Poster: Enhance Hardware Domain Specific Large Language Model with Reinforcement Learning for Resilience

Weimin Fu
weiminf@ksu.edu
Kansas State University
Manhttan, Kansas, USA

Yifang Zhao
zhaoyifang@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, PRC

Yier Jin
jinyier@ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, PRC

Xiaolong Guo
guoxiaolong@ksu.edu
Kansas State University
Manhttan, Kansas, USA

Abstract

To enhance the performance of large language models (LLMs) on hardware design tasks, we focus on training with reinforcement learning(RL) to improve LLMs' syntax synthesis and functional verification performance. We observed significant gains in power, performance, and area (PPA) metrics by applying RL. Specifically, DeepSeek Code saw a 23.6% performance increase, while the RTL-Coder improved by 7.86%. Our findings demonstrate the effectiveness of RL in refining LLMs for more accurate hardware generation, considering power and area consumption. This approach offers a promising direction for generating hardware resilient to side-channel attacks in computer systems.

CCS Concepts

• **Hardware → Emerging tools and methodologies**; **Software tools for EDA**.

Keywords

Large language model; EDA Tools; Hardware Security

ACM Reference Format:
Weimin Fu, Yifang Zhao, Yier Jin, and Xiaolong Guo. 2024. Poster: Enhance Hardware Domain Specific Large Language Model with Reinforcement Learning for Resilience. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3658644.3691384

1 Introduction

The hardware design abilities of large language models (**LLMs**) are rapidly improving as measured by design RTL generation benchmarks VerilogEval [6] and RTLLM [8]. Simultaneously, the direction of designing domain-specific LLMs in the hardware field has mainly divided into two approaches: pairing general LLMs with domain-specific prompts, such as in Chip-Chat [2], and fine-tuning LLMs using domain-specific datasets, as RTLCoder [7] and VeriGen [13]. These efforts have introduced LLMs into the hardware domain. However, their ultimate goal only involves achieving circuit synthesis. Current results show that LLMs perform reasonably well in syntax synthesis but struggle with functional verification. For instance, GPT-4 [9] achieves perfect accuracy in syntax synthesis (RTLLM), but in the VerilogEval Benchmark's Eval-Human, its *pass*@1, 5, and 10 scores are only 43.5, 55.8, and 59.9 (out of 100), respectively, even though it performs best among all LLMs. The best-performing open-source model, RTLCoder-DeepSeek, only achieves 41.5, 50.1, and 53.4. These results indicate an ordinary ceiling for both approaches: the limitation of transmitting hardware domain knowledge purely through semantic understanding.

The contradiction stems from the hardware design source code being structural while LLMs operate linearly. Textual similarity cannot guarantee critical factors, such as parenthesis closure, which are crucial for compilation. These elements are often treated as isolated semantic tokens, making it difficult for training. Additionally, many lower-level hardware characteristics cannot be adequately captured through content or "semantic" representation. For example, Power, Performance, and Area (PPA) metrics are often influenced by security measures in hardware, which can intentionally degrade PPA to prevent failures under extreme conditions. Moreover, generating hardware designs with specific security features, such as mitigating power and electromagnetic (EM) side-channel threats, cannot be achieved by simply encouraging the generation of specific content. Instead, it requires penalizing the generation of undesirable patterns or situations.

Previous reinforcement learning (RL) works, such as AlphaGo [12] and OpenAI Dota 2 [1], have demonstrated that RL techniques can train neural networks for complex planning in game environments. Notably, CoderRL [4] enables software code generation optimization based on compiler feedback. Given these prior successes and the inherently interactive nature of problem-solving, applying RL to LLM hardware generation seems a natural next step. In this proposal, we explore leveraging RL concepts to enhance LLMs' capabilities in hardware design across various reward schemes.

Contributions. In summary, the contributions of this proposal are as follows: 1. Demonstrate that RL can enhance the synthesis pass rate of LLMs' generated hardware designs. 2. Proof RL can train LLMs with hardware synthesis metrics and specifications, ensuring the generated designs possess specific characteristics. 3. This work advances the development of LLMs in the hardware domain, assisting practitioners in creating hardware designs that meet specific specifications or rules. 4. Propose the first cross-layer hardware LLM framework to define PPA metrics at the RT-level and use them to enhance resilience against side-channel attacks.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3691384

2 Background

2.1 From Markov Decision Process to Reinforcement Learning

Consider a Markov Decision Process consisting of states $\in \mathcal{S}$, action $\in \mathcal{A}$, rewards $r \in \mathcal{R}$, a discount factor γ, and a transition probability function $p(s_{t+1}|s_t, a_t)$, where t is an integer denoting the timestep and $(\mathcal{S}, \mathcal{A})$ are the state and action spaces. In environments described by a Markov Decision Process, at each timestep t, the agent observes the state s_t, selects an action $a_t \sim \pi(\cdot|s_t)$ from its policy, and then observes the next state $s_{t+1} \sim p(\cdot|s_t, a_t)$ sampled from the environment's transition dynamics. RL algorithms aim to maximize the return, defined as the cumulative sum of rewards $\sum_t \gamma^t r_t$, throughout the training episode. Most RL algorithms maximize returns through trial and error by directly interacting with the environment. However, offline RL [5] has recently emerged as an alternative paradigm, where an agent aims to extract return-maximizing policies from offline data gathered by another agent. The offline dataset consists of (s, a, r) tuples.

2.2 Transfomer, LLM, and RL from Human Feedback

Transformers [14] is the basis of LLM, which are typically pretrained with predicting the next token [10]. However, training LLMs with RL presents significant challenges, as many tasks involve complex, poorly defined, or hard-to-specify goals, leading us to one offline RL: Reinforcement Learning from Human Feedback (RLHF). RLHF [3] typically involves training a reward model r to capture human preferences over a task τ. The reward model is then used to score LLM responses to prompts from the task, followed by policy improvement, often using Direct Preference Optimization(DPO) [11].

3 Methodology and Experiment

Our preliminary experiments employed RLHF and DPO by constructing a dataset comprising hardware implementations and performance scores. The choice of RLHF was driven by the need to validate the conceptual framework under constrained cost conditions. Our ongoing work involves transitioning to RL without Human Feedback, which enables the creation of more complex scenarios, such as CWE and power side-channel.

3.1 PPA Assistant Dataset Construction

The dataset construction process began with aggregating open-source datasets from RTLCoder and VeriGen. These datasets include multiple implementations of the same hardware functionality with identical input/output. We categorized these hardware designs and selected 1000 designs with diverse IO and functionalities.

We evaluated the PPA metrics for each category to identify the optimal hardware implementation as the ideal scenario. Other implementations within the same category served as comparison subjects. The PPA metrics were estimated using Xilinx ISE Vivado v2019.1 (64-bit), targeting the AMD Kintex7 FPGA KC705 Evaluation Kit for synthesis. Specifically, Performance was measured by the total timing check issues reported in the Check Time report; Power consumption was assessed using the Total On-Chip Power (W); Area utilization was determined based on the resource usage reported in the Utilization report (including LUTs, FFs, IOs).

A new PPA metric will also be developed to determine the best hardware implementations for defending against power and EM side-channel attacks. We will collect EM and power traces simulated using EMsim+ and Synopsys PrimeTime PTPX. A Python script will then be developed to perform correlation analysis on these traces. Finally, the number of traces needed to infer sensitive information successfully will be recorded and used to design this new metric.

3.2 Code Similarity Calculation: Integrating Abstract Syntax Tree and Data Flow Graph

We identified the lack of structural comprehension as a primary cause of synthesis failures. To enhance the LLM's understanding of structure, we developed a reward function comprising text similarity, Abstract Syntax Tree (AST) similarity, Data Flow Graph (DFG) similarity, and PPA metrics. Text similarity was computed using cosine similarity. AST and DFG similarities were estimated using graph similarity. The reward was calculated as the average of the similarities.

3.3 LLM Training

We utilized the open-source code LLMs Codellama and DeepSeek and the hardware fine-tuned variant RTLCoder. In Fig. 1, we compare the performance of the reinforcement learning fine-tuned models against their original versions on the RTLLM benchmark. Additionally, we included performance metrics of three closed-source LLMs from OpenAI for workload reference.

It is important to note that a direct performance comparison between open-source and closed-source models is not meaningful, as all the selected open-source models have a parameter size of 7B, significantly smaller than ChatGPT. Larger models inherently possess advantages in semantic understanding and complex generation tasks. Therefore, we use these comparisons to illustrate differences in workload duration and total effort.

Given that our reinforcement learning approach primarily aimed to enhance the models' understanding of hardware design structures and syntax, we focused on these aspects in our evaluation. Before training, Codellama's performance was comparable to the GPT-3.5 Turbo(Nov 6, 2023). After training, Codellama surpassed the GPT-4(Jan 15, 2024) in handling hardware-specific tasks. DeepSeek Coder initially performed close to GPT-4, and after RL, its performance matched the GPT-4o(May 13, 2024), achieving 100% accuracy.

For the RTLCoder, the syntax correctness rate remained unchanged before and after training. However, despite not explicitly targeting semantic understanding in our design, reinforcement learning improved functional accuracy (7% increase).

3.4 PPA Comparison

Table 1 compares the hardware generation performance of DeepSeek Code and RTLCoder on the RTLLM benchmark before and after reinforcement training. The results demonstrate that reinforcement learning consistently improves Performance metrics while inducing noticeable changes in Power and Area metrics. Specifically, for DeepSeek Code, reinforcement learning resulted in an average Performance increase of 23.6%, a Power improvement of 12.02%,

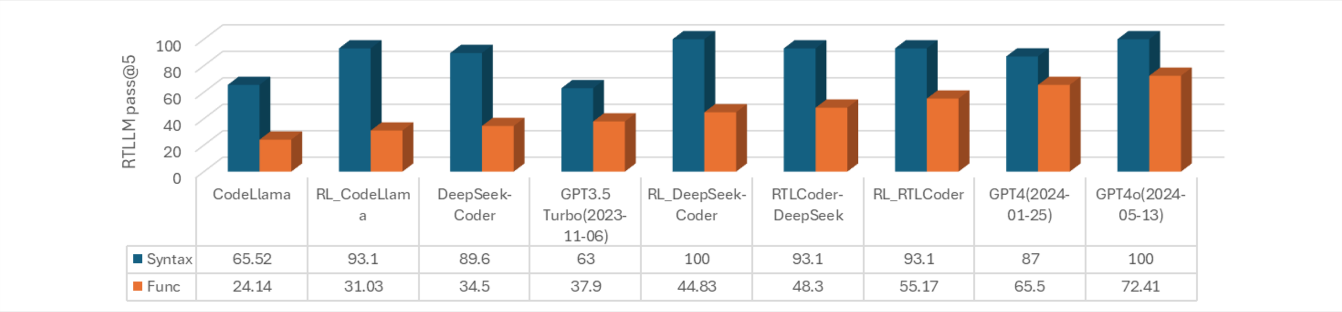

Figure 1: Performance of Original LLMs and the RF finetuned version on RTLLM Benchmark

Table 1: Comparison of PPA for Hardware Designs Generated by LLMs on the RTLLM Benchmark Before and After RL

Design	Deepseek-Coder-6.7B									RTLCoder-DeepSeek								
	Perf			Power			Area			Perf			Power			Area		
	Orignal	RL Ver.	Improve	Orignal	RL Ver.	Improve	Original	RL Ver.	Improve	Orignal	RL Ver.	Improve	Orignal	RL Ver.	Improve	Original	RL Ver.	Improve
Design 1-6			0.00%			0.00%			0.00%			0.00%			0.00%			0.00%
RAM	309	231	25.24%	2.063	2.063	0.00%	4.42	4.38	0.90%	1581	212	86.59%	3.762	3.762	0.00%	5.76	5.75	0.17%
width_8to16	126	94	25.40%	1.558	1.866	-19.77%	5.61	5.6	0.18%	130	126	3.08%	1.161	2.24	-92.94%	5.61	5.6	0.18%
adder_8bit	9	9	0.00%	5.753	5.751	0.03%	5.22	5.2	0.38%	9	9	0.00%	5.753	5.752	0.02%	5.22	5.2	0.38%
adder_16bit	17	17	0.00%	11.269	11.139	1.15%	10.01	10	0.10%	17	16	5.88%	12.324	10.174	17.45%	10.01	10.01	0.00%
radix2_div	N/A	122	100.00%	N/A	0.652	N/A	N/A	7.42	N/A	167	94	43.71%	0.187	0.433	-131.55%	3.4	4.1	-20.59%
trafic_light	61	33	45.90%	3.003	4.535	-51.02%	2.81	2.6	7.47%	33	33	0.00%	4.2	4.196	0.10%	2.81	2.6	7.47%
freq_div	21	21	0.00%	2.358	2.358	0.00%	1	1	0.00%	38	38	0.00%	2.82	2.82	0.00%	1	1	0.00%
accu	198	129	34.85%	2.579	3.745	-45.21%	4.42	4.42	0.00%	121	159	-31.40%	1.264	3.691	-192.01%	4.41	4.42	-0.23%
fsm	15.5	15	3.23%	0.541	0.247	54.34%	0.8	0.8	0.00%	15	16	-6.67%	0.308	0.265	13.96%	0.8	0.8	0.00%
JC_Counter	443	443	0.00%	129.119	129.119	0.00%	13.22	13.22	0.00%	443	443	0.00%	129.119	129.119	0.00%	13.22	13.22	0.00%
multi16bit	336	336	0.00%	9.76	4.605	52.82%	10.16	8.63	15.06%	347	337	2.88%	5.669	1.002	82.32%	13.66	13.66	0.00%
multibooth8bit	301	257	14.62%	0.563	0.577	-2.49%	7.05	7.04	0.14%	400	400	0.00%	2.974	2.974	0.00%	7.05	7.05	0.00%
multipipe4bit	157	113	28.03%	3.932	8.321	-111.62%	3.62	3.62	0.00%	89	67	24.72%	5.046	1.691	66.49%	3.61	2.41	33.24%
multipipe8bit	458	167	63.54%	12.729	16.533	-29.88%	7.25	7.24	0.14%	577	N/A	#VALUE!	12.133	N/A	N/A	7.25	N/A	N/A
parallel2serial	43	43	0.00%	4.293	4.293	0.00%	2.2	2.2	0.00%	31	28	9.68%	2.255	2.249	0.27%	1.6	1.6	0.00%
pulse_detect	14	12	14.29%	0.713	0.416	41.65%	0.8	0.8	0.00%	14	15	-7.14%	0.258	0.296	-14.73%	0.8	0.8	0.00%
serial2parallel	88	88	0.00%	2.264	2.499	-10.38%	2.6	2.6	0.00%	88	64	27.27%	2.499	0.569	77.23%	2.6	2.6	0.00%
signal_generator	29	24	17.24%	3.942	4.065	-3.12%	1.41	1.4	0.71%	24	24	0.00%	4.069	4.069	0.00%	1.4	1.4	0.00%
alu	370	356	3.78%	16.397	12.173	25.76%	21.68	21.6	0.37%	370	363	1.89%	12.529	12.441	0.70%	21.61	21.56	0.23%
div_16bit	134	16	88.06%	14.293	4.167	70.85%	11.36	9.6	15.49%	N/A	32	N/A	N/A	33.139	N/A	N/A	11.35	N/A
adder32bit	N/A	16	N/A	N/A	10.689	N/A	N/A	13.01	N/A	32	11	65.63%	18.425	4.788	74.01%	18.62	18.61	0.05%
adderpipe	731	583	20.25%	53.239	52.206	1.94%	39.47	39.43	0.10%	729	781	-7.13%	56.17	54.517	2.94%	39.47	39.47	0.00%
asynfifo	N/A	485	N/A	N/A	9.223	N/A	N/A	16.7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

and an Area enhancement of 11.79%. Similarly, for the RTLCoder, we observed an average Performance increase of 7.86%, a Power increase of 5.27%, and a slight Area improvement of 0.88%.

4 Acknowledgements

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

References

[1] Christopher Berner, Greg Brockman, et al. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR* abs/1912.06680 (2019). arXiv:1912.06680

[2] Jason Blocklove, Siddharth Garg, et al. 2023. 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, 1–6. https://doi.org/10.1109/MLCAD58807.2023.10299874

[3] Paul F. Christiano, Jan Leike, et al. 2017. Deep reinforcement learning from human preferences. In *31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4302–4310.

[4] Hung Le, Yue Wang, et al. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. 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.*

[5] Sergey Levine, Aviral Kumar, et al. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *CoRR* abs/2005.01643 (2020).

[6] Mingjie Liu, Nathaniel Ross Pinckney, et al. 2023. 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, 1–8. https://doi.org/10.1109/ICCAD57390.2023.10323812

[7] Shang Liu, Wenji Fang, et al. 2023. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *CoRR* abs/2312.08617 (2023). https://doi.org/10.48550/ARXIV.2312.08617 arXiv:2312.08617

[8] Yao Lu, Shang Liu, et al. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. In *29th Asia and South Pacific Design Automation Conference* (Incheon, Republic of Korea) *(ASPDAC '24)*. IEEE Press, 722–727. https://doi.org/10.1109/ASP-DAC58780.2024.10473904

[9] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[10] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

[11] Rafael Rafailov, Archit Sharma, et al. 2023. 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.*

[12] David Silver, Julian Schrittwieser, et al. 2017. Mastering the game of Go without human knowledge. *Nat.* 550, 7676 (2017), 354–359. https://doi.org/10.1038/NATURE24270

[13] Shailja Thakur, Baleegh Ahmad, et al. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. Design Autom. Electr. Syst.* 29, 3 (2024), 46:1–46:31. https://doi.org/10.1145/3643681

[14] Ashish Vaswani, Noam Shazeer, et al. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA.* 5998–6008.