0.21	30.30	52.40	0.21	30.30
mul_subexpression	345.27	1.23	1400	506.46	1.53	819.0	364.15	0.83	596.0	339.68	0.78	489.0	335.69	0.72	443.0
multi_if	10.91	0.15	4.21	7.98	0.10	3.18	10.91	0.15	4.21	✗	✗	✗	7.98	0.10	3.18
mux_4to1_16bit	83.79	0.19	29.60	83.79	0.19	29.60	83.79	0.19	29.60	89.38	0.11	29.30	89.38	0.11	29.30
mux_4to1_64bit	326.38	0.60	138.0	326.38	0.60	138.0	326.38	0.59	138.0	357.50	0.11	117.0	357.50	0.11	117.0
mux_dead	21.28	0.04	7.06	21.28	0.04	7.06	21.28	0.04	7.06	21.28	0.04	7.06	21.28	0.04	7.06
mux_encode	63.57	0.21	23.80	63.57	0.21	23.80	63.57	0.21	23.80	63.57	0.20	23.80	63.57	0.20	23.50
mux_large	124.49	0.34	53.00	90.17	0.30	29.90	90.17	0.30	29.90	92.57	0.30	29.30	88.58	0.30	28.60
register	9712.19	0.88	14300	✗	✗	✗	✗	✗	✗	✗	✗	✗	10143.38	1.07	8290
saturating_add	78.47	0.26	83.60	✗	✗	✗	78.47	0.26	83.60	42.29	0.26	25.00	42.29	0.26	25.00
selector	42.29	0.12	37.80	37.24	0.11	36.60	37.24	0.11	36.60	10.11	0.11	4.30	10.11	0.11	4.30
sub_4bit	22.34	0.17	11.50	22.34	0.17	11.40	✗	✗	✗	20.75	0.17	11.30	20.48	0.23	10.70
sub_8bit	48.15	0.34	28.80	48.15	0.34	29.40	48.15	0.34	28.80	48.15	0.34	28.80	46.28	0.33	27.40
sub_16bit	102.14	0.62	63.60	98.95	0.57	63.50	98.95	0.57	63.40	97.89	0.64	61.80	96.56	0.57	58.10
sub_32bit	208.54	1.31	137.0	✗	✗	✗	202.16	1.22	130.0	✗	✗	✗	196.57	1.10	121.0
ticket_machine	59.32	0.39	75.70	57.72	0.35	62.90	57.72	0.35	62.90	33.25	0.26	37.90	40.43	0.20	29.10

POET assigns each level a quota proportional to its priority:

$$s_k = \max(1, \lfloor N \cdot w_k \rfloor), \quad w_k = \frac{L - k + 1}{\sum_{j=1}^{L}(L - j + 1)} \qquad (10)$$

Higher-priority levels (lower k) receive larger quotas, while every level retains at least one representative to preserve potentially valuable optimization strategies. Individuals are selected first by Pareto level priority, then by power priority within each level.

3 Experiments

We conduct comprehensive experiments to answer three research questions: **RQ1**: How does POET compare with existing LLM-based RTL optimization methods in functional correctness and PPA quality? **RQ2**: What is the contribution of each core component? **RQ3**: How does the evolutionary process optimize PPA over generations?

3.1 Experimental Setup

Benchmark and Baselines. We evaluate on the RTL-OPT benchmark [14], which contains 40 expert-crafted RTL designs spanning arithmetic circuits, control logic, and datapath modules. Since RTL-OPT provides two versions per design, we select the one with higher

power as the optimization target to align with our power-centric objective. We compare POET against three baselines: (1) **I/O prompting**, which directly instructs the LLM to optimize the RTL code; (2) **Chain-of-Thought (CoT)** [24], which guides the LLM through step-by-step optimization reasoning; and (3) **REvolution** [15], a state-of-the-art evolutionary framework using weighted-sum fitness. All methods use GPT-4o-mini as the backbone LLM with the same total number of LLM generation calls for fair comparison.

Evaluation. PPA metrics (area in μm^2, critical path delay in ns, power in μW) are obtained using Yosys for logic synthesis and OpenSTA for timing and power analysis with the NanGate 45nm standard cell library. Functional correctness is verified by both simulation via Icarus Verilog against POET's generated testbenches (manually inspected for reliability) and Yosys equivalence checking. Since RTL-OPT does not provide pre-built testbenches, REvolution uses LLM-generated testbenches for its internal verification.

Hyperparameters. POET uses population size N=10, offspring count λ=10, maximum generations G=10, repair attempts R=3.

3.2 Main Results (RQ1)

Table 1 presents the per-design PPA comparison. POET achieves the best power on all 40 designs (100%), the best area on 29 (72.5%),
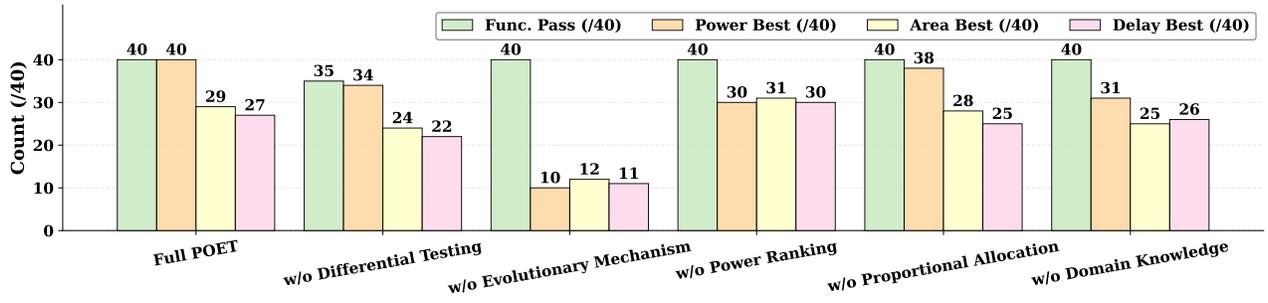
Figure 2: Ablation study on RTL-OPT. Each bar indicates the number of designs achieving the best result among all baselines.
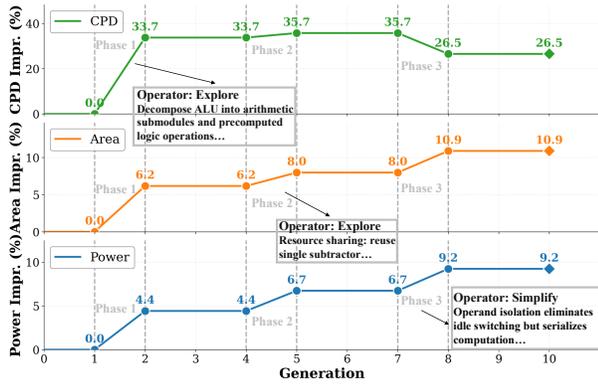

Figure 3: Evolutionary trajectory of POET on `alu_64bit`.

and the best delay on 27 (67.5%), while being the only method to produce functionally correct results for all 40 benchmarks.

Functional Correctness. The differential testing pipeline enables POET to achieve 100% functional correctness (40/40), compared to 72.5% for I/O prompting (29/40), 82.5% for CoT (33/40), and 90.0% for REvolution (36/40). I/O, CoT, and REvolution fail on 11, 7, and 4 designs respectively, with failures concentrated on complex arithmetic and sequential circuits. For `fsm_encode` and `register`, where all three baselines fail, POET is the only method that succeeds by leveraging its differential testing pipeline to generate reliable testbenches from the original design, enabling comprehensive verification of the restructured state encoding and register access patterns.

Power-Centric PPA Optimization. POET achieves the best power on every design, demonstrating the effectiveness of the power-oriented evolutionary mechanism. The results highlight two key optimization capabilities of POET. First, when the design is not yet Pareto-optimal, POET improves all PPA metrics simultaneously. For example, `ticket_machine` achieves 31.8% area, 48.7% CPD, and 61.6% power reduction over the original, and `comparator_16bit` achieves 65.0% area, 24.1% CPD, and 67.9% power reduction over the original. Second, when the design is near the Pareto front, POET further reduces power through controlled trade-offs. For `comparator`, POET reduces power by 13.6% and area by 13.9% over the original while CPD increases slightly from 0.30 ns to 0.37 ns; for `gray`, POET achieves the lowest power (102.0μW, 12.8% below the best baseline) at the cost of increased area.

3.3 Ablation Study (RQ2)

We evaluate five ablation configurations, each removing one component from the full POET framework: (1) **w/o Differential Testing (DT)**, replacing the differential testing pipeline with LLM-generated testbenches; (2) **w/o Evolutionary Mechanism (EM)**, replacing evolutionary survivor selection with random selection; (3) **w/o Power Ranking (PR)**, using random intra-level selection instead of power-ascending ranking; (4) **w/o Proportional Allocation (PA)**, using standard NSGA-II sequential fill; and (5) **w/o Domain Knowledge (DK)**, removing circuit-specific techniques from operator prompts. Figure 2 reports the Best count for each setting, where each bar indicates the number of designs (out of 40) achieving the best result among all baselines (Original, I/O, CoT, REvolution).

w/o DT is the only setting that degrades functional correctness, with Func. Pass dropping from 40 to 35, which cascades into reduced Power Best (34), Area (24), and Delay (22) as unreliable LLM-generated testbenches compromise verification throughout evolution. **w/o EM** causes the largest degradation across all metrics (Power: 40→10, Area: 29→12, Delay: 27→11), confirming that evolutionary selection is the fundamental driver of optimization. **w/o PR** drops Power Best from 40 to 30 while Area and Delay Best shift to 31 and 30, revealing that without power-oriented ranking the search redistributes capacity to other metrics. **w/o PA** shows a smaller drop (Power: 40→38), indicating proportional allocation provides a consistent but moderate benefit through diversity preservation. **w/o DK** reduces Power Best to 31, Area to 25, and Delay to 26, demonstrating that hardware-specific techniques in operator prompts provide meaningful guidance for RTL transformations.

3.4 Case Study (RQ3)

Figure 3 traces POET's optimization trajectory on `alu_64bit` across 10 generations, where improvements are measured relative to the original design. In Phase 1 (generation 2), **Explore** decomposes the monolithic ALU into specialized submodules, achieving 33.7% CPD, 6.2% area, and 4.4% power reduction. In Phase 2 (generation 5), **Explore** applies resource sharing, further improving to 8.0% area and 6.7% power reduction while CPD reaches 35.7%. In Phase 3 (generation 8), **Simplify** applies operand isolation to eliminate idle switching activity, pushing power to 9.2% and area to 10.9%, while CPD settles at 26.5% due to serialization overhead. This trajectory exemplifies two capabilities of POET: collectively improving all PPA metrics through complementary operators (Phases 1 and 2),

and further reducing power through controlled trade-offs near the Pareto front at modest CPD cost (Phase 3).

4 Conclusion

We presented POET, a framework for power-centric RTL PPA optimization that addresses two fundamental challenges: functional correctness and multi-objective optimization directionality. The differential-testing-based testbench generation pipeline leverages the original design as a functional oracle, producing reliable verification through deterministic simulation rather than LLM reasoning. The power-oriented evolutionary mechanism combines non-dominated sorting, power-first intra-level ranking, and proportional survivor selection to systematically steer the search toward low-power solutions while preserving Pareto optimality, without manual weight tuning. Experiments on RTL-OPT demonstrate that POET achieves 100% functional correctness and the best power on all 40 designs, with competitive area and delay improvements. Ablation studies and a case study further validate each component's contribution.

References

[1] Manar Abdelatty, Maryam Nouh, Jacob K. Rosenstein, and Sherief Reda. 2025. Pluto: A Benchmark for Evaluating Efficiency of LLM-Generated Hardware Code. *arXiv preprint arXiv:2510.14756* (2025).

[2] Anantha P. Chandrakasan, Denis C. Daly, Joyce Kwong, and Yogesh K. Ramadass. 2008. Next Generation Micro-Power Systems. In *2008 IEEE Symposium on VLSI Circuits*. IEEE, 2–5.

[3] Ching Chang, Yidan Shi, Defu Cao, Wei Yang, Jeehyun Hwang, Haixin Wang, Jiacheng Pang, Wei Wang, Yan Liu, Wen-Chih Peng, et al. 2025. A survey of reasoning and agentic systems in time series with large language models. *arXiv preprint arXiv:2509.11575* (2025).

[4] Yiqun Chen, Jinyuan Feng, Wei Yang, Meizhi Zhong, Zhengliang Shi, Rui Li, Xiaochi Wei, Yan Gao, Yi Wu, Yao Hu, et al. 2026. Self-Compression of Chain-of-Thought via Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2601.21919* (2026).

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[6] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make Every Move Count: LLM-Based High-Quality RTL Code Generation Using MCTS. *arXiv preprint arXiv:2402.03289* (2024).

[7] Shukai Duan, Heng Ping, Nikos Kanakaris, Xiongye Xiao, Panagiotis Kyriakis, Nesreen K. Ahmed, Peiyu Zhang, Guixiang Ma, Mihai Capotă, Shahin Nazarian, Theodore L. Willke, and Paul Bogdan. 2024. A Structure-Aware Framework for Learning Device Placements on Computation Graphs. *Advances in Neural Information Processing Systems* 37 (2024), 81748–81772.

[8] Kevin Immanuel Gubbi, Marcus Halm, Sarbani Kumar, Arvind Sudarshan, Pavan Dheeraj Kota, Mohammadnavid Tarighat, Avesta Sasan, and Houman Homayoun. 2025. Prompting for Power: Benchmarking Large Language Models for Low-Power RTL Design Generation. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*. IEEE, 1–7.

[9] Wei-Po Hsin, Ren-Hao Deng, Yao-Ting Hsieh, En-Ming Huang, and Shih-Hao Hung. 2026. EvolVE: Evolutionary Search for LLM-based Verilog Generation and Optimization. *arXiv preprint arXiv:2601.18067* (2026).

[10] Shixuan Li, Wei Yang, Peiyu Zhang, Xiongye Xiao, Defu Cao, Yuehan Qin, Xiaole Zhang, Yue Zhao, and Paul Bogdan. 2025. Climatellm: Efficient weather forecasting via frequency-aware large language models. *arXiv preprint arXiv:2502.11059* (2025).

[11] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[12] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[13] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. 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)*. IEEE, 722–727.

[14] Yao Lu, Shang Liu, Hangan Zhou, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2026. A New Benchmark for the Appropriate Evaluation of RTL Code Optimization. *arXiv preprint arXiv:2601.01765* (2026).

[15] Kyungjun Min, Kyumin Cho, Junhwan Jang, and Seokhyeong Kang. 2025. REvolution: An Evolutionary Framework for RTL Generation Driven by Large Language Models. *arXiv preprint arXiv:2510.21407* (2025).

[16] Jingyu Pan, Guanglei Zhou, Chen-Chia Chang, Isaac Jacobson, Jiang Hu, and Yiran Chen. 2025. A Survey of Research in Large Language Models for Electronic Design Automation. *ACM Transactions on Design Automation of Electronic Systems* 30, 3 (2025), 1–21.

[17] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Transactions on Design Automation of Electronic Systems* 30, 6 (2025), 1–20.

[18] Heng Ping, Arijit Bhattacharjee, Peiyu Zhang, Shixuan Li, Wei Yang, Anzhe Cheng, Xiaole Zhang, Jesse Thomason, Ali Jannesari, Nesreen Ahmed, and Paul Bogdan. 2025. VeriMoA: A Mixture-of-Agents Framework for Spec-to-HDL Generation. *arXiv preprint arXiv:2510.27617* (2025).

[19] Heng Ping, Shixuan Li, Peiyu Zhang, Anzhe Cheng, Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Wei Yang, Shahin Nazarian, Andrei Irimia, and Paul Bogdan. 2025. HDLCore: A Training-Free Framework for Mitigating Hallucinations in LLM-Generated HDL. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 108–116.

[20] Biqing Qi, Zhouyi Qian, Yiang Luo, Junqi Gao, Dong Li, Kaiyan Zhang, and Bowen Zhou. 2024. Evolution of Thought: Diverse and High-Quality Reasoning via Multi-Objective Optimization. *arXiv preprint arXiv:2412.07779* (2024).

[21] Kimia Tasnia, Alexander Garcia, Tasnuva Farheen, and Sazadur Rahman. 2025. VeriOpt: PPA-Aware High-Quality Verilog Generation via Multi-Role LLMs. In *2025 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9.

[22] Kiran Thorat, Jiahui Zhao, Yaotian Liu, Amit Hasan, Hongwu Peng, Xi Xie, Bin Lei, and Caiwen Ding. 2025. 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)*. IEEE, 1–7.

[23] Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Bowei Tian, Shwai He, Guangyu Sun, Zhaoyang Shen, Sanmao Chen, and Anurag Srivastava. 2025. SymRTLO: Enhancing RTL Code Optimization with LLMs and Neuron-Inspired Symbolic Reasoning. *arXiv preprint arXiv:2504.10369* (2025).

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[25] Yangbo Wei, Zhen Huang, Huang Li, Wei W. Xing, Ting-Jung Lin, and Lei He. 2025. VFlow: Discovering Optimal Agentic Workflows for Verilog Generation. *arXiv preprint arXiv:2504.03723* (2025).

[26] Kangwei Xu, Denis Schwachhofer, Jason Blocklove, Ilia Polian, Peter Domanski, Dirk Pflüger, Siddharth Garg, Ramesh Karri, Ozgur Sinanoglu, Johann Knechtel, and Zhufei Zhao. 2025. Large Language Models (LLMs) for Electronic Design Automation (EDA). *arXiv preprint arXiv:2508.20030* (2025).

[27] Guang Yang, Wei Zheng, Xiang Chen, Dong Liang, Peng Hu, Yukui Yang, Shaohang Peng, Zhaoji Li, Junfeng Feng, Xinyao Wei, and Kai Sun. 2025. Large Language Model for Verilog Code Generation: Literature Review and the Road Ahead. *arXiv preprint arXiv:2512.00020* (2025).

[28] Wei Yang, Shixuan Li, Heng Ping, Peiyu Zhang, Paul Bogdan, and Jesse Thomason. 2026. Auditing Multi-Agent LLM Reasoning Trees Outperforms Majority Vote and LLM-as-Judge. *arXiv preprint arXiv:2602.09341* (2026).

[29] Wei Yang, Jiacheng Pang, Shixuan Li, Paul Bogdan, Stephen Tu, and Jesse Thomason. 2025. Maestro: Learning to Collaborate via Conditional Listwise Policy Optimization for Multi-Agent LLMs. *arXiv preprint arXiv:2511.06134* (2025).

[30] Wei Yang and Jesse Thomason. 2025. Learning to deliberate: Meta-policy collaboration for agentic llms with multi-agent reinforcement learning. *arXiv preprint arXiv:2509.03817* (2025).

[31] Wei Yang, Muyan Weng, Jiacheng Pang, Defu Cao, Heng Ping, Peiyu Zhang, Shixuan Li, Yue Zhao, Qiang Yang, Mengdi Wang, et al. 2025. Toward Evolutionary Intelligence: LLM-based Agentic Systems with Multi-Agent Reinforcement Learning. *Available at SSRN 5819182* (2025).

[32] Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. 2024. RTLRewriter: Methodologies for Large Models Aided RTL Code Optimization. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–7.

[33] Wen Ye, Wei Yang, Defu Cao, Yizhou Zhang, Lumingyuan Tang, Jie Cai, and Yan Liu. 2024. Domain-Oriented Time Series Inference Agents for Reasoning and Automated Analysis. *arXiv preprint arXiv:2410.04047* (2024).

[34] Zelin Zang, Yuhang Song, Aili Wang, Bingo Wing-Kuen Ling, Qi Sun, Zhen Lei, Fuji Yang, Cheng Zhuo, and Jiebo Luo. 2025. The Dawn of Agentic EDA: A Survey of Autonomous Digital Chip Design. *arXiv preprint arXiv:2512.23189* (2025).

[35] Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. 2025. ASPEN: LLM-Guided E-Graph Rewriting for RTL Datapath Optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*. IEEE, 1–9.

[36] Zeyue Zhang, Heng Ping, Peiyu Zhang, Nikos Kanakaris, Xiaoling Lu, Paul Bogdan, and Xiongye Xiao. 2025. MIHC: Multi-View Interpretable Hypergraph Neural Networks with Information Bottleneck for Chip Congestion Prediction. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[37] Yifang Zhao, Weimin Fu, Shijie Li, Yi-Xiang Hu, Xiaolong Guo, and Yier Jin. 2025. Hardware Generation with High Flexibility using Reinforcement Learning Enhanced LLMs. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.