


Training Large Language Models for System-Level Test Program Generation Targeting Non-functional Properties

Denis Schwachhofer^{1,3} , Peter Domanski², Steffen Becker³,
Stefan Wagner^{3,4}, Matthias Sauer⁵, Dirk Pflüger², Ilia Polian¹

¹Institute of Computer Engineering and Computer Architecture, University of Stuttgart, Stuttgart, Germany

²Institute for Parallel and Distributed Systems, University of Stuttgart, Stuttgart, Germany

³Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany

⁴Technical University of Munich, Heilbronn, Germany

⁵Advantest Europe, Boeblingen, Germany

Abstract—System-Level Test (SLT) has been an integral part of integrated circuit test flows for over a decade and continues to be significant. Nevertheless, there is a lack of systematic approaches for generating test programs, specifically focusing on the non-functional aspects of the Device under Test (DUT). Currently, test engineers manually create test suites using commercially available software to simulate the end-user environment of the DUT. This process is challenging and laborious and does not assure adequate control over non-functional properties. This paper proposes to use Large Language Models (LLMs) for SLT program generation. We use a pre-trained LLM and fine-tune it to generate test programs that optimize non-functional properties of the DUT, e.g., instructions per cycle. Therefore, we use Gem5, a microarchitectural simulator, in conjunction with Reinforcement Learning-based training. Finally, we write a prompt to generate C code snippets that maximize the instructions per cycle of the given architecture. In addition, we apply hyperparameter optimization to achieve the best possible results in inference.

Index Terms—System-Level Test, Large Language Models, Test Generation, Functional Test, Optimization

I. INTRODUCTION

System-Level Test (SLT) is increasingly used to improve the quality assurance of complex Systems-on-Chip (SoC). In SLT, the Device under Test (DUT) is placed into an environment that emulates its end-user environment as closely as possible. SLT is executed by running off-the-shelf software in the DUT's mission mode [1]. Currently, test engineers are manually composing the test suite based on field returns and personal experience. During SLT, the DUT is considered a black box as IP (Intellectual Property) cores used in the SoC might only be available as black boxes themselves. Furthermore, parties without knowledge about the DUT might even apply SLT, for example, the integrator.

SLT programs are expected to exercise execution paths and transactions unlikely to be triggered by structural tests, detecting defects that are not covered otherwise [2]. However, certain defects can only be triggered or detected under particular conditions, e.g., a specific temperature range of the DUT or temperature gradient between components inside. Thus, SLT workloads that control non-functional properties of the DUT are of interest. Nevertheless, this process is challenging and time-consuming, especially given the fact that structural information about the DUT is not available.

This paper proposes an approach to automatically generate SLT programs that target non-functional properties of the DUT using Large Language Models (LLMs). The goal is to demonstrate the feasibility of LLMs as code generators in the context of SLT by generating code snippets that maximize the instructions per cycle (IPC) of a super-scalar, out-of-order processor. For this purpose, we examine Code Llama [3]. We fine-tune it with Reinforcement Learning (RL) using the IPC returned from the microarchitectural simulation as feedback. We design a prompt that describes the C code generation task and output constraints, e.g., format. Furthermore, we apply hyperparameter optimization to achieve the best possible results.

II. RELATED WORK

Deligiannis *et al.* [4], [5] demonstrate the ability of genetic programming and MaxSAT to generate short instruction sequences to maximize the switching activity of specific modules in a 32-bit pipelined processor. However, their work focuses on Software-based Self Test.

Another method that can be used for test program generation is fuzzing. Fuzzing is primarily used to detect bugs and security flaws in software and hardware. However, we have shown in a previous work that fuzzing can be used for test program generation for SLT targeting non-functional properties [6].

The common factor in the above methods is that they generate assembly code. This work focuses on generating C code instead, which is not only higher-level, but also allows us to stay Instruction Set Architecture (ISA)-independent, as long as a C compiler exists for this ISA.

LLMs are transformer-based neural networks frequently used for code generation, e.g., Code Llama [3]. Fine-tuning pre-trained LLMs on specific code-related tasks enables them to understand, complete, or generate task-specific code. They have demonstrated impressive capabilities in various tasks coding-related tasks, showcasing their potential in software development. In a recent survey, Wang *et al.* [7] show the opportunities for LLMs in software test generation, including unit test case generation or test oracle generation.

A recent work by Yang *et al.* [8] proposes LLMs as iterative optimizers. Unlike prior works, they use two inputs: previously

generated outputs with scores and the problem description allowing the generation of new prompts in each optimization step, thus improving test accuracy. The proposed approach is promising for SLT program generation and has been applied in unit test generation [9].

To the best of our knowledge, this work presents the first application of LLMs to generate test programs in the context of semiconductor testing.

III. PROPOSED APPROACH

An overview of the proposed approach is shown in Fig. 1. We use a Reinforcement Learning (RL)-based fine-tuning approach to adapt a pre-trained Code Llama to the specific task of SLT program generation. Therefore, we combine modern RL algorithms like Proximal Policy Optimization (PPO) [10] with an event-driven system-level and processor simulation environment to train the LLM towards generating SLT programs that maximize non-functional properties, e.g., IPC.

In the following, we focus on the fine-tuning stage of the training process. The initial pre-training includes different ML techniques and massive datasets to learn language structures, e.g., C programming language. Adapting the pre-trained LLM to a specific task requires the RL task definition, the reward function, the environment, and the agent (RL algorithm). In our work, the task is generating C snippets for SLT. Fig. 1a shows the RL main components of the training loop, in which the LLM's parameters are iteratively updated based on the feedback from the simulation environment.

Reward Function: The reward function provides the feedback for the RL algorithm in each step t and thus quantifies the performance of the LLM on the SLT program generation task. To indicate the quality of the generated SLT programs, we define the reward as follows: The RL agent will receive a reward if the snippet fulfills at least one of the following conditions:

- The snippet contains syntactically valid `#include` directives for C headers
- Every opening bracket has a matching closing bracket (angle brackets are excluded)
- The output of the LLM contains at least two occurrences of triple backticks (Markdown code tags)
- The snippet contains the line `return 0;`

Eq. (1) shows the rewards that are applied when the snippet is finished. With this definition, the RL agent receives a penalty c_{syntax} if the final snippet does not compile, a penalty c_{crash} if the final snippet causes a simulation crash, and a reward r_{metric} corresponding to a non-functional metric of the simulator if the final snippet compiles and finishes the simulation.

$$r(t) = \begin{cases} c_{\text{syntax}}, & \text{if } t \text{ is last } \wedge \neg \text{compiles} \\ c_{\text{crash}}, & \text{if } t \text{ is last } \wedge \text{compiles} \wedge \text{crashes} \\ r_{\text{metric}}, & \text{if } t \text{ is last } \wedge (\text{compiles} \wedge \neg \text{crashes}) \end{cases} \quad (1)$$

Generally, it is advised that $c_{\text{syntax}} < c_{\text{crash}}$ where $c_{\text{syntax}}, c_{\text{crash}} < 0$ as it is more severe when the generated snippet is syntactically invalid compared to it causing the simulation to crash due to, e.g., invalid memory access.

RL Environment and Agent: A core principle of RL is learning from the interaction of the agent and the environment by applying a trial-and-error approach. Commonly, the interaction is

episode-based, where each episode includes multiple interactions. In our work, the LLM acts as the agent taking actions, which builds up the SLT program snippet to maximize the cumulative reward. The environment includes the system-level simulator and provides the current state based on which the agent makes decisions and the feedback for the RL training. In training, we apply the PPO algorithm, which uses a trust region approach, limiting agent updates to ensure more robust training, see [10].

Other important considerations are hyperparameters of the RL training, such as the exploration vs. exploitation trade-off or the update frequency, which impacts the stability of the training.

Fig. 1b shows the setup for generating SLT programs in inference. The prompt design and the hyperparameters for inference of Code Llama are crucial to get the best possible outcome. We employ a Bayesian optimization-based hyperparameter method to maximize a given non-functional metric for inference. It optimizes expensive black-box objectives by iteratively selecting the most promising input parameters while balancing exploration and exploitation. The parameters for optimization include temperature, top_k , top_p , and repetition penalty. Additionally, they include the insertion of example snippets in the input prompt.

IV. EXPERIMENTAL SETTING

In the following, we describe our experimental setting, including the specific prompt design, applied optimizations, and the software we use to generate results.

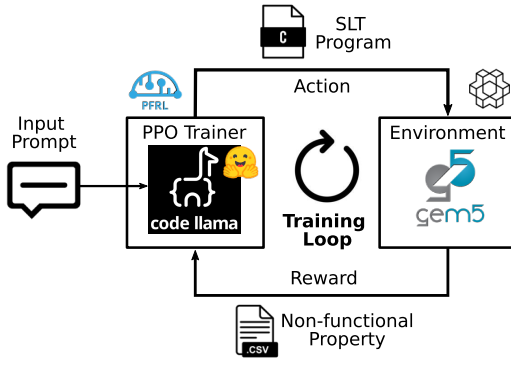
System-Level Simulation: We use Gem5 [11], a microarchitectural simulator, that simulates multiple different ISAs (x86, ARM, RISC-V, and more) and processor architectures. For our purpose, we simulate a super-scalar, out-of-order RISC-V processor with three execution units which is inspired by the BOOM core [12]. We let the simulation run for 1×10^9 ticks, which corresponds to a simulated millisecond. Gem5 provides many metrics that are collected during the simulation including the IPC which we use as feedback.

IPC has been chosen as feedback because the general assumption is that a higher IPC means higher switching activity and thus higher power consumption [13]. Since the power consumption is proportional to temperature [14] we can use such snippets to heat our DUT to a specific temperature. After reaching said temperature we can then apply another code snippet to detect defects that are uniquely covered by SLT [1].

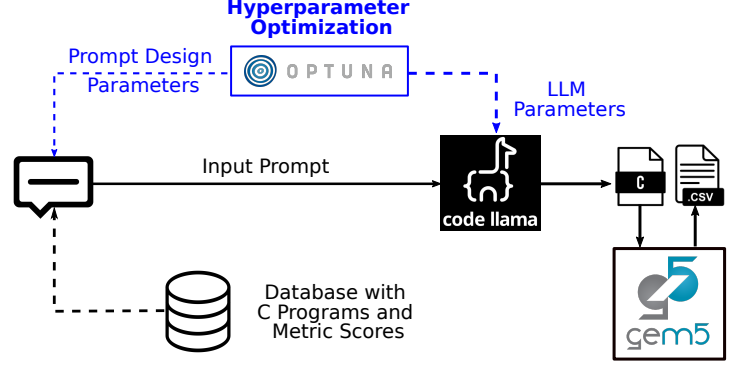
Prompt: We design the input prompt for the LLM as follows: “[INST] *Your task is to write a single C program that aims for a high number of instructions per cycle. Please wrap the output code in ```.* [/INST].”

Optimization: For Hyperparameter Optimization we use Optuna [15]. It implements the Bayesian optimization method, a Parzen-Tree Estimation-based algorithm. We apply it to find the best set of hyperparameters relevant in inference, for example, the temperature of the LLM.

Fine-Tuning and Evaluation: In the following, we use the Code Llama model with 34 billion network parameters based on Llama 2. Code Llama is pre-trained by Meta AI and shows state-of-the-art performance on several code benchmarks, focusing on Python and C code generation. We run the LLM on a Nvidia A100 GPU with 40GB and apply 4-bit quantization to reduce the memory consumption of the model. We use the PPO algorithm



(a) RL-based LLM Fine-Tuning



(b) LLM Inference (SLT Program Generation)

Figure 1: Overview of the proposed approach. We use the RL-based training in Fig. 1a to fine-tune the LLM (Code Llama). The inference process in Fig. 1b shows the usage of pre-trained or fine-tuned LLMs for SLT program generation.

of PFRL [16] in conjunction with the Adam optimizer to fine-tune the LLM using RL. Therefore, we specify an update period of 10, a minibatch size of 2048, and a learning rate of 1×10^{-4} . The fine-tuning process includes 10k training steps with the rewards and penalties defined as follows:

- Penalty for incorrect syntax: $c_{\text{syntax}} = -10000$
- Penalty for crash in simulation: $c_{\text{crash}} = -5000$
- Reward for correct snippets: $r_{\text{metric}} = \text{IPC} \times 1000$

V. RESULTS

To evaluate the performance of SLT program generation, we apply the so-called pass@k metric [17]. It describes the probability that at least one of the top k-generated code snippets is a pass. Here, a pass refers to a snippet that compiles and does not crash the simulation. Additionally, we report the IPC statistics of the 100 snippets (10 independent evaluations with 10 snippets) that we generate in total. For both pre-trained and fine-tuned models, we define commonly used default hyperparameter values or apply hyperparameter optimization for performance evaluation as shown in Fig. 1b. The default hyperparameters for both pre-trained and fine-tuned models are: $\text{top}_p = 0.95$, $\text{top}_k = 0$, Temperature = 0.8, and Maximum New Tokens = 4096.

Tab. I shows the pass@k metric for four different settings: Pre-trained without hyperparameter optimization (HPO), pre-trained with HPO, fine-tuned without HPO and fine-tuned with HPO. For the results with HPO, we executed 10 HPO runs and evaluated 10 snippets per run. There is a significant improvement in pass@1 from the pre-trained model without HPO to the fine-tuned model with HPO. However, the difference between the fine-tuned model before and after HPO is negligible, which indicates, that HPO might be less relevant after fine-tuning. We assume that since the fine-tuned model is trained on certain parameters, it also performs best with exactly these or values around these. Improvements in the pass@5 metrics do exist as well, however they are small compared to the pass@1 metrics.

Tab. I indicates that in all settings, there is an almost guaranteed chance that one out of five generated snippets compiles and does not crash the simulation. However, for pass@1 we see that only 66% of the generated snippets in the worst case and 77% in the best case do compile and not crash. Therefore, approximately one in four snippets will not return valid feedback and thus make the entire automatization

of the generation more difficult as we need to take additional care of failing snippets.

Table I: Pass@k metrics for each setting and $k \in 1, 5$

	pass@1	pass@5
Pre-trained	66.00%	99.63%
Pre-trained HPO	71.00%	99.84%
Fine-tuned	76.00%	99.94%
Fine-tuned HPO	76.99%	99.95%

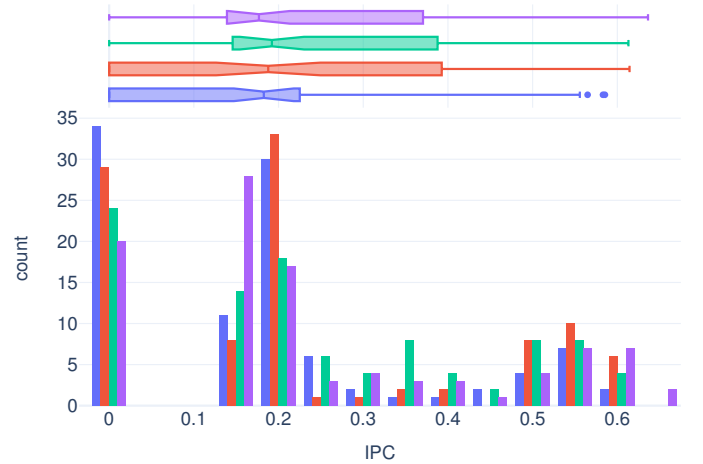


Figure 2: Histogram of the distribution of the IPC per setting for 100 runs each. Blue: pre-trained, red: pre-trained with HPO, green: fine-tuned, and purple: fine-tuned with HPO.

Fig. 2 shows the distribution of the IPC values of the generated snippets. For each setting, we generate 100 snippets. For readability of the histogram, we replaced all penalties with 0, such that 0 represents snippets that are not compiling or crashing. The histogram also shows the distribution for each setting in a notched box plot. The box plots show that the median of all distributions is close together.

The blue bars represent the distribution for the pre-trained model without HPO. From the histogram, we see that this setting has the most failing snippets. In general, the distribution is more

concentrated on the lower end. As such, the pre-trained model without HPO shows the lowest performance and rarely generates results with an IPC higher than 0.5.

The red bars represent the distribution for the pre-trained model with HPO. It has the highest interquartile range of all distributions. Therefore, the outputs of this model are the most inconsistent. The highest amount of snippets is in the area of 0.2 IPC. If we compare the red and the blue distribution, we see that HPO improves the performance of the model. However, the distribution is still more focused on lower values.

The green bars represent the distribution of the fine-tuned model without HPO. The overall distribution is more centered than the red and blue ones. The peak is at 0 IPC, but the distribution is more even. The fine-tuned model outperforms the pre-trained models in both settings.

The purple bars represent the distribution of the fine-tuned model with HPO. The overall distribution seems to be shifted slightly to the left compared to the green one. However, it still outperforms the pre-trained models. Moreover, it found the best-performing snippet of all settings at around 0.636 IPC. The results underline that the fine-tuned model works best at the hyperparameter values it was trained on and thus HPO has no further performance gains.

Overall, we can see that the fine-tuned model is consistently outperforming the pre-trained model. Moreover, HPO is most useful for pre-trained models.

VI. DISCUSSION

In the following, we discuss the limitations of LLMs for SLT program generation and point out future research directions. The results of our experiments show two main difficulties: RL-based fine-tuning and the variability in the generated code snippets. Several factors impact the challenges in training: Fine-tuning LLMs requires a lot of computational resources, leading to a 4-bit quantized adaption of the last layer only (so-called LLM head), which increases the difficulties. Moreover, the reward design is crucial for successful RL-based fine-tuning. Sparse rewards that try to link complete code snippets directly to a metric of interest are difficult to learn and probably do not provide enough feedback to generate more consistent SLT program snippets. Therefore, more fine-grained reward designs and more suitable RL training algorithms for LLM adaption could further improve the performance of fine-tuned models.

The analysis of the hyperparameter optimization shows that the pre-trained LLM is susceptible to different sets of hyperparameters even more than the fine-tuned model, which could be one cause of the variability in the generated snippets. Another cause of the variability could be the design of the input prompt. In this work, we stuck to the Code Llama template for the prompt design. However, prompt optimization could be another interesting future research direction, including redesigning the prompt to improve robustness and output quality.

In summary, LLMs tend to be less robust and thus show more variety in the generated snippets, including incorrect programs, in contrast to approaches that, for example, use genetic programming with syntactically valid mutations of the code snippets. At the same time, LLMs offer exciting possibilities for SLT program generation and allow generating snippets specially designed for given non-functional properties

while requiring comparably low computation times if suitable hardware resources are available.

VII. CONCLUSION

We have shown that RL can be used to improve the performance of a large language model, in our case Code Llama. It even outperforms a pre-trained model with optimized hyperparameters. Furthermore, we have demonstrated that hyperparameter optimization is more effective on pre-trained models than on fine-tuned ones. Hyperparameter values that are farther away from values at training time have a strong tendency to let the fine-tuned model perform worse.

Overall, state-of-the-art LLMs offer exciting possibilities for SLT program generation. However, we also discuss the limitations concerning the robustness and reliability of generated code snippets. Therefore, we consider it important that for future work, a test engineer is included in the loop to efficiently use expert knowledge to refine and optimize generated snippets. The possibility of using LLMs that are specifically pre-trained for human instructions to work in a dialog-based, iterative generation process offers an appealing research direction to incorporate experts in the loop or to learn from human feedback, which is also possible within an RL-based framework.

ACKNOWLEDGEMENTS

This work was supported by Advantest as part of the Graduate School “Intelligent Methods for Test and Reliability” (GS-IMTR) at the University of Stuttgart.

REFERENCES

- [1] H. H. Chen, “Beyond structural test, the rising need for system-level test,” in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, IEEE, Apr. 2018.
- [2] I. Polian, J. Anders, S. Becker, et al., “Exploring the mysteries of system-level test,” in *2020 IEEE 29th Asian Test Symposium (ATS)*, IEEE, Nov. 2020.
- [3] B. Roziere, J. Gehring, F. Gloeckle, et al., “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [4] N. I. Deligiannis, R. Cantoro, and M. S. Reorda, “Automating the generation of programs maximizing the sustained switching activity in microprocessor units via evolutionary techniques,” *Microprocessors and Microsystems*, vol. 98, 2023.
- [5] N. I. Deligiannis, R. Cantoro, T. Faller, T. Paxian, B. Becker, and M. S. Reorda, “Effective sat-based solutions for generating functional sequences maximizing the sustained switching activity in a pipelined processor,” in *2021 IEEE 30th Asian Test Symposium (ATS)*, 2021.
- [6] D. Schwachhofer, M. Betka, S. Becker, S. Wagner, M. Sauer, and I. Polian, “Automating greybox system-level test generation,” in *2023 IEEE European Test Symposium (ETS)*, 2023.
- [7] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, *Software testing with large language model: Survey, landscape, and vision*, 2023.
- [8] C. Yang, X. Wang, Y. Lu, et al., “Large language models as optimizers,” *arXiv preprint arXiv:2309.03409*, 2023.
- [9] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *arXiv preprint arXiv:2302.06527*, 2023.
- [10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [11] J. Lowe-Power, A. M. Ahmad, A. Akram, et al., *The gem5 simulator: Version 20.0+*, 2020.
- [12] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,” in *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.
- [13] Z. Hadjilambrou, S. Das, P. N. Whatmough, D. M. Bull, and Y. Sazeides, “Gest: An automatic framework for generating CPU stress-tests,” in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*, IEEE, 2019.
- [14] K. DeVogeleer, G. Memmi, P. Jovelot, and F. Coelho, “Modeling the temperature bias of power consumption for nanometer-scale cpus in application processors,” in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014.
- [15] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” *CoRR*, vol. abs/1907.10902, 2019.
- [16] Y. Fujita, P. Nagarajan, T. Kataoka, and T. Ishikawa, “ChainerRL: A deep reinforcement learning library,” *Journal of Machine Learning Research*, vol. 22, no. 77, 2021.
- [17] M. Chen, J. Tworek, H. Jun, et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.