Hardware Generation with High Flexibility using Reinforcement Learning Enhanced LLMs

Yifang Zhao*, Weimin Fu†, Shijie Li*, Yi-Xiang Hu*, Xiaolong Guo†✉, Yier Jin*
*University of Science and Technology of China, {zhaoyifang, shijie_li, yixianghu}@mail.ustc.edu.cn, jinyier@ustc.edu.cn
†Kansas State University, {weiminf, guoxiaolong}@ksu.edu

***Abstract*—The increasing complexity of integrated circuit design requires customizing Power, Performance, and Area (PPA) metrics according to different application demands. However, most engineers cannot anticipate requirements early in the design process, often discovering mismatches only after synthesis, necessitating iterative optimization or redesign. Some works have shown the promising capabilities of large language models (LLMs) in hardware design generation tasks, but they fail to tackle the PPA trade-off problem. In this work, we propose an LLM-based reinforcement learning framework, PPA-RTL, aiming to introduce LLMs as a cutting-edge automation tool by directly incorporating post-synthesis metrics PPA into the hardware design generation phase. We design PPA metrics as reward feedback to guide the model in producing designs aligned with specific optimization objectives across various scenarios. The experimental results demonstrate that PPA-RTL models, optimized for Power, Performance, Area, or their various combinations, significantly improve in achieving the desired trade-offs, making PPA-RTL applicable to a variety of application scenarios and project constraints.**

***Index Terms*—Hardware generation, PPA, Reinforcement learning.**

I. Introduction

With the broader application of Integrated circuits (ICs) in various scenarios, IC design requirements have become increasingly diverse [1]. Different applications demand tailored Power, Performance, and Area (PPA) metrics, often resulting in designs that fulfill similar functions but with vastly different optimization priorities. For instance, high-performance computing prioritizes performance over power efficiency, while IoT devices are susceptible to power consumption.

The diversity of design requirements presents challenges for hardware engineers. Hardware designs are described at the behavioral or Register Transfer Level (RTL), focusing on functional logic, while customized PPA optimization requires physical implementation with precise layout, routing, and cell library. Engineers begin with high-level hardware designs and then leverage Electronic Design Automation (EDA) tools for logic synthesis [2], verification [3], layout and routing [4], and timing analysis [5]. By defining appropriate constraints, fine-tuning tool settings, and adjusting outputs, they strive to meet the stringent power, performance, and area demands of modern ICs. However, these EDA tools are expertise-driven, raising the entry barriers for circuit design. Most engineers cannot anticipate requirements early in the design process, often discovering mismatches only after synthesis, necessitating iterative optimization or redesign. Thus, creating a PPA-oriented hardware design generation tool can greatly improve development efficiency.

With the advent of ChatGPT, its powerful natural language processing capabilities have brought new perspectives to many fields [6], making large language models (LLMs) a highly promising automation tool. In the field of hardware design generation, several studies [7]–[10] have shown that LLMs can autonomously produce hardware designs that are both syntactically and functionally accurate, offering convenience to hardware engineers. However, these models fail to consider the practical applicability of designs across different scenarios. To integrate post-synthesis metrics with the generation model, we aim to construct a hardware generation LLM with high flexibility.

LLM research technologies include prompting engineering, supervised fine-tuning (SFT), and Reinforcement Learning from Human Feedback (RLHF). Prompting methods rely on commercial general-purpose LLMs, struggling with the hallucination problem in general LLMs [11]. SFT typically enhances an LLM's performance by training the model to mimic behaviors on domain-specific datasets, but it falls short of meeting specific needs in specialized scenarios, such as generating hardware designs that align with PPA preferences tailored to specific application requirements. Moreover, SFT is highly data-intensive [12], requiring datasets of over $10,000$ examples to induce noticeable shifts in model behavior. Constructing such large-scale datasets for a specific scenario is nearly impossible, making it challenging to achieve tailored optimization using SFT alone. RLHF optimizes model behavior by incorporating human feedback and reinforcement learning (RL), which helps mitigate hallucinations [13] and reduces the need for large-scale datasets, typically requiring only a few thousand samples focused on quality rather than quantity.

In this work, we leverage RLHF-based LLM technology to generate hardware designs that better suit the diverse needs of real-world applications. We design PPA metrics as rewards, enabling the model to iteratively adjust its parameters based on feedback. Through continuous updates to its policy, the model progressively optimizes outputs to align with specific PPA preferences. However, this process faces two challenges: **(C1)** The complex trade-off between power, performance, and area typically demands sophisticated strategies and human intervention, while RL needs to formally represent these trade-offs in the reward feedback to meet different PPA objectives. **(C2)** Performing PPA calculations in every iteration would significantly slow down the model training process, as generating PPA evaluation feedback requires logic synthesis and simulation, which are computationally intensive steps that consume significant time and resources. One solution is to use machine learning (ML)–based models to approximate PPA metrics, which address real-time concerns but may lack precision. Another approach separates PPA calculations from model training by creating an offline PPA metric dataset, which balances computational demands with real-time requirements.

To overcome these challenges, we propose the PPA-RTL framework, which constructs an offline dataset of *Function Description*⇔*Reference Code* pairs with PPA metrics and introduces a PPA weighted formula to represent optimization rewards across diverse scenarios. Integrating PPA into the hardware generation process through RL allows models to more effectively navigate the complex trade-offs in hardware design, such as balancing chip performance and power consumption or optimizing the layout to reduce area without compromising functionality.

The main contributions are outlined as follows:

1) For the first time, we optimize key post-synthesis metrics, PPA, in

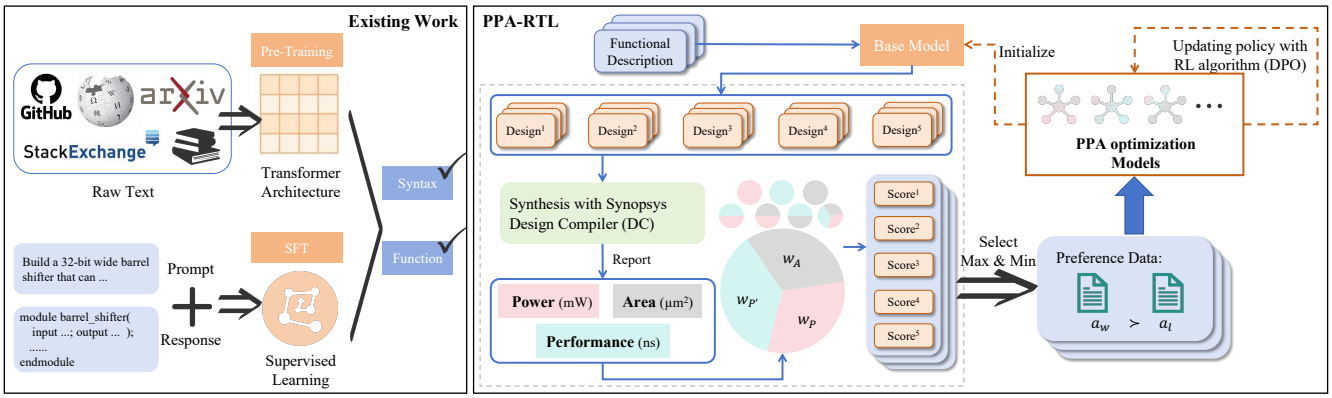

Fig. 1: PPA-RTL Framework. Existing hardware code generation LLMs primarily focus on two training methods (Pre-Training and SFT) to acquire syntactic and functional correct output. The PPA-RTL framework focuses on post-synthesis PPA metrics to enhance hardware generation quality. It first extracts PPA values from the Power, Performance, and Area synthesis report generated by RTL synthesis tools for each design. Adjusting the weights of PPA generates various preference datasets that adapt to different PPA preferences, thereby obtaining hardware code generation LLMs guided by different PPA preferences.

the context of hardware code generation using LLMs. We develop an RL-based hardware generation framework, named **PPA-RTL**, that directly integrates PPA considerations into the design process, allowing users to prioritize one or more of these metrics based on specific project requirements.

2) We configure various optimization objectives to reflect the common trade-offs in real-world hardware designs, such as Power-only, Performance-only, Area-only, and their combinations (e.g., Power & Performance, Power & Area, Performance & Area, and Power & Performance & Area). The experimental results demonstrate that PPA-RTL models significantly enhance the ability to achieve desired trade-offs, making PPA-RTL adaptable to a range of application scenarios and project constraints. Compared to the SFT-only model, the SFT-RL-PPA model improves Power by an average of 20.97%, Performance by 14.68%, and Area by 29.05%.

II. Background and Related Work

A. LLMs for Hardware Code Generation

As early as 2020, the hardware field began experimenting with fine-tuned GPT-2 models for generating hardware designs from natural language. DAVE [14] demonstrated the potential of language models in automated hardware design. However, DAVE has notable limitations: difficulties with complex, multi-task instructions, limited generalization to tasks outside its training templates, and a restriction on output length (1024 tokens).

After OpenAI released ChatGPT in 2022, it quickly attracted many researchers, sparking studies on the hardware domain, primarily focusing on two research methods: Prompt-based and SFT. Prompt-based methods [10], [15]–[17] guide advanced general-purpose LLMs to perform tasks within the hardware domain by providing domain-specific knowledge as prompts, relying on powerful commercial models like GPT-4 [6] and Claude [18]. However, the inability to modify these closed-source commercial models makes it difficult to address hallucination issues, particularly in the hardware domain [19]. SFT methods [8], [9], [20]–[22] train LLMs on hardware-specific datasets consisting of functional descriptions and corresponding hardware code to achieve higher syntactic and functional accuracy. However, generating codes is only the first step in the IC design process, which must also undergo logical synthesis, layout and routing, timing analysis, and physical verification. In these stages, the post-synthesis metrics PPA directly impact the chip's efficiency, speed, and cost. While SFT-based models can generate syntactically correct hardware designs, they do not account for post-synthesis metrics, limiting their practical applicability.

B. Reinforcement Learning from Human Feedback

In standard RLHF workflow, an LLM is modeled as a policy π, which takes a prompt $x \in \mathcal{X}$ and generates a response $a \in \mathcal{A}$ from the distribution $\pi(\cdot|x)$. The initial model, π_0, is typically fine-tuned on instruction-following data after pre-training [23]. The central idea of RLHF is to train the model using *relative feedback*, rather than relying on an absolute reward signal commonly used in traditional RL. It addresses the challenge that human evaluators often struggle to provide consistent absolute ratings but tend to be more reliable when tasked with comparing two responses and selecting the preferred one [24]. To formalize preference feedback, we introduce the concept of *Preference Oracle*.

Definition 1 (Preference Oracle). A preference oracle $\mathbb{P} : \mathcal{X} \times \mathcal{A} \times \mathcal{A} \to [0, 1]$ is a function that, when queried with a prompt x and two candidate responses a^1 and a^2, returns a preference signal. Specifically, we can sample a binary outcome y from the Bernoulli distribution $\text{Ber}(t)$: $y \sim \text{Ber}\left(\mathbb{P}\left(a^1 \succ a^2 | x, a^1, a^2\right)\right)$, if $y = 1$, it indicates that a^1 is preferred over a^2; if $y = 0$, it indicates that a^2 is preferred.

To simplify the learning process, it is often assumed that human preferences can be modeled using the Bradley-Terry (BT) model, a widely-used framework in preference learning [25].

Definition 2 (Bradley-Terry Model). Assume the existence of a ground-truth reward function r^*. Given two options a^1 and a^2, the preference probability is calculated using a sigmoid function, which reflects the difference in reward values between the two options. Specifically, the probability of preferring a^1 over a^2 given prompt x is modeled as: $\mathbb{P}(a^1 \succ a^2 | x, a^1, a^2) = \sigma(r^*(x, a^1) - r^*(x, a^2))$, where $\sigma(z) = \frac{1}{1+\exp(-z)}$ is the sigmoid function, which maps the difference in reward values to a preference probability.

Although the BT model may not fully capture the complex human preference, it provides a reasonable approximation and has proven to play a critical role in the development of ChatGPT [6] and Claude [18] by combining RLHF with reward maximization.

III. Methodology

We propose the **PPA-RTL** framework, which leverages RL to incorporate feedback from post-synthesis PPA metrics, enabling LLMs to generate hardware designs adaptable to various optimization objectives. PPA-RTL addresses the limitations of existing hardware code generation LLMs, which primarily focus on syntactic and functional correctness. It aims to improve the quality of hardware generation and ensure that the generated designs meet specific physical scenarios.

A. Motivation

LLMs are initially pre-trained on extensive universal datasets, equipping them with understanding ability. Then, the pre-trained LLM was fine-tuned on a specific task, which is IC design here, to become a domain expert. However, achieving optimal performance in real-world IC design involves navigating complex trade-offs among PPA. Decisions, such as prioritizing power efficiency over performance or balancing area constraints with power demands, require sophisticated decision-making, which labeled data cannot fully address. Therefore, SFT, which relies on labeled data, can not handle the flexible optimization to achieve PPA requirements. This leads to our first challenge: **(C1)** *How can we design a flexible framework that accommodates different PPA requirements?*

A straightforward approach for guiding LLMs in meeting diverse PPA requirements is to align models with preference via RL. Unfortunately, real-time PPA computing contradicts the overhead of LLM training, which raises our second challenge: **(C2)** *How can we avoid real-time evaluation to reduce the time delay during LLM training?*

B. PPA-RTL Framework

To construct PPA-oriented flexible hardware generation LLMs, we propose the PPA-RTL framework, which incorporates PPA metrics into the hardware code generation stage to target different application optimization objectives, as shown in Fig. 1. We address two challenges as follows.

Challenges 1 (C1) Different applications require tailored optimization strategies to balance power, performance, and area, while a single optimization strategy cannot meet the requirements of various applications. PPA-RTL needs to formally represent these trade-offs in the reward feedback to effectively guide model parameter updates.

To address **(C1)**, PPA-RTL formulates the hardware design generation as an RLHF workflow, denoted by $\langle \pi, \mathcal{X}, \mathcal{A}, \mathbb{P} \rangle$. **Policy** π represents a hardware design generation model, with the base model denoted as π_{ref}. **Prompt set** $\mathcal{X}$**:** Each prompt $x \in \mathcal{X}$ represents a functional description. **Response set** $\mathcal{A}$**:** The response $a \in \mathcal{A}$ is generated design from the distribution $\pi(\cdot|x)$. **Preference model** $\mathbb{P}\left(a^1 \succ a^2 | x, a^1, a^2\right)$ provides a relative preference signal for multiple generated designs under a specified PPA optimization strategy. In PPA-RTL, we sample k designs from the base model for each functional description x in the set $\mathcal{X}$. RTL synthesis tool then performs logic synthesis and simulation, resulting in reports about power, timing, and area. By specifying an optimization strategy, comparing PPA values, and selecting better & worse designs, the model's policy is updated using an RL algorithm to align outputs more closely with the desired PPA strategy.

To accommodate various applications, we implement a flexible approach that adjusts the weights $w_P, w_{P'}$, and w_A for each objective to simulate the priority of PPA metrics across different applications. The preference dataset is then created by selecting better or worse designs based on the scoring function $s(x, a)$.

$$s(x,a) = -\left(w_P \times v'_P + w_{P'} \times v'_{P'} + w_A \times v'_A\right)$$

where v'_P (resp. $v'_{P'}$, v'_A) represents the normalized value of power (resp. performance, area) extracted from the post-synthesis reports. The non-negative hyperparameters $w_P, w_{P'}$, and w_A can be adjusted to satisfy different PPA requirements. The negative sign indicates that the objective is to minimize these metrics: lower power consumption, higher frequency (lower critical path delay), and smaller area.

Challenge 2 (C2) Acquiring PPA values of design requires using time-consuming and computationally intensive EDA tools for synthesis, simulation, and analysis. Each update requires performing these steps, resulting in the inability to obtain real-time feedback during model training. To bridge hardware design synthesis and model training, approximate evaluation and pre-generated offline PPA datasets provide promising methods. For the approximate method, training an ML-based approximation model to predict PPA values can quickly provide PPA estimates, but it has limitations in accuracy and reliability. The cumulative error of the approximate model combined with the randomness in LLM generation creates a dual source of error, which may cause the model to gradually deviate from the optimal solution during training, leading to designs that do not meet expectations. For offline PPA datasets, using pre-generated data directly during training eliminates the intermediate steps of generating PPA values each time, simplifying the training process. The offline approach also allows for more accurate model optimization, as the dataset is already precisely calculated through EDA tools, avoiding the prediction errors associated with approximate models.

To address **(C2)**, we construct an offline dataset with PPA values for the training process by employing an EDA tool to map hardware designs into gate-level netlists. The synthesis process generates three key reports—power, timing, and area—which collectively capture the physical and performance characteristics of the design.

1) ***Power Estimation****:* Power estimation includes dynamic power from switching activity and static power dissipated by transistors without activity, providing an essential view into the design's energy efficiency, expressed in milliwatts (mW).

2) ***Performance Estimation****:* Performance is measured by the number of operations executed per unit of time. Increasing the frequency reduces the clock period, allowing the design to perform more operations within the same time frame, thereby improving overall system performance. The maximum frequency (f_{max}) is inversely related to the critical path delay, as expressed by the formula: $f_{\text{max}} = 1/critical\_path\_delay$. Therefore, performance can be quantified by the critical path delay (measured in nanoseconds, ns): the smaller the critical path delay, the higher the frequency, and thus, the better the overall performance.

3) ***Area Estimation****:* The total cell area is the physical area required by the synthesized design, including contributions from combinational logic, sequential elements, buffers/inverters, and other components. It reflects the cumulative area of all logic cells mapped from the technology library, measured in square micrometers (μm^2), ensuring the design fits within layout constraints.

C. PPA-RTL Training

In Algorithm 1, training the PPA-RTL model begins with the preference dataset generation, followed by the training phase. We leverages the Direct Preference Optimization (DPO) [26] to update the policy π_θ.

In the preference dataset generation stage (lines 2-13), for each functional description $x \in \mathcal{X}$, the reference model π_{ref} generates k candidate code samples. The samples that pass the synthesis check are collected in the candidate code set A. From set A, the best and worst samples, denoted as a_w and a_l respectively, are identified based

TABLE I: **Optimization objectives setting:** seven common trade-offs in real-world hardware design. **Preference dataset generation:** the valid subsets generated by $3,000$ function descriptions on the base model, excluding unsynthesizable and non-differentiated outputs. **Model training:** loss and accuracy metrics during model training based on different training strategies (RL-only and SFT-RL).

Optimization objectives	Weights			Preference dataset size ($\|\mathcal{D}\|$)	RL-only		SFT-RL	
	w_P	$w_{P'}$	w_A		Accuracy	Loss	Accuracy	Loss
Power-Only (Power)	1	0	0	1966	0.8375	0.5352	1.0	0.0322
Performance-Only (Perf)	0	1	0	1827	0.8375	0.5356	1.0	0.0540
Area-Only (Area)	0	0	1	1936	0.8250	0.6148	0.9875	0.0408
Power and Performance (PP)	1/2	1/2	0	1808	0.7625	0.5595	1.0	0.0610
Power and Area (PowerA)	1/2	0	1/2	1927	0.8625	0.5454	1.0	0.0410
Performance and Area (PerfA)	0	1/2	1/2	1813	0.7875	0.5750	1.0	0.0683
Power, Performance, Area (PPA)	1/3	1/3	1/3	1892	0.7875	0.5444	1.0	0.0360

Algorithm 1 PPA-RTL Training

Input: Functional descriptions $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, base model π_{ref}, hyperparameters w_P, $w_{P'}$,w_A, sample times k, training iterations T.

Output: Trained hardware generation model π_θ^*.

1: $\mathcal{D} \leftarrow \emptyset$; $\pi_\theta \leftarrow \pi_{ref}$ ▷ *Initialize*
2: **for** each $x \in \mathcal{X}$ **do** ▷ *Preference dataset generation*
3: $A \leftarrow \emptyset$ ▷ *Candidate code set*
4: **for** $i \leftarrow 1$ to k **do**
5: $a_i \leftarrow \pi_{ref}(x)$
6: **if** a_i passes synthesis check **then**
7: $A \leftarrow A \cup \{a_i\}$; Record $v_P(a_i)$,$v_{P'}(a_i)$,$v_A(a_i)$
8: Obtain $v'_P(a_i)$,$v'_{P'}(a_i)$,$v'_A(a_i)$ with min-max normalization
9: **for** $a_i \in A$ **do**
10: $s(x, a_i) \leftarrow -(w_P \times v'_P(a_i) + w_{P'} \times v'_{P'}(a_i) + w_A \times v'_A(a_i))$
11: $a_w \leftarrow \arg\max_{a_i \in A} s(x, a_i)$; $a_l \leftarrow \arg\min_{a_i \in A} s(x, a_i)$
12: **if** $a_w \neq a_l$ **then**
13: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, a_w, a_l)\}$
14: **for** $t \leftarrow 1$ to T **do** ▷ *Training*
15: **for** each $(x, a_w, a_l) \in D$ **do**
16: $\mathcal{L}_t(\pi_\theta) \leftarrow -\log \sigma \left(\beta \log \frac{\pi_\theta(a_w|x)}{\pi_{\text{ref}}(a_w|x)} - \beta \log \frac{\pi_\theta(a_l|x)}{\pi_{\text{ref}}(a_l|x)}\right)$
17: Update π_θ via gradient descent on $\mathcal{L}_t(\pi_\theta)$
18: $\pi_\theta^* \leftarrow \pi_\theta$
19: **return** π_θ^*

on the scoring function $s(x, a_i)$ and stored in the preference dataset $\mathcal{D}$ in the form of preference tuples (x, a_w, a_l).

In the training phase (lines 14-17), the model π_θ undergoes T iterations of optimization. The implicit reward function $r_\theta^*(x, a) = \beta \log \frac{\pi_\theta(a|x)}{\pi_{\text{ref}}(a|x)}$ represents the preference of the current model relative to the reference model for a given prompt-response pair. In each iteration, the loss function $\mathcal{L}_t(\pi_\theta)$ optimizes the model by maximizing the likelihood of preferred data a_w and minimizing the likelihood of non-preferred data a_l through the difference $r^*(x, a_w) - r^*(x, a_l)$ Gradient descent is used to update the model parameters iteratively. The resulting model π_θ^* can generate optimized hardware designs that align with learned preferences.

IV. Experiments

In this section, we validate the integration of RL with PPA metrics to enhance the capability of hardware code generation LLMs in meeting diverse real-world application physical demands.

A. Experimental Setup

PPA-RTL is implemented on a Linux machine with eight A100 80GB GPUs. We consider two code generation models as baselines, Deepseek-coder [27] and RTLCoder [21]. Deepseek-coder is an open-source code generation model closest to GPT-4-Turbo. RTLCoder is an open-source hardware code LLM based on Deepseek-coder with supervised fine-tuning. We use two versions of these models, Deepseek-coder-6.7b (Original) and RTLCoder-DeepSeek-6.7b (SFT-only), thereby exploring two distinct training strategies: 1) **RL-only**: Direct reinforcement training using the Deepseek-coder-6.7b model (Original). 2) **SFT-RL**: Reinforcement training based on supervised fine-tuning RTLCoder of the original model. Additionally, to demonstrate the applicability of our PPA-based RL approach to various PPA scenarios, Table I designs seven optimization objectives that reflect the common trade-offs of real-world hardware design.

B. Dataset Generation

To acquire PPA-oriented preference datasets, we leverage Synopsys Design Compiler (DC) [28] with the TSMC 90nm CMOS technology library to synthesize designs and adjust the weights of PPA based on different optimization objectives in Table I. Specifically, we sampled $3,000$ (*Function Description⇔Reference Code*) pairs from the RTLCoder dataset [29]. Function descriptions were inputs for two base models, Deepseek and RTLCoder, to generate multiple hardware designs. For each function description, we construct five hardware designs, including generated and reference code, and synthesize these designs. Samples that failed to generate synthesizable designs were filtered out. By analyzing the resulting Power, Timing, and Area reports, we extracted power (mW), critical path delay (ns), and area (μm^2). Based on different optimization objectives, calculate scores for each design, filter out samples with no score differences, and then select the maximum and minimum values to construct seven distinct chosen-rejected preference datasets.

C. Model Training

We employ DPO to train PPA-RTL hardware generation LLMs. DPO simplifies the traditional complexities of RLHF by leveraging the best and worst selected responses to directly optimize the model's predictive capabilities. Table I shows the training loss and reward accuracy for models trained with two strategies: RL-only and SFT-RL. $Loss$ quantifies the discrepancy between the model's predictions and the preferred outcomes, with lower values indicating closer alignment with preferences. $Accuracy$ shows the proportion of predictions that match the preferred outcomes, with higher values indicating better alignment with the target preferences. The SFT-RL exhibits lower loss and higher accuracy across all optimization objectives, indicating that SFT-based RL leads to more stable performance compared to RL-only.

TABLE II: PPA evaluation results. The **Original** refers to the DeepSeek-Coder-6.7B model, while the **SFT-only** is a supervised fine-tuned model based on DeepSeek-Coder-6.7B specifically for hardware code generation. **Metrics Power**, **Perf**, and **A** represent power (mW), performance (ns), and area (μm^2), respectively. Each row corresponds to a hardware design from the RTLLM benchmark. **Green** squares show better metric than the original model, while **red** squares show worse metric. The SFT-RL models are significantly better than RL-only models in PPA improvement compared to the base model.

Model	Orignal			vs Orignal																						
				GPT-4o			RL-Power			RL-Perf			RL-Area			RL-PP			RL-PowerA			RL-PerfA			RL-PPA		
Metric	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A
accu	0.055	1250.888	0.169	36.59%	-0.85%	46.81%	10.39%	-0.22%	7.04%	36.84%	11.14%	-0.06%	-0.38%	-0.82%	3.49%	2.59%	-0.17%	-0.20%	4.76%	5.21%	1.61%	0.88%	25.95%	-0.33%	4.13%	1.26%	44.78%
adder_16bit	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
adder_32bit	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
adder_8bit	0.002	16.934	0.240	-0.60%	-14.09%	-0.23%	-0.12%	-2.03%	-0.06%	-0.25%	0.09%	-0.09%	0.00%	-1.87%	0.00%	-0.12%	0.00%	-0.06%	-0.10%	-1.92%	-0.01%	-0.10%	-1.83%	-0.05%	3.17%	-1.87%	-0.05%
adder_pipe_64bit	0.141	2773.478	0.240	-12.80%	0.00%	-7.14%	0.00%	-2.51%	29.34%	-49.78%	5.21%	-100.00%	-2.05%	-3.80%	44.36%	21.76%	15.28%	68.85%	9.56%	-0.11%	7.78%	-14.23%	14.58%	55.83%	-2.27%	15.42%	42.66%
alu	0.672	6504.926	6.737	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	51.23%	96.29%	60.62%	30.18%	61.23%	24.40%	52.56%	85.57%	55.77%	N/A	N/A	N/A	51.23%	96.29%	60.62%
asyn_fifo	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%
calendar	0.015	496.742	0.244	-0.02%	-0.37%	-0.05%	10.73%	-0.16%	-0.03%	-0.01%	2.13%	-0.03%	-0.04%	-0.39%	12.03%	-0.01%	5.16%	-0.02%	3.27%	-0.39%	12.03%	-0.01%	15.62%	-0.03%	-0.02%	7.38%	15.27%
counter_12	0.004	98.078	0.265	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
div_16bit	0.164	1414.728	5.695	-3.29%	-100.00%	-2.03%	97.59%	95.39%	93.12%	97.59%	95.39%	93.12%	97.59%	95.39%	93.12%	97.59%	95.39%	93.12%	97.59%	95.39%	93.12%	-6.01%	13.20%	34.09%	97.59%	95.39%	93.12%
edge_detect	0.003	53.484	0.228	0.00%	-0.13%	0.00%	3.60%	-0.06%	-0.01%	0.00%	2.63%	-0.01%	0.00%	0.00%	1.58%	0.51%	1.75%	1.32%	2.98%	-0.13%	0.00%	-0.01%	10.53%	9.20%	-0.01%	5.26%	1.58%
freq_div	0.023	765.012	0.252	44.17%	0.79%	55.82%	79.03%	0.79%	87.59%	54.78%	0.26%	65.06%	72.75%	0.79%	82.14%	75.33%	0.79%	83.60%	66.73%	0.79%	77.07%	75.53%	0.79%	84.89%	74.80%	0.79%	82.86%
fsm	0.005	100.195	0.300	N/A	N/A	N/A	0.00%	20.00%	-0.01%	-0.02%	18.67%	-0.03%	-0.03%	20.00%	0.54%	42.61%	19.33%	-0.03%	-0.01%	18.00%	4.61%	-0.03%	18.89%	2.82%	53.65%	18.67%	2.92%
JC_counter	0.031	706.541	0.247	N/A	N/A	N/A	26.23%	-0.04%	-0.57%	-0.43%	38.95%	-0.57%	-0.42%	-0.04%	15.58%	25.35%	-0.04%	-0.57%	0.74%	-0.01%	-0.14%	-0.42%	-0.04%	1.45%	5.65%	15.09%	-0.57%
multi_16bit	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
multi_booth_8bit	0.070	1281.793	0.259	-2.33%	-0.22%	-0.99%	0.25%	-0.11%	-1.78%	0.25%	25.62%	2.87%	0.05%	8.29%	2.87%	12.37%	25.33%	22.13%	10.41%	12.87%	11.51%	-0.35%	12.96%	13.85%	5.01%	23.78%	3.78%
multi_pipe_4bit	0.055	887.645	0.261	16.11%	7.87%	-0.04%	-1.01%	-0.22%	-0.87%	N/A	N/A	N/A	N/A	N/A	N/A	24.64%	37.50%	15.09%	N/A	N/A	N/A	-0.92%	3.74%	-0.24%	5.48%	7.87%	18.43%
multi_pipe_8bit	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	0.00%	0.00%	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
parallel2serial	0.008	211.398	0.241	22.83%	0.50%	20.23%	6.10%	-0.11%	-0.46%	6.10%	0.50%	2.20%	16.43%	-0.02%	11.22%	24.20%	12.00%	-0.25%	39.63%	0.50%	39.84%	16.39%	0.50%	21.90%	16.43%	0.50%	11.22%
pe	2.878	11053.224	0.271	0.00%	0.00%	0.00%	0.38%	0.00%	0.00%	0.00%	0.00%	0.00%	0.38%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
pulse_detect	0.003	65.903	0.246	N/A	N/A	N/A	9.57%	0.00%	4.07%	-0.01%	18.70%	-0.01%	-20.51%	-0.04%	4.07%	6.37%	0.00%	-0.01%	3.23%	0.00%	3.54%	0.00%	0.81%	1.75%	4.18%	0.81%	6.82%
radix2_div	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RAM	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	N/A	N/A	N/A	0.00%	0.00%	0.00%
right_shifter	0.006	107.251	0.189	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
serial2parallel	0.013	356.469	0.223	20.39%	-0.36%	25.38%	22.52%	-0.22%	28.11%	14.72%	1.80%	18.33%	12.12%	-0.36%	15.12%	13.92%	-0.05%	17.22%	14.33%	-0.37%	18.29%	15.03%	-0.36%	18.61%	21.84%	-0.23%	27.87%
signal_generator	0.006	203.777	0.287	-0.01%	0.42%	-0.02%	4.79%	1.11%	4.09%	3.02%	1.11%	-0.02%	0.68%	1.67%	1.38%	4.05%	2.37%	-0.01%	1.03%	1.25%	1.58%	0.00%	1.53%	0.48%	2.67%	20.61%	1.18%
synchronizer	0.010	196.157	0.250	-0.05%	0.00%	-0.05%	0.00%	0.00%	0.00%	-0.03%	5.60%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.65%	-0.07%	21.71%	0.00%	0.00%	0.00%	0.00%	0.00%	0.79%
traffic_light	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
width_8to16	0.030	615.812	0.250	0.00%	0.00%	5.93%	11.40%	0.00%	8.56%	7.13%	1.33%	10.21%	11.11%	0.00%	8.79%	0.00%	0.00%	5.36%	27.69%	-0.04%	20.18%	11.40%	0.00%	8.56%	10.97%	0.00%	12.37%
Average Improvement Rates				5.26%	-4.63%	6.24%	10.82%	4.29%	9.93%	7.72%	10.42%	4.13%	8.85%	7.97%	13.22%	13.62%	9.85%	11.78%	12.74%	8.02%	13.65%	3.89%	4.68%	10.11%	12.66%	10.97%	15.20%

Model	SFT-only			vs Orignal			vs SFT-only																			
				SFT-only			SFT-RL-Power			SFT-RL-Perf			SFT-RL-Area			SFT-RL-PP			SFT-RL-PowerA			SFT-RL-PerfA			SFT-RL-PPA		
Metric	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A	Power	Perf	A
accu	0.035	738.058	0.250	36.18%	-0.85%	41.00%	65.83%	-0.07%	72.47%	-0.89%	21.40%	-2.34%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
adder_16bit	0.013	174.989	0.074	0.00%	0.00%	0.00%	31.37%	-4.59%	52.69%	-0.98%	-10.03%	-1.02%	-0.06%	-5.73%	30.11%	-0.73%	-6.59%	-0.79%	57.72%	-15.69%	54.78%	0.65%	33.78%	46.24%	30.44%	4.05%	66.26%
adder_32bit	0.083	550.368	4.761	0.00%	0.00%	0.00%	93.59%	98.36%	86.67%	N/A	N/A	N/A	N/A	N/A	N/A	56.71%	59.62%	18.65%	57.91%	58.47%	60.83%	72.47%	78.16%	63.26%	57.36%	62.63%	45.59%
adder_8bit	0.017	135.475	1.147	-0.51%	-9.59%	-0.23%	-0.08%	-4.44%	0.00%	-1.19%	65.12%	-0.19%	28.25%	83.65%	90.23%	79.85%	12.60%	-0.63%	47.75%	45.64%	46.09%	13.45%	27.79%	7.40%	22.27%	37.25%	17.50%
adder_pipe_64bit	0.831	20015.049	0.240	-25.20%	0.00%	-35.49%	74.96%	-100.00%	93.13%	-4.31%	45.83%	93.13%	97.50%	86.25%	98.78%	64.67%	-0.93%	92.74%	60.02%	-13.52%	95.28%	5.69%	0.00%	99.92%	51.05%	-1.91%	96.91%
alu	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
asyn_fifo	N/A	N/A	N/A	N/A	N/A	N/A	0.00%	0.00%	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	0.00%	0.00%	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
calendar	0.016	536.962	0.281	-0.04%	-0.39%	-0.08%	7.64%	13.52%	8.15%	0.28%	64.41%	0.39%	4.89%	10.32%	5.75%	7.42%	13.52%	7.84%	5.25%	9.11%	7.99%	24.59%	7.20%	28.69%	6.38%	13.17%	7.49%
counter_12	0.004	98.078	0.265	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	-0.40%	9.43%	0.00%	-0.40%	-0.10%	5.06%	13.81%	8.30%	-3.63%	68.29%	6.94%	4.88%	-0.02%	5.66%	1.08%	-0.25%	14.86%	10.79%
div_16bit	0.004	97.373	0.263	97.59%	95.39%	93.12%	0.00%	0.00%	0.00%	-0.40%	38.10%	-0.08%	-0.40%	-0.12%	4.40%	19.55%	7.43%	-3.64%	0.00%	0.00%	0.00%	-0.37%	7.43%	-3.64%	0.00%	0.00%	0.00%
edge_detect	0.003	53.626	0.240	0.00%	-0.13%	0.00%	2.75%	-0.21%	-0.40%	-0.06%	58.33%	-0.40%	23.68%	0.00%	35.53%	8.47%	-0.03%	-4.02%	-0.02%	-0.21%	11.66%	0.00%	0.00%	0.00%	-0.02%	-0.21%	-0.40%
freq_div	0.014	364.442	0.233	41.46%	7.41%	52.36%	70.82%	-0.28%	-1.23%	17.56%	57.14%	27.59%	32.24%	-0.10%	46.76%	53.99%	8.29%	-1.23%	50.07%	-0.27%	51.44%	-0.04%	1.43%	29.77%	53.13%	-0.16%	58.76%
fsm	0.004	80.791	0.248	10.17%	17.50%	19.37%	54.01%	-0.02%	68.56%	-0.02%	23.23%	17.03%	54.85%	3.03%	61.57%	0.00%	3.03%	-0.01%	-0.47%	-0.12%	-3.75%	-0.36%	1.82%	-3.78%	28.17%	3.03%	30.13%
JC_counter	0.043	995.602	0.251	-0.43%	-0.04%	-0.57%	98.27%	4.38%	98.44%	0.96%	0.27%	0.00%	0.27%	0.40%	0.07%	0.47%	0.30%	0.09%	0.28%	0.40%	0.07%	1.30%	0.20%	-0.01%	0.00%	0.00%	0.00%
multi_16bit	0.043	995.602	0.251	0.00%	0.00%	0.00%	N/A	N/A	N/A	96.19%	16.33%	98.30%	-0.20%	4.38%	96.88%	1.85%	2.05%	0.00%	0.28%	0.40%	0.07%	1.26%	0.00%	0.42%	0.28%	0.40%	0.07%
multi_booth_8bit	0.089	1446.762	0.269	-0.69%	-0.11%	-0.33%	74.39%	0.19%	66.96%	41.37%	13.20%	27.62%	30.27%	-0.17%	25.62%	37.39%	5.35%	30.16%	47.29%	3.62%	31.05%	37.39%	6.69%	30.16%	47.29%	3.62%	31.05%
multi_pipe_4bit	0.093	1438.718	0.281	-1.34%	-0.22%	-1.09%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
multi_pipe_8bit	0.093	1438.718	0.281	0.00%	0.00%	0.00%	N/A	N/A	N/A	46.64%	23.20%	33.08%	-0.20%	14.59%	97.84%	95.59%	14.59%	-1.29%	99.83%	75.09%	99.75%	40.58%	-0.16%	-1.29%	99.83%	75.09%	99.75%
parallel2serial	0.006	159.466	0.240	25.03%	0.50%	24.57%	1.04%	-0.05%	-0.06%	-1.49%	10.08%	-7.37%	-2.53%	-1.06%	45.72%	34.45%	0.00%	37.17%	83.01%	38.33%	86.55%	-0.15%	6.29%	37.17%	61.04%	38.33%	86.55%
pe	2.878	11053.224	0.271	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	-42.51%	14.76%	0.20%	-33.33%	0.00%	0.00%	0.00%	0.00%	0.00%	-100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	-41.67%	0.00%	0.00%
pulse_detect	0.003	66.326	0.245	7.52%	0.41%	0.00%	-0.01%	-0.05%	2.13%	-0.03%	0.68%	-0.04%	-0.05%	0.00%	-0.21%	14.21%	2.04%	28.99%	2.74%	-0.25%	19.75%	-0.05%	25.67%	-0.17%	16.05%	-0.55%	30.59%
radix2_div	0.035	622.574	0.186	0.00%	0.00%	0.00%	90.09%	-0.61%	-1.36%	-0.18%	2.16%	-1.44%	-5.43%	-0.51%	96.37%	-5.43%	-18.84%	24.63%	7.55%	-0.57%	28.74%	96.60%	3.24%	97.17%	-0.60%	-0.57%	28.74%
RAM	0.002	16.934	0.210	0.00%	0.00%	0.00%	-0.06%	-0.38%	-0.43%	-1.64%	0.00%	-8.57%	0.00%	-0.29%	0.00%	N/A	N/A	N/A	-0.07%	-0.29%	36.77%	-2.07%	4.76%	20.16%	2.42%	3.17%	8.87%
right_shifter	0.006	107.251	0.189	0.00%	0.00%	0.00%	43.46%	-0.58%	0.00%	-1.49%	39.50%	0.00%	0.00%	-0.12%	21.05%	79.80%	87.70%	0.00%	37.18%	-0.49%	15.43%	-1.10%	26.99%	0.00%	-0.39%	34.30%	15.43%
serial2parallel	0.013	317.661	0.238	0.07%	-0.16%	10.89%	13.53%	-0.12%	4.49%	-0.25%	26.22%	-0.65%	-0.28%	20.54%	66.24%	52.38%	0.00%	-0.64%	25.39%	-5.14%	4.34%	-0.11%	1.20%	8.78%	8.92%	36.86%	70.01%
signal_generator	0.006	201.096	0.284	6.33%	1.11%	1.32%	-0.19%	11.97%	-0.39%	-0.01%	2.11%	-0.04%	-0.01%	33.54%	46.67%	-0.01%	2.25%	6.18%	11.07%	-6.01%	1.94%	-1.08%	27.78%	6.18%	80.72%	36.97%	3.13%
synchronizer	0.010	196.157	0.250	0.00%	0.00%	0.00%	5.18%	0.00%	-0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	45.32%	5.18%	0.00%	6.33%	0.00%	0.00%	0.00%	12.13%	7.20%	14.32%	0.00%	0.00%	0.00%
traffic_light	0.010	196.157	0.250	0.00%	0.00%	0.00%	-0.04%	0.00%	0.00%	-0.56%	2.60%	-3.76%	20.98%	4.00%	26.98%	80.55%	2.81%	2.91%	0.00%	0.00%	0.00%	0.00%	0.00%	7.91%	0.00%	0.00%	0.00%
width_8to16	0.030	579.062	0.250	1.01%	0.00%	5.97%	88.01%	1.33%	89.76%	11.30%	0.71%	26.94%	45.41%	0.44%	33.59%	35.75%	2.81%	53.98%	61.49%	0.00%	66.13%	15.91%	1.78%	35.35%	1.78%	6.67%	19.05%
Average Improvement Rates				7.30%	4.11%	7.81%	32.58%	0.73%	25.58%	6.31%	20.99%	11.94%	12.31%	10.54%	40.85%	29.44%	8.25%	11.75%	24.90%	7.82%	28.79%	12.67%	11.00%	21.00%	20.97%	14.68%	29.05%

D. Experimental Evaluation

To evaluate the effectiveness of PPA-RTL models in improving various optimization objectives, we utilize the RTLLM-v1.1 benchmark [30] to generate designs for each example using the advanced general-purpose language model GPT-4o [31] and 14 models trained through RL-only and SFT-RL methods. These generated designs were synthesized with DC and evaluated based on power consumption, performance, and area. Table II shows evaluation results.

- **Lack of Sensitivity to PPA in Existing Models:** Compared to the original model, the current state-of-the-art model GPT-4o lacks effective optimization for these critical post-synthesis metrics, and the existing supervised fine-tuning hardware generation model SFT-only has limitations in achieving comprehensive performance improvements.
- **Trade-offs Between Metrics:** Optimizing a single metric can negatively impact others. Focusing on power metric, RL-only and SFT-RL models optimized for power (Power, PP, PowerA, and PPA) show significant improvements compared to models that prioritize only area or performance (Area, Perf, and PerfA) at the expense of power.
- **Balanced Optimization Across Metrics:** By comprehensively considering all metrics, the (SFT-)RL-PPA models demonstrate superior performance in both training strategies. The SFT-RL-PPA model improves Power by 20.97%, Performance by 14.68%, and Area by 29.05%, on average compared to the SFT-only model.
- **Design Quality Improvements with RL-based Training:** RL-based training using PPA metrics effectively improves hardware code generation quality. Moreover, the SFT-RL models show more

TABLE III: The PPA values for the calendar task based on SFT-only model and SFT-RL-PPA model. Performance is represented by the critical path delay.

Model	Power (mW)	Performance (ns)	Area (μm^2)
SFT-only	1.5884	0.29	536.9616
SFT-RL-PPA	1.4871	0.26	496.0368

significant performance improvements than RL-only models, with more green cells across a broader range of objectives.

Fig. 2 provides a detailed comparison of the Area and Power changes between the RL-Power model and its original model, Deepseek, for the designs generated on the RTLLM benchmark. The RL-Power model reduces power consumption without significantly increasing area, showcasing its ability to balance power and area trade-offs effectively.

E. Case Study

In this case study, we provide a detailed analysis of the designs generated by the PPA-RTL models before and after optimization, using the design of a `calendar` module as an example. The `calendar` module has two inputs: `CLK` (clock) and `RST` (reset). It includes three variables: `Secs`, `Mins`, and `Hours`, all initialized to zero. Each clock cycle, `Secs` increments by one. When `Secs == 59`, `Mins` increments, and `Secs` resets. When both `Secs == 59` and `Mins == 59`, `Hours` increments, and both `Secs` and `Mins` reset. This cycle continues until `Hours == 23`, `Mins == 59`, and `Secs == 59`, after which all values reset.

Table III compares the PPA values of the `calendar` designs generated by the SFT-RL-PPA model and its base model SFT-only model. Through RL targeted at Power & Performance & Area, the SFT-RL-PPA model effectively balances power, performance, and area requirements. To further analyze model differences, Table IV provides the designs generated by both models.

TABLE IV: The calendar module designs generated by SFT-only model and SFT-RL-PPA model.

SFT-only model

```
always @(posedge CLK or posedge RST)
      begin
   if (RST) begin Secs <= 6'b0; end
   else if (Secs == 6'd59) begin Secs
         <= 6'b0;end
   else begin Secs <= Secs + 1; end
end
always @(posedge CLK or posedge RST)
      begin
   if (RST) begin Mins <= 6'b0; end
   else if (Mins == 6'd59 && Secs ==
         6'd59) begin Mins <= 6'b0;
         end
   else if (Secs == 6'd59) begin Mins
         <= Mins + 1; end
end
always @(posedge CLK or posedge RST)
      begin
   if (RST) begin Hours <= 6'b0; end
   else if (Hours == 6'd23 && Mins ==
         6'd59 && Secs == 6'd59)
         begin Hours <= 6'b0; end
   else if (Mins == 6'd59 && Secs ==
         6'd59) begin Hours <= Hours
         + 1; end
end
```

SFT-RL-PPA model

```
always @(posedge CLK or posedge RST)
      begin
   if (RST) begin Secs <= 0; Mins <=
         0; Hours <= 0; end
   else begin
      if (Secs == 59) begin
      Secs <= 0;
         if (Mins == 59) begin
            Mins <= 0;
            if (Hours == 23) begin
                  Hours <= 0; end
            else begin Hours <= Hours
                  + 1; end end
         else begin Mins <= Mins + 1;
               end
      end else begin Secs <= Secs +
            1; end
   end
end
```

The SFT-only model uses three separate `always` blocks to manage updates for seconds, minutes, and hours, respectively. The processing logic for different time units is separated, enabling the logic clearer and the code easier to manage and expand. However, since the time units are scattered in different `always` blocks, potential race conditions may arise due to execution order. For example, updating `Secs` to zero might occur slightly before updating `Mins`. This brief delay could result in the minute and hour logic not detecting `Secs` as 59 in a given instant, preventing the intended increment. During logical synthesis, we observe this change that `Hours[0]` and `Mins[0]` are at $0.29ns$ on critical path delay, while `Secs[0]` is at $0.28ns$. The SFT-RL-PPA model updates of `Secs`, `Mins`, and `Hours` are completed in the same logical condition, and their updates are synchronized with each other. When `Secs` reaches 59 and resets to 0, `Mins` and `Hours` are updated within the same clock cycle. Concentrating the update logic within a single block reduces potential jitter caused by signal propagation between different `always` blocks, ensuring the atomicity of the update operations and avoiding potential race conditions.

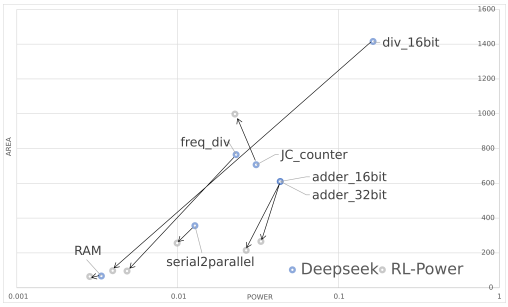

Fig. 2: The comparison of power and area between **Deepseek** (original model) and **RL-Power** (RL model based on Deepseek, optimized for Power) in RTLLM benchmark. The **x-axis** represents power consumption, and the **y-axis** represents area. Arrows indicate the performance shift from the original model (blue) to the RL-Power optimized model (gray) for the same design.

TABLE V: The Syntax and Function accuracy on the RTLLM-v1.1 (pass@5) between the SFT-only model and SFT-RL models.

		SFT-only	SFT-RL					
			Power	Perf	Area	PP	PowerA	PerfA	PPA
RTLLM-v1.1	Syn.(%)	93.1	**93.1**	**96.6**	89.6	**93.1**	86.2	**96.6**	**93.1**
pass@5	Func.(%)	48.3	**48.3**	44.8	41.4	44.8	44.8	**48.3**	41.4

V. Discussion

Although the PPA-RTL framework effectively achieves optimization goals tailored to different application scenarios, it is essential to evaluate whether integrating PPA into the hardware generation model affects the original model's code generation accuracy. We assess the syntactic and functional accuracy of the SFT-RL-based models on the RTLLM benchmark. Table V compares the SFT-RL models with its base model SFT-only. For syntactic accuracy, the SFT-RL models maintain the same or higher accuracy across five optimization objectives, as syntactically incorrect designs were filtered when selecting preference data based on PPA values, which benefits syntactic correctness. Two models retain their original accuracy levels for functional accuracy, while the other five models show a slight decrease of less than 7%. Overall, PPA-RTL can maintain the original model's accuracy within a limited error.

VI. Conclusion

In this paper, we propose PPA-RTL, a novel approach to align hardware code generation LLMs with variable power, performance, and area requirements. In experimental evaluation, PPA-RTL demonstrates significant improvements compared to traditional LLM-based hardware design, providing a flexible framework for balancing performance, efficiency, and resource usage. Our work paves the way for more adaptive, efficient, cost-effective IC design automated processes.

Acknowledgment

Portions of this work were supported by the National Science Foundation (2340949, 2419880, and the EPSCoR ARISE 2148878 REI Award).

References

[1] J. Kilby, "Invention of the integrated circuit," *IEEE Transactions on Electron Devices*, vol. 23, no. 7, pp. 648–654, 1976.

[2] L. G. Amarù, P. Vuillod *et al.*, "Logic optimization and synthesis: Trends and directions in industry," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, D. Atienza and G. D. Natale, Eds. IEEE, 2017, pp. 1303–1305. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927194

[3] J. Hu, T. Li, and S. Li, "Formal equivalence checking between SLM and RTL descriptions," in *28th IEEE International System-on-Chip Conference, SOCC 2015, Beijing, China, September 8-11, 2015*. IEEE, 2015, pp. 131–136. [Online]. Available: https://doi.org/10.1109/SOCC.2015.7406927

[4] P. Wei and B. Murmann, "Analog and mixed-signal layout automation using digital place-and-route tools," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 29, no. 11, pp. 1838–1849, 2021. [Online]. Available: https://doi.org/10.1109/TVLSI.2021.3105028

[5] G. J. Milne, "The formal description and verification of hardware timing," *IEEE Trans. Comput.*, vol. 40, no. 7, p. 811–826, Jul. 1991. [Online]. Available: https://doi.org/10.1109/12.83619

[6] OpenAI, "Gpt-4 Technical Report," ArXiv, Tech. Rep. abs/2303.08774, 2023.

[7] W. Fu, S. Li *et al.*, "Hardware phi-1.5b: A large language model encodes hardware domain specific knowledge," in *Proceedings of the 29th Asia and South Pacific Design Automation Conference, ASPDAC 2024, Incheon, Korea, January 22-25, 2024*. IEEE, 2024, pp. 349–354. [Online]. Available: https://doi.org/10.1109/ASP-DAC58780.2024.10473927

[8] M. Liu, N. R. Pinckney *et al.*, "Invited paper: Verilogeval: Evaluating large language models for verilog code generation," in *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*. IEEE, 2023, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ICCAD57390.2023.10323812

[9] Z. Pei, H. Zhen *et al.*, "Betterv: Controlled verilog generation with discriminative guidance," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=jKnW7r7de1

[10] S. Thakur, J. Blocklove *et al.*, "Autochip: Automating HDL generation using LLM feedback," *CoRR*, vol. abs/2311.04887, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.04887

[11] J. Schulman, "Reinforcement learning from human feedback: Progress and challenges," in *Berkeley EECS Colloquium. YouTube www. youtube. com/watch*, 2023.

[12] H. Mehrafarin, S. Rajaee, and M. T. Pilehvar, "On the importance of data size in probing fine-tuned models," in *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 228–238. [Online]. Available: https://doi.org/10.18653/v1/2022.findings-acl.20

[13] J. Li, J. Chen *et al.*, "The dawn after the dark: An empirical study on factuality hallucination in large language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 10 879–10 899. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.586

[14] H. Pearce, B. Tan, and R. Karri, "DAVE: deriving automatically verilog from english," in *MLCAD '20: 2020 ACM/IEEE Workshop on Machine Learning for CAD, Virtual Event, Iceland, November 16-20, 2020*, U. Schlichtmann, R. Gal *et al.*, Eds. ACM, 2020, pp. 27–32. [Online]. Available: https://doi.org/10.1145/3380446.3430634

[15] J. Blocklove, S. Garg *et al.*, "Chip-chat: Challenges and opportunities in conversational hardware design," in *5th ACM/IEEE Workshop on Machine Learning for CAD, MLCAD 2023, Snowbird, UT, USA, September 10-13, 2023*. IEEE, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.1109/MLCAD58807.2023.10299874

[16] K. Chang, Y. Wang *et al.*, "Chipgpt: How far are we from natural language hardware design," *CoRR*, vol. abs/2305.14019, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.14019

[17] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "Generating secure hardware using chatgpt resistant to cwes," *IACR Cryptol. ePrint Arch.*, p. 212, 2023. [Online]. Available: https://eprint.iacr.org/2023/212

[18] Anthropic. (2023) Meet claude. [Online]. Available: https://www.anthropic.com/claude

[19] W. Fu, S. Li *et al.*, "A generalize hardware debugging approach for large language models semi-syntectic datasets," *IEEE Trans. Circuits Syst. I Regul. Pap.*, 2024. [Online]. Available: https://doi.org/10.1109/TCSI.2024.3487486

[20] W. Fu, K. Yang *et al.*, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2023, Tianjin, China, December 13-15, 2023*. IEEE, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.1109/AsianHOST59942.2023.10409307

[21] S. Liu, W. Fang *et al.*, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," 2024.

[22] M. Liu, T.-D. Ene *et al.*, "Chipnemo: Domain-adapted llms for chip design," 2024.

[23] H. Dong, W. Xiong *et al.*, "RLHF workflow: From reward modeling to online RLHF," *CoRR*, vol. abs/2405.07863, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.07863

[24] P. F. Christiano, J. Leike *et al.*, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg *et al.*, Eds., 2017, pp. 4299–4307. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html

[25] L. Ouyang, J. Wu *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed *et al.*, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[26] R. Rafailov, A. Sharma *et al.*, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann *et al.*, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/a85b405ed65c6477a4fe8302b5e06ce7-Abstract-Conference.html

[27] D. Guo, Q. Zhu *et al.*, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[28] S. D. Compiler, "Synopsys design compiler," *Pages/default. aspx*, 2016.

[29] "Rtlcoder dataset." [Online]. Available: https://github.com/hkust-zhiyao/RTL-Coder

[30] Y. Lu, S. Liu *et al.*, "Rtllm: An open-source benchmark for design rtl generation with large language model," 2023.

[31] OpenAI, "Hello gpt-4o," 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/