AutoPPA: Automated Circuit PPA Optimization via Contrastive Code-based Rule Library Learning

Chongxiao Li[1,2], Pengwei Jin[1], Di Huang[1], Guangrun Sun[1,2], Husheng Han[1], Jianan Mu[1], Xinyao Zheng[1,2], Jiaguo Zhu[1,3], Shuyi Xing[1,3], Hanjun Wei[1,2], Tianyun Ma[4], Shuyao Cheng[1], Rui Zhang[1], Ying Wang[1], Zidong Du[1], Qi Guo[1], and Xing Hu[1]

[1] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] University of Science and Technology of China, Hefei, China
[4] Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, China

Abstract. Performance, power, and area (PPA) optimization is a fundamental task in RTL design, requiring a precise understanding of circuit functionality and the relationship between circuit structures and PPA metrics. Recent studies attempt to automate this process using LLMs, but neither feedback-based nor knowledge-based methods are efficient enough, as they either design without any prior knowledge or rely heavily on human-summarized optimization rules.

In this paper, we propose AutoPPA, a fully automated PPA optimization framework. The key idea is to automatically generate optimization rules that enhance the search for optimal solutions. To do this, AutoPPA employs an *Explore-Evaluate-Induce* (E^2I) workflow that contrasts and abstracts rules from diverse generated code pairs rather than manually defined prior knowledge, yielding better optimization patterns. To make the abstracted rules more generalizable, AutoPPA employs an adaptive multi-step search framework that adopts the most effective rules for a given circuit. Experiments show that AutoPPA outperforms both the manual optimization and the state-of-the-art methods SymRTLO and RTLRewriter.

1 Introduction

Performance, power, and area (PPA) constitute the fundamental metrics for evaluating integrated circuit design quality. As a critical design challenge, PPA optimization demands substantial expertise in hardware implementation, especially in RTL design, because it requires a precise understanding of circuit functionality and how circuit structures affect post-synthesis PPA results.

Recently, some large-language-model(LLM)-based efforts have attempted to automate this process. These studies fall into two categories: 1) Direct feedback methods that utilize post-synthesis PPA metrics as LLM inputs [4, 24]. While

straightforward, these approaches demonstrate limited efficacy because LLMs lack an understanding of the correlations between circuit structure and PPA metrics. 2) Knowledge-based methods that employ manually curated or human-summarized PPA optimization rules [29, 26]. Although potentially more targeted, these solutions face scalability challenges, for the labor-intensive knowledge base construction inherently limits both the coverage of circuit patterns and the diversity of optimization techniques, consequently limiting their practical utility. Even in some commercial tool guidebooks, only a few dozen sample entries are provided [22].

In this paper, to address the scalability problem in knowledge-based optimization, we investigate the fundamental problem: *Can we synthesize reusable PPA optimization knowledge automatically, without any human intervention, from just raw RTL code?* We observe that, leveraging the strong code-generation capabilities of LLMs, it is possible to rewrite functionally equivalent RTL codes with diverse structures. As shown in Figure 1, these contrastive code pairs can be used to induce the optimization rule library, without relying on manual rules.

However, synthesizing PPA optimization knowledge automatically remains critically challenging due to the following reasons: **Challenge 1: the evaluation of optimization rules.** Assessing rewritten RTL is inherently multifaceted. First, any candidate that is not functionally equivalent is unusable, so robust equivalence checking between any original RTL and its rewritten ones is essential. Second, beyond functional correctness, evaluation must balance PPA with the diversity of RTL samples. **Challenge 2: the high-quality rule induction.** Even when optimized code pairs are available, directly extracting high-quality rules is difficult. Naive summarization produces rules cluttered with low-level, implementation-specific idiosyncrasies, and different code pairs can produce redundant or contradictory rules. Such noisy rule sets are hard to apply and lead to suboptimal optimization in practice. **Challenge 3: the abstraction gap between rules and designs.** Overly specific rules introduce retrieval noise, while excessively abstract rules lack optimization value and can hardly benefit PPA results in practical usage. With a large-scale, automatically generated rule library, efficiently incorporating rules for PPA optimization is challenging because of the gap between abstract rules and specific designs. The rules provide general circuit descriptions, making it difficult for LLMs to locate relevant RTL code snippets and implement compliant modifications, resulting in functionally inequivalent or suboptimal results.

To this end, we propose a fully automated PPA optimization framework, AutoPPA, which includes an *Explore-Evaluate-Induce* (E^2I) workflow and an adaptive multi-step search method. As shown in Figure 1(b), the E^2I workflow (1) **explores** the Verilog code pairs via multiple rounds of random LLM rewriting sampling; (2) **evaluates** and verifies the functional equivalence between the original code and the rewritten code, and builds functional-equivalent Verilog code pairs via verification tools against **Challenge 1**; and (3) **induces** optimization rules with the form of (*snippet*, *condition*, *action*) triples by analysing paired code snippets and their PPA labels against **Challenge 2**. The adaptive

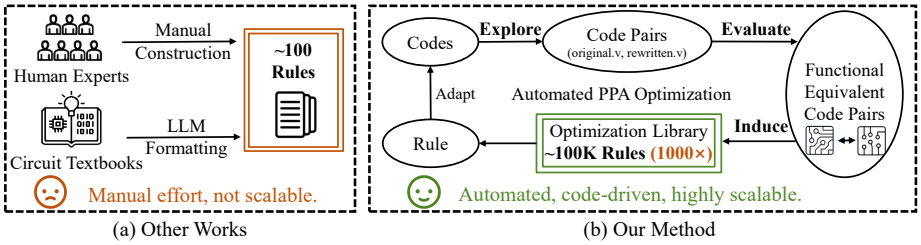

Fig. 1. Constructing the Rule Library for PPA optimization.

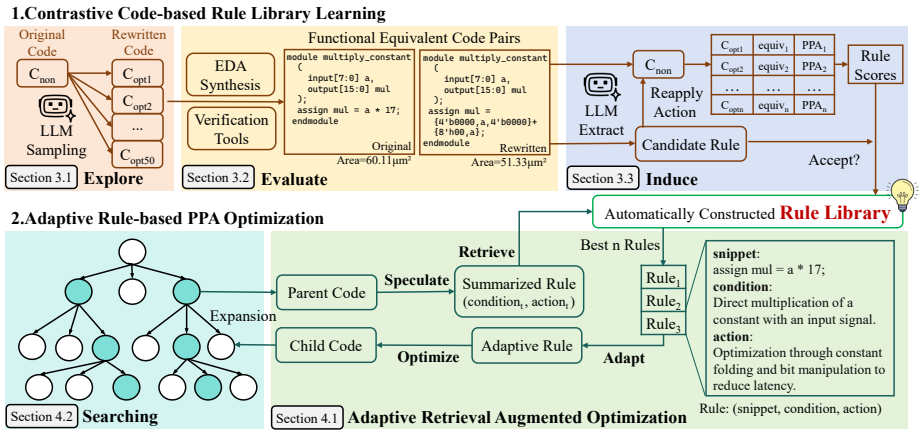

Fig. 2. Overview of AutoPPA. AutoPPA includes the pipeline of the rule library learning workflow and the adaptive rule-based PPA optimization.

multi-step search framework is a rule-enhanced beam search method that leverages the rule to better guide the LLM's exploration of higher-quality Verilog code samples, increasing the probability of PPA-optimized implementations against **Challenge 3**.

Experiments show that AutoPPA achieves a maximum of 15.31% area improvements and 11.28% delay improvements on 60 comprehensive benchmark circuits, and surpasses the area optimization results of manual optimization by 19.25% and state-of-the-art by 7.56% on 11 representative circuits, respectively.

2 Overview

2.1 Background: RTL PPA Optimization

From a circuit-quality perspective, integrated-circuit (IC) design proceeds through three main stages [9, 16]: At the Register-Transfer Level (RTL), engineers specify functionality in Verilog or VHDL; Automated logic synthesis then converts the RTL description into a gate-level netlist; Finally, place-and-route tools in the

physical-design stage produce the final layout while optimizing timing, power, and area. Although each stage affects PPA (performance, power, and area), RTL, sitting at the top of the design hierarchy, has the greatest influence: redundant or suboptimal RTL limits what downstream synthesis and physical tools can recover. Consequently, producing high-quality RTL is essential to meet stringent PPA targets. Despite advances in automation, RTL PPA optimization still relies heavily on expertise and manual tuning, as it is time-consuming and its outcomes can vary substantially with the designer's experience.

2.2 AutoPPA

As shown in Figure 2, AutoPPA consists of two phases: 1. Contrastive Code-based Rule Library Learning (Section 3), and 2. Adaptive Rule-based PPA Optimization (Section 4).

Contrastive Code-based Rule Library Learning is proposed to automatically construct a comprehensive rule library without any human intervention. It is composed of a *Explore-Evaluate-Induce* workflow, where we first explore optimization opportunities through code pair sampling, then evaluate the code pairs based on their equivalence and PPA, and finally induce natural language optimization rules from the code pairs.

Adaptive Rule-based PPA Optimization is a novel retrieval-augmented optimization approach proposed to exploit the learned rule library. It integrates single-step adaptive retrieval with multi-step rule-based enhanced search, enabling automatic selection and combination of the most effective rules for circuit PPA optimization.

3 Contrastive Code-based Rule Library Learning

Contrastive Code-based Rule Library Learning is proposed to construct the rule library from raw RTL code without any human intervention. It consists of three stages, *Explore-Evaluate-Induce*, whose details are shown as follows.

3.1 Explore: Code Sampling

In this stage, we aim to create a sufficiently large and diverse corpus of functionally valid RTL code base from which meaningful optimization rules can emerge. Approximately 130K Verilog RTL codes are crawled from public GitHub repositories. These codes undergo a two-phase synthesis process:

Lightweight Filtering: We initially synthesize all collected designs using the lightweight `Yosys` tool [27] to discard any non-self-contained or non-synthesizable designs.

Accurate Synthesis and PPA Metrics: The remaining filtered codes are then synthesized using `SiliconCompiler` [15], which combines `Yosys` and `OpenSTA` [2] to provide more accurate PPA measurements at the cost of increased synthesis time. Both the RTL code and the associated PPA performance metrics are stored for subsequent rule library learning.

With a large corpus of original circuit code, we explore diverse rewrite forms of the initial designs. General-purpose language models lack accurate and specialized knowledge for circuit optimization but show strong ability to sample massive and varied code. We leverage this ability to generate multiple rewrites of the initial circuits for potential optimization insight discovery. We first select 100K RTL codes with mid-range area values from the filtered corpus. Each code is rewritten N (default N = 50) times by the LLM Qwen2.5-Coder-7B-Instruct [10], producing over 5M pairs of circuit codes. These extensively explored pairs are subsequently passed to the evaluation stage to identify and retain those that may contain valuable optimization insights.

3.2 Evaluate: Contrastive Code Pairs Evaluation Using EDA Tools

In this stage, we aim to evaluate the massive code pairs, identifying functionally equivalent pairs that exhibit significant differences in PPA through evaluation, laying the groundwork for extracting robust optimization rules. After obtaining a large number of code pairs with diverse rewrite patterns, we first evaluate their PPA metrics and functional equivalence.

For PPA evaluation, we adopt the same settings as in Section 3.1. Functional equivalence evaluation is more challenging because it is difficult to construct testbenches for circuits with diverse trigger patterns, timing behaviors, and functionalities. To address this, we develop an automatic testbench generator. Using `Yosys`, we automatically extract the top module, port, clock and reset information of each circuit. We then construct a testbench for all code pairs. The testbench fully verifies clock and reset behaviors, applies multiple long random stimulus sequences, and compares the outputs of the original and rewritten circuits to verify equivalence. The generated testbenches achieve 100% line coverage and high reliability on our test circuits.

After verifying and synthesizing all rewrites of each RTL design, we compute the Shannon entropy of the PPA distribution among functionally equivalent versions to measure diversity. A higher entropy value indicates greater diversity in possible optimization patterns that can be extracted. Non-equivalent codes are given zero PPA improvement and excluded from the calculation. We rank designs by entropy and select the top $K\%$ ($K = 25$) as candidates. For each selected design, we keep only equivalent code pairs with relative PPA differences above 5%, denoted as (C_{non}, C_{opt}) for non-optimized and optimized versions.

3.3 Induce: Contrastive-Code Based Rule Induction

In this stage, we summarize key optimization insights into higher-level rules. To extract actionable circuit optimization strategies, we define each rule as a triplet (*snippet*, *condition*, *action*):

snippet: Representative low-efficiency RTL code snippets that serve as optimization targets.

condition: Generalized contextual conditions describing when the optimization rule becomes applicable.
action: Transformation actions specifying how to modify original RTL codes to achieve optimized PPA.

For each pair (C_{non}, C_{opt}), we first generate a set of candidate rules via the LLM, leading to n (default $n = 2$) distinct rules that might explain the improvement observed in C_{opt}.

We then identify high-quality rules through systematic evaluation. Each rule is reapplied to the original circuit with multiple LLM optimization attempts, and compared with the original non-optimized and optimized circuit PPA in the code pair. We normalize outcomes using Eq 1:

$$s_i = \mathbf{1}_{\text{eq}}(i) \cdot \text{clip}_{[0,1]}\left(\alpha + \beta \frac{\text{PPA}_n - \text{PPA}_i}{\text{PPA}_n - \text{PPA}_o}\right) \tag{1}$$

where s_i denotes the score of the i-th rewritten circuit. PPA_n, PPA_o, and PPA_i are the PPA values of the non-optimized circuit C_{non}, the optimized circuit C_{opt}, and the i-th rewrite, respectively. $\mathbf{1}_{\text{eq}}(i)$ ensures that only functionally equivalent rewrites receive nonzero scores. The parameters α (default 0.25) and β (default 0.5) normalize the improvement: an equivalent rewrite with no PPA gain receives a score of 0.25, while one matching the optimized circuit's improvement receives 0.75.

Scores are then averaged across all attempts per rule. Rules with average scores above 0.7 are classified as high-quality and added into the rule library, indicating strong circuit improvement potential.

4 Adaptive Rule-based PPA Optimization

To fully leverage the optimization rule library in practical scenarios, we propose an integrated framework called *Adaptive Rule-based PPA Optimization*. This framework integrates single-step Adaptive Retrieval Augmented Optimization (ARAO) with multi-step Rule-based Enhanced Searching. The single-step optimization extracts structural conditions from the input RTL code and applies the most adaptive actions to the circuits based on rules, while the multi-step searching iteratively refines candidates via beam search, balancing both PPA gains and circuit diversity.

4.1 Adaptive Retrieval Augmented Optimization

To harness the optimization rule library for RTL code PPA optimization, we retrieve rules by matching structural conditions and potential actions in the RTL code, and use the retrieved rules to guide LLMs to optimize the RTL code. We observe that some rules contain descriptions targeting their source code-pairs, which may contaminate the optimization context of the current circuit. We also observe that direct retrieval by semantic embedding similarity may

introduce noise. Therefore, we design an Adaptive Retrieval Augmented Optimization (ARAO) process that reduces the impact of these issues. This process contains four steps:

Speculate: The LLM is prompted to summarize an optimization rule ($snippet_t$, $condition_t$, $action_t$) for the target code.
Retrieve: Using the $condition_t$ and $action_t$, we retrieve the three most similar rules from the rule library based on cosine similarity between rule embeddings.
Adapt: To mitigate interference from source-specific information in retrieved rules, we utilize the LLM to adapt these rules with the target code, ensuring their applicability and relevance.
Optimize: The target code and adaptive rules are input to the LLM, which generates optimized code variants. These variants are then verified for functional equivalence and synthesized to assess PPA improvements.

4.2 Rule-based Enhanced Searching

Although ARAO is effective for single-step optimization, comprehensive PPA improvement often requires efficient search in a large search space of complex code optimization. We thus propose a multi-step search framework based on beam search. At each iteration, the top-k scored candidate codes (beam width k) are selected. For each, the ARAO process extends m optimized variants. The next iteration's candidate pool comprises all new variants and the current top candidates, up to a maximum of s iterations.

The candidate scoring function is defined as:

$$\text{Score} = \omega \cdot \text{Score}_{diversity} + (1 - \omega) \cdot \text{Score}_{ppa} \tag{2}$$

where $\text{Score}_{diversity}$ quantifies diversity using TF-IDF similarity [1] to parent codes, Score_{ppa} incorporates functional equivalence and relative PPA performance, and $\omega = 0.25$. This encourages both PPA improvement and exploration of diverse optimization strategies.

5 Experiments

We first introduce our experimental setup in Section 5.1. Then, we conduct comprehensive experiments and thorough ablation studies to validate the effectiveness of our approach in Section 5.2.

5.1 Experimental Setup

Benchmarks. We use the RTLRewriter [29] benchmark, which is specifically designed for RTL code optimization. It contains 54 designs with comprehensive optimization patterns and 3 practical, large designs that are synthesizable. Following RTLRewriter, we partition these 3 practical designs into 6 distinct

modules, resulting in a total of 60 designs. For each design, we generate a rigorous testbench (100% line and branch coverage on non-redundant code) to verify the equivalence between the optimized code and the original implementation.

Baselines. ***(1) SOTA RTL code optimization methods.*** To compare with RTLRewriter, we synthesize all 11 optimized circuits from their public repository using our aligned synthesis flow. For SymRTLO [26], we compare our results with their self-reported results on 5 circuits that implement complex algorithms. ***(2) Proficient Verilog engineers***. We compare our method with results manually optimized by a Verilog engineer with over two years of experience on the complete benchmark. The engineer is provided with standard verification and synthesis environments and allowed 16 working hours to perform thorough optimizations while summarizing optimization rules during this process. ***(3) Representative LLMs.*** We compare our results with representative LLMs, listed in Table 1. For a fair comparison, we evaluate the most optimized code generated by each LLM under an identical amount of sampled RTL code. In all experiments, we use a unified generation temperature of 0.6 and provide prompts that are consistent with our approach.

Metrics. We report the synthesized total cell area (in μm^2), cycle delay (in ns), and dynamic power (in mW) to evaluate circuit area, performance, and power. Unless otherwise specified, all syntheses use `SiliconCompiler` with the FreePDK 45nm process [21]. For equivalent and optimized circuits, we report their PPA improvement according to $Impr = 1 - PPA_{opt}/PPA_{original}$. If the generated circuits are inequivalent or with worse target PPA, we mark them as 0 improvement. For the vanilla LLM baseline, following the widely adopted $Pass@k$ metric [5] in code generation tasks, we propose the **Impr@k** metric to estimate the expected optimal PPA optimization from k samples, using the results from a total of $n > k$ samples, as shown in Equation 3:

$$\text{Impr@k} := \mathbb{E}_{\text{circuits}}\left[\sum_{j=1}^{n}\left(Impr_j^{\downarrow}\cdot\frac{\binom{n-j}{k-1}}{\binom{n}{k}}\right)\right] \tag{3}$$

where the $Impr_j^{\downarrow}$ denotes the j-th largest PPA improvements in n samples , i.e., $Impr_1^{\downarrow} \geq Impr_2^{\downarrow} \geq \cdots \geq Impr_n^{\downarrow}$.

AutoPPA settings. We configure different search scales according to Table 2, generating 15, 50, 104, and 210 RTL code samples for each configuration. We utilize gte_Qwen2-7B-instruct [12] as the embedding model to embed the rule library and retrieval queries. We configure AutoPPA with both smaller Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct models and larger models DeepSeek-V3-0324 model during search to show the generality of our framework. We set the LLM generation temperature to 0.6 for all models.

5.2 Experimental Results

AutoPPA surpasses manual effort and SOTA methods. *(1) AutoPPA outperforms RTLRewriter and manual optimizations on area-oriented circuit optimizations.* We conduct a circuit-by-circuit comparison against RTLRewriter's

Table 1. Baseline language models in our experiments.

Type	Model	Abbr.	# Params
RTL Specific	CodeV-All-QC	CodeV	7.62B
	HaVen-DeepSeek	HaVen	6.74B
Coding Specific	Qwen2.5-Coder-7B-Instruct	Qwen Coder	7.62B
Reasoning	DeepSeek-R1-Distill-Qwen-7B	DS-R1-Dist	7.62B
General Purpose	DeepSeek-V3-0324	DeepSeek-V3	685B

Table 2. Configurations for AutoPPA's searching scales, where $n = (1 + k \cdot (s - 1)) \cdot m$.

Config Abbr.	Beam Width Num (k)	Expand Max Steps (m)	(s)	Total RTL Searched (n)
2-3-3	2	3	3	15
3-5-4	3	5	4	50
3-8-5	3	8	5	104
5-10-5	5	10	5	210

open-sourced optimized RTL code and manually optimized results. We employ DeepSeek-V3 as AutoPPA's backbone model with 5-10-5 search settings, and align the optimization target with RTLRewriter for area-oriented optimization. The results are shown in Table 3, with the best results highlighted in bold and results that fail to achieve optimization compared to the original design shown in light color. Our method achieves superior performance, obtaining the smallest area in 10 out of 11 circuits. On average, compared to RTLRewriter, our approach delivers significant improvements: 7.56% in area and 9.00% in power. The relatively poor performance of manual optimization demonstrates that optimizing these circuits presents considerable challenges even for human engineers with proficient experience.

Table 3. Area-oriented optimization comparison of AutoPPA with manual efforts and RTLRewriter. AutoPPA outperforms manual efforts by 19.25% and RTLRewriter by 7.56%.

Design	Original	Manual	RTLRewriter	AutoPPA-DS
	Area / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power
add3	65.17 / 0.26 / 0.85	65.17 / 0.26 / 0.85	**42.83** / 0.21 / 0.62	65.17 / 0.26 / 0.85
mux_type1	2.39 / 0.04 / 0.00010	2.39 / 0.04 / 0.00010	**2.13** / 0.04 / 0.00010	**2.13** / 0.04 / 0.00010
mux_type3	2.39 / 0.05 / 0.00010	2.39 / 0.05 / 0.00010	2.39 / 0.05 / 0.00010	2.39 / 0.05 / 0.00010
mux_type5	6.12 / 0.13 / 0.00010	6.12 / 0.13 / 0.00010	6.12 / 0.13 / 0.00010	6.12 / 0.13 / 0.00010
example1	61.71 / 0.22 / 0.78	60.116 / 0.21 / 0.87	48.15 / 0.24 / 0.84	**29.26** / 0.23 / 0.68
example3	52.40 / 0.27 / 2.33	49.476 / 0.31 / 2.07	37.51 / 0.21 / 1.54	**36.44** / 0.21 / 1.25
com_subexp	11967.34 / 2.98 / 0.26	11370.17 / 3.00 / 0.24	11389.58 / 3.12 / 0.24	**11310.85** / 3.06 / 0.24
add_bit_wid	63.57 / 0.32 / 0.0016	63.57 / 0.32 / 0.0016	63.57 / 0.32 / 0.0016	**36.44** / 0.59 / 0.00070
add_subexp	132.73 / 0.57 / 0.0029	132.73 / 0.57 / 0.0029	132.74 / 0.57 / 0.0029	132.73 / 0.57 / 0.0029
m_con_mul	1374.16 / 1.17 / 0.031	1374.16 / 1.17 / 0.031	1026.76 / 1.13 / 0.023	**876.47** / 1.04 / 0.02
m_con_mul2	1628.72 / 1.22 / 0.038	1628.72 / 1.22 / 0.038	1370.17 / 1.22 / 0.031	**873.01** / 1.16 / 0.021
Avg. Impr.	-	1.20% / -0.99% / 0.60%	12.89% / 2.83% / 9.35%	**20.45%** / -4.85% / 18.35%

(2) AutoPPA outperforms SymRTLO on more challenging optimization tasks regarding complex and large circuits. Since SymRTLO has not open-sourced its

Table 4. Comparison of the area-oriented optimization results of AutoPPA, manual optimization, vanilla DeepSeek-V3 sampling, and SymRTLO, synthesizing with `Design Compiler`. Results for SymRTLO are copied from their original paper.

Design		spmv	subexp_elim	adder_architecture
Metric		**Area** / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power
SSC	Original	22908.69 / 7.95 / 1.46	9484.15 / 11.78 / 4.61	541.92 / 2.29 / 0.17
	SymRTLO	22908.69 / 7.95 / 1.46	6791.88 / 11.78 / 3.53	531.88 / 2.48 / 0.17
SMIC 12nm	Original	423.94 / 0.16 / 6.71	162.75 / 1.27 / 0.22	8.48 / 0.15 / 0.01
	Manual	423.94 / 0.16 / 6.71	162.75 / 1.27 / 0.22	8.48 / 0.15 / 0.01
	DeepSeek-V3	**295.28** / 0.19 / 4.36	123.54 / 1.27 / 0.18	8.48 / 0.15 / 0.01
	AutoPPA-DS	**295.28** / 0.19 / 4.36	**116.81** / 1.27 / 0.17	**8.48** / 0.14 / 0.01
TSMC 65nm	Original	4021.20 / 0.60 / 4.61	1588.68 / 4.03 / 0.97	80.64 / 0.69 / 0.04
	Manual	4021.20 / 0.60 / 4.61	1588.68 / 4.03 / 0.97	80.64 / 0.69 / 0.04
	DeepSeek-V3	**2904.12** / 0.59 / 3.54	1195.92 / 4.03 / 0.78	80.64 / 0.69 / 0.04
	AutoPPA-DS	**2904.12** / 0.59 / 3.54	**1130.04** / 4.03 / 0.75	**79.92** / 0.71 / 0.04

Design		vending_machine	fft	**Avg. Impr.**
Metric		**Area** / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power
SSC	Original	240079.30 / 7.90 / 11.46	1857805.00 / 7.90 / 51.12	-
	SymRTLO	151593.90 / 7.90 / 8.18	1471378.00 / 8.98 / 26.32	17.58% / -4.39% / 20.12%
SMIC 12nm	Original	3747.36 / 0.23 / 17.59	31271.32 / 0.75 / 46.68	-
	Manual	3747.36 / 0.23 / 17.59	29346.88 / 0.81 / 37.23	1.23% / -0.27% / 5.27%
	DeepSeek-V3	3608.39 / 0.23 / 17.59	31271.32 / 0.75 / 46.68	11.63% / -2.42% / 12.30%
	AutoPPA-DS	**2966.91** / 0.36 / 9.70	**27769.19** / 0.79 / 42.85	**18.12%** / -14.79% / 23.57%
TSMC 65nm	Original	38829.96 / 0.72 / 26.75	304913.90 / 2.63 / 39.05	-
	Manual	38829.96 / 0.72 / 26.75	295164.72 / 2.51 / 35.67	0.64% / 0.91% / 1.73%
	DeepSeek-V3	37383.12 / 0.75 / 24.96	304913.88 / 2.63 / 39.05	11.25% / -0.50% / 10.00%
	AutoPPA-DS	**29798.64** / 1.12 / 13.51	**262633.70** / 2.73 / 35.47	**18.93%** / -12.12% / 21.85%

optimized RTL code or synthesis scripts, we cannot conduct comparisons under identical standards. In Table 4, we present the self-reported area optimization results from SymRTLO for circuits with complex functionality, manual optimizations, vanilla LLM results, as well as AutoPPA results, using DeepSeek-V3 [7] as the backbone model with 5-10-5 search settings. To best align with SymRTLO's synthesis settings, we use `Design Compiler` 2018, synthesizing circuits into SMIC 12nm and TSMC 65nm processes, respectively. The experimental results demonstrate that AutoPPA achieves higher area improvement in circuit area compared with SymRTLO in both processes (18.12% / 18.93% *v.s.* 17.58%). However, we emphasize again that differences in process nodes and synthesis scripts can lead to significant variations in PPA outcomes. Since the specific process node and scripts used by SymRTLO were inaccessible to us, achieving a fully fair comparison remains a challenge.

(3) AutoPPA outperforms all baseline LLM methods on both area and delay-oriented optimizations. We extend our evaluation to compare our method with baseline LLMs across all *60 circuits* in the complete benchmark, applying settings from 2-3-3 to 5-10-5. Using Qwen2.5-7B series models and DeepSeek-V3 as backbone models for search, we compare against the language models listed in Table 1 targeting either area or delay optimization. The results are reported in Tables 5 and illustrated in Figure 3.

The results in Tables 5 demonstrate that our method achieves the best optimization among models of similar size for both area and delay optimization objectives. Importantly, our method shows consistent improvement as the search budget increases. Figure 3 further illustrates the comparison between AutoPPA and the vanilla DeepSeek-V3 for optimization: as the number of samples

Table 5. Comparison of area-oriented and delay-oriented optimization with vanilla LLMs across 60 circuits.

Area-oriented Optimization					
Method		*Impr*@15 (%)	*Impr*@50 (%)	*Impr*@104 (%)	*Impr*@210 (%)
		Area / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power	**Area** / Delay / Power
RTL-Specific Models	HaVen	1.33% / 0.12% / 1.48%	3.02% / 0.33% / 3.46%	4.14% / 0.51% / 4.82%	4.77% / 0.60% / 5.57%
	CodeV	1.35% / 0.47% / 1.61%	3.10% / 1.06% / 3.53%	4.36% / 1.63% / 4.75%	5.32% / 2.27% / 5.45%
Reasoning	DS-R1-Dist	3.74% / 0.55% / 3.86%	6.01% / 1.21% / 6.27%	7.78% / 1.90% / 8.11%	9.22% / 2.76% / 9.55%
General Models	Qwen-Coder	4.79% / 1.29% / 5.52%	6.90% / 1.97% / 8.00%	8.74% / 2.87% / 10.10%	10.60% / 3.97% / 12.37%
	DeepSeek-V3	8.98% / 1.63% / 9.95%	10.55% / 1.85% / 11.40%	11.04% / 2.09% / 11.97%	11.20% / 2.36% / 12.29%
Ours	AutoPPA-Qw	4.98% / 2.12% / 5.41%	8.83% / 3.25% / 9.97%	9.50% / 2.94% / 11.13%	11.05% / 4.87% / 14.31%
	AutoPPA-DS	**9.56%** / 1.66% / 10.84%	**11.91%** / 2.57% / 13.44%	**12.45%** / 2.05% / 13.58%	**15.31%** / 0.43% / 15.97%
Delay-oriented Optimization					
Method		*Impr*@15 (%)	*Impr*@50 (%)	*Impr*@104 (%)	*Impr*@210 (%)
		Delay / Area / Power	**Delay** / Area / Power	**Delay** / Area / Power	**Delay** / Area / Power
RTL-Specific Models	HaVen	1.00% / 0.46% / 0.93%	1.88% / 1.38% / 1.89%	2.33% / 2.11% / 2.60%	2.49% / 2.97% / 3.30%
	CodeV	1.92% / 1.98% / 2.50%	2.49% / 2.49% / 3.36%	2.80% / 2.39% / 3.34%	3.15% / 1.69% / 2.61%
Reasoning	DS-R1-Dist	1.51% / 1.87% / 2.32%	2.52% / 2.47% / 3.00%	3.02% / 2.48% / 3.06%	3.33% / 2.00% / 2.56%
General Models	Qwen-Coder	2.76% / 3.61% / 4.51%	4.61% / 4.02% / 5.11%	6.11% / 4.56% / 5.89%	7.74% / 5.83% / 7.49%
	DeepSeek-V3	4.20% / 1.65% / 2.90%	5.15% / 2.11% / 3.44%	5.67% / 2.48% / 3.96%	6.37% / 2.90% / 4.49%
Ours	AutoPPA-Qw	4.65% / 3.10% / 4.62%	**6.15%** / 4.82% / 5.52%	**9.80%** / 9.75% / 11.79%	**11.28%** / 8.00% / 10.87%
	AutoPPA-DS	**5.36%** / 1.09% / 1.54%	5.87% / 0.73% / 1.74%	7.14% / 2.10% / 2.44%	9.95% / 6.37% / 7.78%

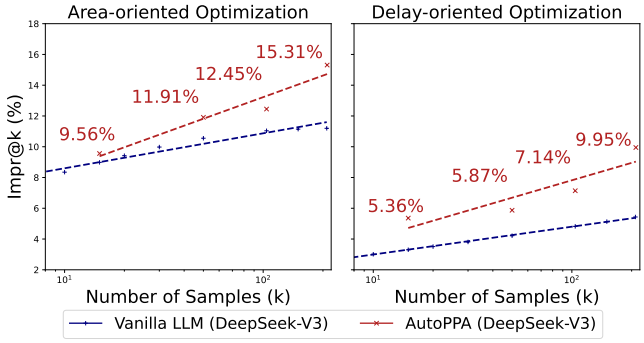

Fig. 3. Area and Delay improvement comparison with vanilla LLM sampling. AutoPPA yields consistently higher impr@k and better growth rate than DeepSeek-V3.

grows, AutoPPA yields consistently higher impr@k and better growth rate than DeepSeek-V3. At the largest search budget, our method outperforms vanilla LLM by 4.11% for area-oriented optimization and 3.58% for delay-oriented optimization.

Notably, when targeting delay optimization, utilizing the smaller Qwen2.5-7B series models as a search backbone yields better results than using DeepSeek-V3. We attribute this to Qwen2.5-7B's superior baseline performance on delay optimization tasks compared to DeepSeek-V3. Despite being distilled from DeepSeek-R1 and fine-tuned on Qwen2.5-Math-7B, DS-R1-Distill's optimization capability falls short of the Qwen2.5-Coder-Instruct model with the same architecture and parameter scale, suggesting that distillation of reasoning models leads to performance degradation on specific tasks. HaVen and CodeV, despite being specialized models for RTL generation, demonstrate the lowest optimiza-

Table 6. Area-oriented optimization comparison between AutoPPA's learned rule library with the manually constructed one, tested with AutoPPA-Qw.

Settings	*Impr*@50 (%)
	Area / Delay / Power
AutoPPA-Qw with E^2I rules (ours)	**8.83%** / 3.25% / 9.97%
with manual rules	6.56% / 0.80% / 5.46%

tion on the RTL optimization task. This highlights the importance of multi-task fine-tuning for promoting domain-specific capabilities.

AutoPPA is effective on various synthesis settings. Combining results from Table 3 and Table 4, AutoPPA shows significant PPA improvements on both open-source and commercial EDA toolchains. This result highlights the wide applicability of AutoPPA. Notably, AutoPPA achieves significant area optimization across 12nm to 65nm process nodes in 2 large, practical circuits from RTLRewriter Benchmark (*fft* and *vending_machine*), reaching 11.20% and 20.83% in the 12nm process, and 13.87% and 23.26% in the 65nm process. We observe that, although the same circuit exhibits similar optimization trends across different process nodes, there are noticeable differences in the specific optimization ratios. This underscores the importance of employing unified synthesis toolchains and process nodes to ensure fair evaluation across different works. To the best of our knowledge, we are the first work to evaluate LLM-based RTL code PPA optimization across multiple EDA tools (`SiliconCompiler` and `Design Compiler`) and processes (from 12nm, 45nm, to 65nm).

AutoPPA's rule library surpasses manually crafted rule libraries. We conduct an experiment to compare the effectiveness of optimization rules automatically derived from our *explore-evaluate-induce* (E^2I) process against those manually summarized by Verilog engineers. The comparison uses an identical 3-5-4 search setting, employing Qwen2.5 series models as the search backbone for area-oriented optimization on all 60 circuits. The only difference is the rule library used during optimization. Table 6 presents the experimental results. The scalability of our E^2I process generates significantly more rules (101,987 rules) compared to 16 hours of manual efforts (12 rules), resulting in superior optimization outcomes.

Ablation Study. To validate the effectiveness of each component in our proposed approach, we conduct a series of ablation studies. Our complete design incorporates LLM-generated rule **speculation**, rule library **retrieval** with these speculative results, **adaptation** of retrieved rules, and the **search** framework. We perform experiments for ablations under equal RTL code sampling overhead. Table 7 presents the experimental results, which demonstrate the efficacy of each component in our design. The performance degradation observed in each ablation setting validates our integrated approach for optimal performance.

Table 7. Ablation on AutoPPA's adaptive rule-based PPA optimization framework on area-oriented optimizations.

Settings	*Impr*@50 (%)
	Area / Delay / Power
AutoPPA-Qw (ours)	**8.83**% / 3.25% / 9.97%
w/o Search	7.83% / 3.14% / 7.94%
w/o Adapt	6.96% / 2.58% / 7.62%
w/o Retrieve, Adapt	5.34% / 2.41% / 5.67%
w/o Speculate, Retrieve, Adapt	6.39% / 1.95% / 5.84%

6 Related work

LLM-based RTL Generation. Recent works leverage Large Language Models (LLMs) to assist circuit design flows. Frameworks such as Chip-Chat [3], OriGen [6], RTLFixer [25], and VerilogCoder [8] use self-reflection [20] or multi-agent systems for RTL generation. Others, including VeriGen [23], RTLCoder [13], BetterV [18], CodeV [30, 31], and HaVen [28], build domain-specific datasets and fine-tune LLMs to improve generation quality. However, these efforts primarily address circuit generation [14, 19] rather than circuit optimization challenges.

LLM-based RTL Optimization. Circuit optimization differs from generation tasks as it requires maintaining functional equivalence while understanding the complex relation among hardware code, circuit structure, and post-synthesis QoR metrics. Some approaches attempt to address circuit optimization directly. ChipGPT [4] and VeriPPA [24] feed post-synthesis PPA metrics back to LLMs and request optimization directly. RTLRewriter [29] builds a manual knowledge base, retrieves information from code and diagrams, and applies Monte Carlo Tree Search (MCTS) for rewriting. SymRTLO [26] formats manually collected optimization materials [11, 17] for retrieval during iterative refinement. These approaches depend on a manually constructed knowledge base with limited entries and task-specific optimization capabilities.

7 Conclusion

In this paper, we present AutoPPA, a fully automated PPA optimization framework comprising an automatically generated optimization rule library and an enhanced optimization search method. To mitigate the scarcity of optimization rules and the challenges in rule induction, we develop a rule library learning framework based on an *Explore-Evaluate-Induce* workflow. To address the inefficiency in applying abstract optimization rules to concrete designs, we propose an Adaptive Retrieval Augmented Optimization (ARAO) mechanism integrated with a multi-step search strategy. Experimental results demonstrate that AutoPPA surpasses baseline LLM methods, achieves a 19.25% improvement in area optimization over manual methods and a 7.56% improvement over RTLRewriter.

References

1. Aizawa, A.: An information-theoretic perspective of tf–idf measures. Information Processing & Management **39**(1), 45–65 (2003)
2. Ajayi, T., Blaauw, D.: Openroad: Toward a self-driving, open-source digital layout implementation tool chain. In: Proceedings of Government Microcircuit Applications and Critical Technology Conference (2019)
3. Blocklove, J., Garg, S., Karri, R., Pearce, H.: Chip-chat: Challenges and opportunities in conversational hardware design. In: 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD). pp. 1–6. IEEE (2023)
4. Chang, K., Wang, Y., Ren, H., Wang, M., Liang, S., Han, Y., Li, H., Li, X.: Chipgpt: How far are we from natural language hardware design (2025), https://arxiv.org/abs/2305.14019
5. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., et al.: Evaluating large language models trained on code (2021), https://arxiv.org/abs/2107.03374
6. Cui, F., Yin, C., Zhou, K., Xiao, Y., Sun, G., Xu, Q., Guo, Q., Liang, Y., Zhang, X., Song, D., 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)
7. DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., et al.: Deepseek-v3 technical report (2025), https://arxiv.org/abs/2412.19437
8. Ho, C.T., Ren, H., Khailany, B.: Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 39, pp. 300–307 (2025)
9. Huang, G., Hu, J., He, Y., Liu, J., Ma, M., Shen, Z., Wu, J., Xu, Y., Zhang, H., Zhong, K., et al.: Machine learning for electronic design automation: A survey. ACM Transactions on Design Automation of Electronic Systems (TODAES) **26**(5), 1–46 (2021)
10. Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., Feng, Y., Ren, X., Ren, X., Zhou, J., Lin, J.: Qwen2.5-coder technical report (2024), https://arxiv.org/abs/2409.12186
11. Knoop, J., Rüthing, O., Steffen, B.: Partial dead code elimination. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. p. 147–158. PLDI '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/178243.178256, https://doi.org/10.1145/178243.178256
12. Li, Z., Zhang, X., Zhang, Y., Long, D., Xie, P., Zhang, M.: Towards general text embeddings with multi-stage contrastive learning (2023), https://arxiv.org/abs/2308.03281
13. Liu, S., Fang, W., Lu, Y., Zhang, Q., Zhang, H., Xie, Z.: Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In: 2024 IEEE LLM Aided Design Workshop (LAD). pp. 1–5. IEEE (2024)
14. Liu, S., Lu, Y., Fang, W., Li, M., Xie, Z.: 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. ICCAD '24, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3676536.3697118, https://doi.org/10.1145/3676536.3697118

15. Olofsson, A., Ransohoff, W., Moroze, N.: A distributed approach to silicon compilation: Invited. In: Proceedings of the 59th ACM/IEEE Design Automation Conference. p. 1343–1346 (2022)
16. Pan, J., Zhou, G., Chang, C.C., Jacobson, I., Hu, J., Chen, Y.: A survey of research in large language models for electronic design automation. ACM Trans. Des. Autom. Electron. Syst. **30**(3) (Feb 2025). https://doi.org/10.1145/3715324, https://doi.org/10.1145/3715324
17. Pasko, R., Schaumont, P., Derudder, V., Vernalde, S., Durackova, D.: A new algorithm for elimination of common subexpressions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18**(1), 58–68 (1999). https://doi.org/10.1109/43.739059
18. Pei, Z., Zhen, H.L., Yuan, M., Huang, Y., Yu, B.: Betterv: Controlled verilog generation with discriminative guidance. arXiv preprint arXiv:2402.03375 (2024)
19. Pinckney, N., Batten, C., Liu, M., Ren, H., Khailany, B.: Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. ACM Trans. Des. Autom. Electron. Syst. **30**(6) (Oct 2025). https://doi.org/10.1145/3718088, https://doi.org/10.1145/3718088
20. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems **36** (2024)
21. Stine, J.E., Castellanos, I., Wood, M., Henson, J., Love, F., Davis, W.R., Franzon, P.D., Bucher, M., Basavarajaiah, S., Oh, J., et al.: Freepdk: An open-source variation-aware design kit. In: 2007 IEEE international conference on Microelectronic Systems Education (MSE'07). pp. 173–174. IEEE (2007)
22. Synopsys, I.: Coding guidelines for datapath synthesis. Tech. rep., Synopsys, Inc. (June 2012), https://solvnet.synopsys.com/retrieve/print/015771.html, doc Id: 015771, Product: Design Compiler, Last Modified: 06/01/2012
23. Thakur, S., Ahmad, B., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R., Garg, S.: Verigen: A large language model for verilog code generation. ACM Transactions on Design Automation of Electronic Systems **29**(3), 1–31 (2024)
24. Thorat, K., Zhao, J., Liu, Y., Peng, H., Xie, X., Lei, B., Zhang, J., Ding, C.: Advanced large language model (llm)-driven verilog development: Enhancing power, performance, and area optimization in code synthesis (2024), https://arxiv.org/abs/2312.01022
25. Tsai, Y., Liu, M., Ren, H.: Rtlfixer: Automatically fixing rtl syntax errors with large language model. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. pp. 1–6 (2024)
26. Wang, Y., Ye, W., Guo, P., He, Y., Wang, Z., Tian, B., He, S., Sun, G., Shen, Z., Chen, S., et al.: Symrtlo: Enhancing rtl code optimization with llms and neuron-inspired symbolic reasoning. In: The Thirty-ninth Annual Conference on Neural Information Processing Systems
27. Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/ (2013)
28. Yang, Y., Teng, F., Liu, P., Qi, M., Lv, C., Li, J., Zhang, X., He, Z.: Haven: Hallucination-mitigated llm for verilog code generation aligned with hdl engineers (2025), https://arxiv.org/abs/2501.04908
29. Yao, X., Wang, Y., Li, X., Lian, Y., Chen, R., Chen, L., Yuan, M., Xu, H., Yu, B.: 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)

30. Zhao, Y., Huang, D., Li, C., Jin, P., Song, M., Xu, Y., Nan, Z., Gao, M., Ma, T., Qi, L., Pan, Y., Zhang, Z., Zhang, R., Zhang, X., Du, Z., Guo, Q., Hu, X.: Codev: Empowering llms with hdl generation through multi-level summarization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems pp. 1–1 (2025). https://doi.org/10.1109/TCAD.2025.3604320
31. Zhu, Y., Huang, D., Lyu, H., Zhang, X., Li, C., Shi, W., Wu, Y., Mu, J., Wang, J., zhao, Y., Jin, P., Cheng, S., shengwen Liang, Zhang, X., Zhang, R., Du, Z., Guo, Q., Hu, X., Chen, Y.: Qimeng-codev-r1: Reasoning-enhanced verilog generation. In: The Thirty-ninth Annual Conference on Neural Information Processing Systems (2025), https://openreview.net/forum?id=ly5DnRIgCZ