Focus Session: Do Agentic LLMs Change the Paradigm of Hardware Test Generation?

Farshad Firouzi*, Pragya Sharma*, Agastya Seth*, Peter Domanski*,
Bahar Farahani†, Sanmitra Banerjee‡, Jonti Talukdar‡, and Krishnendu Chakrabarty*
*Arizona State University, Tempe, USA
†Cyberspace Research Institute, Shahid Beheshti University, Tehran, Iran
‡NVIDIA, USA

***Abstract*—Technology scaling and increasing System-on-Chip (SoC) complexity exacerbate reliability challenges arising from both structural defects and runtime-dependent failures, including Silent Data Corruptions (SDCs) that evade traditional error detection mechanisms. Structural testing remains essential for detecting modeled faults such as stuck-at faults; however, it is inherently limited in capturing failures that arise under dynamic operating conditions. In contrast, functional testing can expose workload-dependent failures, albeit at the cost of high testing overhead and largely unguided workload generation. This paper presents an agentic testing framework that integrates Large Language Models (LLMs) with Reinforcement Learning (RL) and Tree-structured Parzen Estimators (TPE) to guide functional workload generation and Automatic Test Pattern Generation (ATPG) settings under user-defined constraints. The proposed approach leverages feedback-driven optimization to steer test generation toward failure-prone behaviors while reducing reliance on manual expertise. Experimental evaluation on a RISC-V processor core demonstrates that the method outperforms manually generated workloads for functional testing, while experiments on six benchmark circuits show test quality comparable to expert-generated ATPG scripts for structural testing, with improved efficiency and scalability.**

***Index Terms*—Hardware Testing, Silent Data Corruption (SDC), Large Language Models (LLMs), Reinforcement Learning**

I. Introduction

Rapid advances in semiconductor technology and the growing complexity of System-on-Chip (SoC) architectures have significantly increased the challenges of ensuring reliable computation [1]. Modern SoCs integrate billions of transistors to support workloads such as Artificial Intelligence (AI), high-performance computing, and data-intensive applications [2], [3], substantially increasing design and validation complexity. Consequently, these systems are more susceptible to reliability threats including process variations, transient voltage fluctuations, device aging, and signal integrity degradation, which can compromise timing and functional correctness [4]–[9]. These effects can manifest as both structural defects and runtime-dependent functional errors that propagate through the system and lead to incorrect execution outcomes [2].

Addressing these reliability challenges requires complementary testing methodologies across the hardware lifecycle. Structural testing, such as Automatic Test Pattern Generation (ATPG)–based manufacturing tests, is essential for detecting modeled defects [3]; however, it is inherently limited to static operating conditions and predefined fault models. In contrast, workload-dependent phenomena in modern SoCs (e.g., voltage droop caused by rapid power fluctuations) can induce timing violations that occur only during execution and may evade detection. Functional and workload-based testing complement structural testing by exercising the system under realistic operating conditions to expose such runtime-dependent failures [2]. Despite their effectiveness, deploying complementary testing methodologies at scale poses significant practical challenges. Structural testing requires substantial manual effort and domain expertise to configure test scripts and tune the corresponding parameters such as scan chains, directly impacting coverage, test time, power, and area overhead. At the system level, production functional testing approaches rely on proprietary workloads and massive test volumes to achieve sufficient coverage, resulting in largely unguided workload generation, long testing times, and limited efficiency during constrained maintenance windows [2].

To address these limitations, we propose an agentic Large Language Model (LLM)–based testing approach that integrates Reinforcement Learning (RL) and Tree-structured Parzen Estimators (TPE) to guide both functional and structural test generation. The proposed approach employs an LLM to generate candidate functional workloads and ATPG scripts, while RL and TPE provide feedback-driven guidance by adapting prompts, constraints, and optimization policies to steer the LLM toward test cases that better satisfy user-defined objectives. The approach enables systematic tuning of workload characteristics for functional testing as well as ATPG script configuration parameters for structural testing, subject to constraints on fault coverage, test time, power consumption, and area overhead. By leveraging optimization feedback to guide LLM generation rather than relying on unguided sampling, the proposed approach assists in automating several expert-driven decisions involved in workload generation and ATPG configuration. This reduces manual effort in workload design and ATPG configuration while enabling more systematic exploration of the test space.

The rest of this paper is organized as follows: Section II reviews related work. Section III presents the proposed LLM-based framework for functional and structural test generation. Section IV describes the experimental setup and analyzes the

978-3-9826741-1-7/DATE26/© 2026 EDAA

results. Finally, Section V concludes the paper.

II. Related Prior Work

A. LLM for Hardware Design

LLMs have demonstrated considerable potential in advancing Electronic Design Automation (EDA) by supporting design-time tasks such as script generation, code generation, and security analysis [10]–[16]. Prior work has explored the use of LLMs for code generation from high-level specifications, enabling automated assistance in hardware design workflows [17]–[21]. LLM-based script generation has also been proposed to automate and streamline EDA flows, reducing manual effort in design orchestration [22]–[26]. In the domain of hardware security, LLMs have been explored for both adversarial and defensive use cases, including the automated generation of digital Trojans as well as the synthesis of security assertions and design rule checks [27]–[31].

B. LLM for Test Generation

LLMs have been widely applied to automated test generation, demonstrating effectiveness in software testing tasks such as unit testing, system-level input generation, and debugging [32], [33]. In hardware verification, LLMs have been used to generate architecture-aware test programs from system-level specifications, particularly for Verilog [18], [22], [34], [35] and VHDL [36], [37]. Approaches such as LLM4DV show that LLM-generated stimuli can outperform constrained-random testing (CRT) [23]. LLMs have also been applied to system-level functional testing, complementing existing frameworks such as SiliFuzz [38]–[40]. For Silent Data Corruptions (SDC) detection, prior work has explored automated and hardware-in-the-loop testing methodologies, as well as LLM-based generation of workloads to improve in-field fault coverage [2], [41].

III. Methodology

This section introduces an LLM-based methodology for automated generation of both structural and functional tests. LLMs generate ATPG scripts for scan-based testing and optimize software workloads for functional testing (see Fig. 1).

A. Functional Testing

In this work, the goal of functional testing is to generate a stress workload implemented in C code that maximizes on-chip voltage droop in order to increase the likelihood of SDC and timing failures. To achieve this, we exploit two complementary LLM-driven optimization techniques: (1) iterative LLM-based optimization and (2) RL-guided optimization [2].

1) Iterative LLM-based Optimization: The iterative LLM-based optimization approach treats the LLM as an optimizer that incrementally improves previously generated C test programs. The LLM is invoked repeatedly to correct syntactic issues and refine workload structure to maximize a target metric, namely voltage droop within a designated functional unit. Feedback in the form of voltage droop measurements from earlier iterations is provided to guide subsequent refinements. To enhance optimization, the LLM is augmented with a knowledge base derived from design space exploration, which captures relationships between instructions, instruction sequences, and their likelihood of inducing voltage emergencies, defined as critical voltage fluctuations beyond predefined operating margins [42]. Instruction traces from representative workloads are analyzed to identify stress-inducing patterns. The LLM additionally receives layout images annotated with per-cell voltage droop values, enabling correlation between workload behavior and localized droop hotspots.

2) RL-guided Optimization: The RL-guided optimization approach extends the iterative strategy by integrating the LLM with a reinforcement learning agent that provides directed refinement strategies (see Fig. 1) [2]. The optimization process is formulated as a Markov Decision Process (MDP) defined by the tuple (S, A, R), where S represents the workload state, A represents the set of workload transformation actions, and R denotes the reward derived from voltage droop measurements.

The RL agent observes a high-dimensional embedding of the generated test program extracted using a BERT encoder-only model. Formally, each workload W_t at iteration t is encoded as

$$s_t = f_{\text{BERT}}(W_t) \tag{1}$$

where $s_t \in \mathbb{R}^{768}$ captures structural characteristics of the workload, including instruction composition and control-flow patterns. Based on this representation, the RL agent selects high-level refinement actions aimed at maximizing voltage droop in a target functional unit. The action space consists of workload transformation directives derived from design space exploration, including increasing arithmetic or load–store intensity, adjusting control-flow density, and reducing logic- or shift-dominated instruction sequences. After selecting an action $a_t \in A$, the workload is transformed as

$$W_{t+1} = T(W_t, a_t) \tag{2}$$

where $T(\cdot)$ denotes the transformation applied to the program.

The reward function is defined as the improvement in maximum voltage droop observed after applying the selected strategy:

$$R_t = V_{\text{droop}}(W_{t+1}) - V_{\text{droop}}(W_t) \tag{3}$$

where $V_{\text{droop}}(\cdot)$ denotes the measured voltage droop during simulation. This reward encourages the agent to identify instruction patterns (i.e., action) that increase switching activity and stress the Power Delivery Network (PDN). The action space consists of workload transformation directives derived from design space exploration, such as increasing arithmetic or load–store intensity, adjusting control-flow density, and reducing instruction sequences dominated by logic or shift operations. These directives are supplied to the LLM as prompts, allowing it to iteratively refine workload generation based on feedback from the RL agent.

The agent is trained using the Proximal Policy Optimization (PPO) algorithm for 250 episodes with a batch size of 16, employing a multi-layer perceptron with three hidden layers to represent the policy $\pi_\theta(a|s)$. Each episode consists of ten interactions between the agent and the simulation environment, allowing the policy to progressively learn workload refinement strategies that maximize voltage droop in targeted functional units.

B. Structural Testing

For structural testing, we employ two complementary LLM-driven optimization techniques: (1) iterative LLM-based optimization and (2) TPE-guided optimization. Both approaches automate the generation and refinement of ATPG test scripts with the objective of maximizing stuck-at fault coverage under constraints on area, power, and test time overheads.

1) Iterative LLM-based Optimization: Algorithm 1 outlines the proposed iterative LLM-based optimization flow for structural test generation, in which the LLM acts as the primary optimizer. The flow begins by providing few-shot examples to establish the ATPG syntax and context, enabling the LLM to generate an initial ATPG script. This script is then iteratively refined to improve fault coverage under user-defined constraints (i.e., area overhead, power overhead, and test time). The optimization targets key ATPG parameters (e.g., the number of scan chains, test point insertion, compaction effort). In each iteration, the generated script is executed by the ATPG tool, and feedback extracted from the ATPG reports (e.g., achieved fault coverage and associated overheads) is used to guide subsequent refinements. The refinement process is supported by Chain-of-Thought (CoT) reasoning and a knowledge graph (KG) containing ATPG-specific rules and relationships between configuration choices and observed outcomes, enabling systematic exploration of the ATPG configuration space.

2) TPE-guided Optimization: The TPE-guided optimization approach extends the iterative strategy by incorporating a Tree-structured Parzen Estimator (TPE) to guide parameter selection during ATPG script generation. TPE is a Bayesian optimization technique that models the relationship between configuration parameters and observed objective values using probabilistic density estimation. Given previously evaluated configurations, TPE partitions the observations into promising and non-promising regions of the search space and constructs two probability density models: $l(x)$ representing high-performing configurations and $g(x)$ representing lower-performing ones. New candidate parameter configurations are sampled to maximize the ratio $l(x)/g(x)$, which corresponds to maximizing the expected improvement over previously observed results.

In the context of structural test generation, the parameter vector x includes ATPG configuration parameters such as the number of scan chains, test-point insertion effort, and compaction settings. Each candidate configuration proposed by the LLM is evaluated by executing the generated ATPG script, producing metrics such as fault coverage and overheads. Next, the observed results are used to update the TPE model, which guides subsequent parameter exploration toward configurations that improve fault coverage while satisfying user-defined constraints. By combining probabilistic search from TPE with the reasoning capabilities of the LLM, the proposed framework enables efficient exploration of the ATPG configuration space without requiring exhaustive design space exploration.

C. Optimization Strategies

We employ complementary optimization strategies to guide LLM-driven test generation using feedback, structured reasoning, and design space exploration.

In-Context Learning (ICL). ICL enables the LLM to leverage task-specific templates, examples, and instructions provided in the prompt to generate test artifacts without explicit fine-tuning. Example templates and annotated outputs are used to guide output structure and correctness, while embedded domain knowledge and optimization objectives help reduce hallucinations and improve test quality.

Chain-of-Thought Reasoning (CoT). CoT prompting encourages step-by-step reasoning, allowing the LLM to systematically handle complex dependencies among optimization parameters and constraints. CoT supports iterative refinement by enabling the model to reason over feedback from previous iterations, improving constraint satisfaction and robustness.

Design Space Exploration. Design space exploration (DSE) is used to analyze the impact of optimization knobs and workload characteristics on target metrics such as coverage, runtime, and overheads. In the context of Structural testing, by executing multiple test configurations and workloads, we identify relationships between test parameters, design features, and observed outcomes (See Fig. 2). The resulting insights are distilled into heuristic rules and structured knowledge that guide the LLM during iterative test refinement, enabling informed trade-offs and faster convergence under user-defined constraints. For functional testing, DSE analyzes instruction-level behaviors and instruction sequence patterns to identify workloads that induce high dynamic stress, such as supply voltage droops in targeted functional units

IV. Experimental Results

A. Experimental Setup

The first experimental setup corresponds to functional testing and targets functional workload-dependent voltage droop analysis, in contrast to the ATPG-based structural testing setup. It evaluates system behavior and power integrity effects under realistic, mission-mode program execution. The CV32E40P 32-bit RISC-V core is implemented using a complete RTL-to-GDSII design flow in TSMC 28 nm technology. RTL synthesis is performed using Synopsys Design Compiler, followed by scan chain insertion with Synopsys DFT Compiler, and full physical implementation including floorplanning, placement, routing, power distribution network insertion, and clock tree synthesis using Cadence Innovus. Signoff verification is carried out using Synopsys PrimeTime to ensure timing correctness. Functional workloads are compiled and executed on the core, and the resulting switching activity is captured as VCD files from simulations within the CORE-V-Verif framework. These

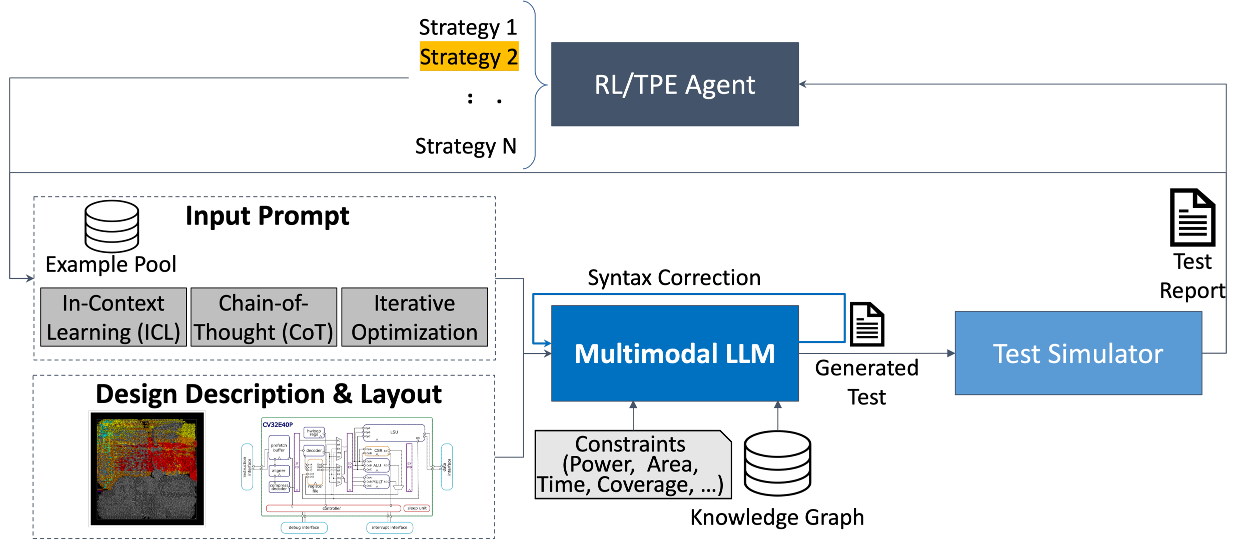

Fig. 1: Proposed agentic LLM-driven framework for functional and structural testing.

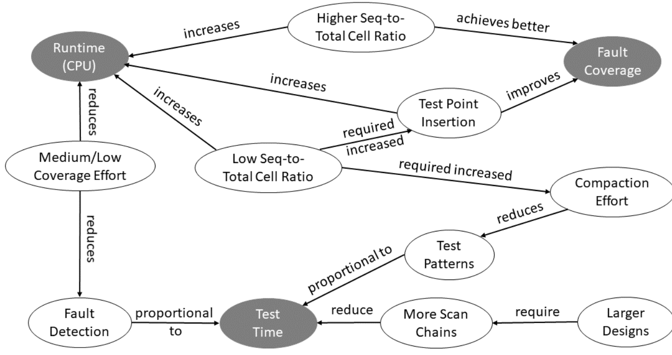

Fig. 2: Partial view of the KG employed in the testing framework.

activity traces are used by Cadence Voltus to perform dynamic voltage droop analysis. The core operates at 1 GHz with a 0.9 V supply voltage and a simulation resolution of 1 ns. Ten representative workloads from CoreMark, MiBench, and Embench are used to exercise the processor, with the analysis focused on the arithmetic logic unit and load store unit due to their high sensitivity to workload-induced voltage droops.

The second experimental setup corresponds to structural testing and follows a scan-based ATPG flow applied to a set of six RTL circuit designs listed in Table I. For each circuit, the RTL is synthesized into a gate-level netlist. GPT-4 is used to automatically generate Siemens Tessent ATPG scripts, which drive scan chain insertion, configuration, and test pattern generation. Using these scripts, Siemens Tessent is employed to insert scan logic and generate ATPG patterns targeting stuck-at fault models. The resulting Tessent reports are then analyzed to extract structural test metrics, including achieved fault coverage and test characteristics, which are used for comparative evaluation across different configurations. In this setup, structural cost metrics are derived directly from Tessent reports. Area and power overheads are approximated as $A_{\text{overhead}} = N_{\text{seq}}\Delta A + \alpha N_{\text{chains}} A_{\text{cell}} + N_{\text{tp}} A_{\text{cell}}$ and $P_{\text{overhead}} = N_{\text{seq}}\Delta P + \alpha N_{\text{chains}} P_{\text{cell}} + N_{\text{tp}} P_{\text{cell}}$, where N_{seq}, N_{chains}, and N_{tp} denote the number of sequential elements, scan chains, and test points, respectively; ΔA and ΔP denote the incremental area and power overhead incurred by converting a functional sequential element into its scan-enabled counterpart; A_{cell} and P_{cell} denote

Algorithm 1 Coverage-aware Structural Testing Flow

1: **Input:** ATPG_Task_Description, User_Constraints, Overhead_Formulas, Max_Iterations
2: **Output:** Optimized_ATPG_Script, Run_Statistics
3: **procedure** TestingOptimizationFlow
4: InitializeDesignExploration()
5: PopulateKnowledgeGraph()
6: PopulateATPGCommandReference()
7: ProvideFewShotExamplesToLLM()
8: iteration_count = 0
9: **while** not Convergence() **and** iteration_count < Max_Iterations **do**
10: GenerateATPGScript() ▷ Uses LLM and self-correction to generate a new syntactically correct ATPG script
11: ExecuteATPGScript() ▷ Run the script using the Tessent ATPG tool
12: EvaluateResults() ▷ Collect and analyze fault coverage, user constraints, and calculate overheads
13: **if** DesiredResultsAchieved() **then**
14: **break** ▷ Exit loop if target fault coverage and user constraints are met
15: **end if**
16: SummarizeResults()
17: ApplyRulesAndGenerateFeedback()
18: iteration_count = iteration_count + 1
19: **end while**
20: **end procedure**

the area and power cost of additional test-related standard cells; and α is a technology-dependent scaling factor. Test time overhead is approximated as $N_{\text{patterns}} \cdot \lceil N_{\text{seq}}/N_{\text{chains}} \rceil \cdot T_{\text{clock}}$, which serves as a proxy for the actual test time, where N_{patterns} is the number of ATPG patterns and T_{clock} is the test clock period.

TABLE I: Structural-test benchmark circuit descriptions.

Design	Description
MIPS_16	16-bit implementation of a MIPS processor, an example of RISC instruction set architecture.
SASC	Serialized implementation of the AES encryption core.
MD5	MD5 hashing algorithm. Generates a 256-bit hash for secure data authentication.
WB_DMA	Wishbone Direct Memory Access controller. Transfers data between peripherals and memory.
AES_192	AES encryption using a 192-bit key. Provides secure data encryption and decryption.
MLP_MODEL	Hardware implementation of a Multi-Layer Perceptron.

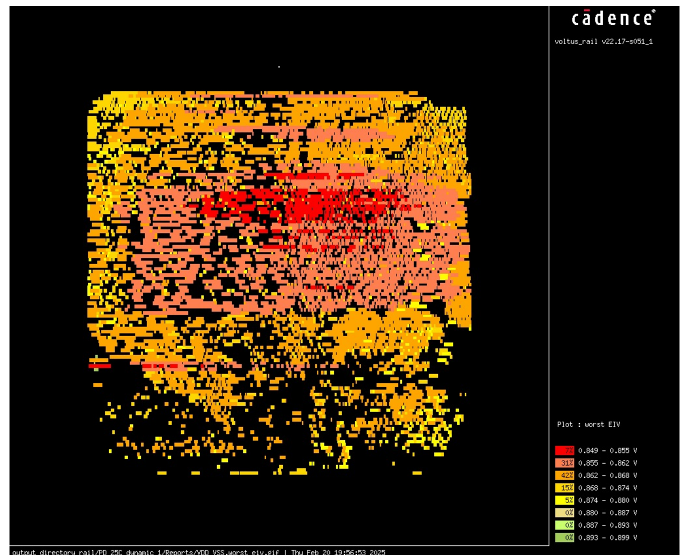

Fig. 3: Voltage droop distribution across the layout.

B. Performance Assessment of the Proposed Flow

1) Functional Testing: For functional testing, the proposed LLM-driven framework outperforms conventional benchmark-based workloads in inducing dynamic voltage stress that can lead to runtime execution errors (e.g., SDC-like effects). Using 10 representative programs from CoreMark, MiBench, and Embench, we observe that the strongest baseline workloads induce at most 6.88% voltage droop relative to V_{DD}, with `crc32` producing the highest maximum droop in the Arithmetic Logic Unit (ALU) (61.94 mV) and `aes` in the Load-Store Unit (LSU) (55.38 mV); these values remain below the commonly used critical threshold of 10% droop relative to V_{DD}. The LLM-based iterative optimization increases voltage droops beyond the benchmark baseline, reaching 72.25 mV in the ALU and 62.501 mV in the LSU, while RL-based prompt optimization achieves the highest stress, reaching 81.55 mV in the ALU and 70.72 mV in the LSU.

2) Structural Testing: For structural testing, the proposed flow targets maximum stuck-at fault coverage while enforcing constraints on area, power, and test time overheads. To derive realistic constraint bounds, multiple runs are performed for each benchmark circuit using a single scan chain and varying numbers of test points, with the resulting area and power overheads as well as the total test time calculated for each run. Table II summarizes the structural testing results of the proposed framework and compares them against a manually optimized baseline. Note that in the LLM + DSE flow, a design space exploration step is required to characterize the circuit and determine suitable configuration parameters prior to ATPG script generation. In contrast, the LLM + TPE flow tightly integrates model-based optimization with LLM-driven reasoning. Rather than performing explicit design space exploration, TPE constructs probabilistic density models from previously evaluated configurations to guide the search toward promising regions of the parameter space. These models capture the distributions of configurations associated with high-performing and low-performing objective values, enabling the optimizer to prioritize configurations that are more likely to yield improvement in fault coverage while satisfying user-defined constraints. In this context, TPE proposes candidate configurations, while the LLM interprets the resulting optimization feedback, such as achieved fault coverage and constraint satisfaction, to synthesize ATPG scripts for subsequent iterations.

The results demonstrate that the LLM-based solution achieves comparable performance and, in some cases, lower overhead, indicating the successful automation of the ATPG script generation workflow. The analysis of the individual optimization iterations indicates that the LLM, together with the KG, can reason about unmet constraints and apply appropriate corrective actions in subsequent iterations. For instance, when the test time constraint is violated in a given iteration, the LLM identifies the violation and increases the number of scan chains in the following iteration to reduce the overall test time.

V. Conclusion

This paper presented an agentic LLM-based framework for functional and structural testing that integrates iterative refinement with RL- and TPE-guided optimization. Experimental results show that the proposed approach can generate stress-inducing functional workloads and high-quality structural tests that are comparable to expert-crafted solutions, while substantially reducing manual effort. By integrating feedback-driven optimization, design space exploration, and structured reasoning, the framework enables efficient test generation under practical area, power, and test time constraints. Overall, these results demonstrate the potential of agentic AI and LLMs to automate and scale testing workflows for increasingly complex SoC designs.

References

[1] D. Sahoo, E. Ortega, P. Domanski, F. Firouzi, and K. Chakrabarty, "Tide: Telemetry-informed delay testing for silent data corruption," in *2025 IEEE International Test Conference (ITC)*. IEEE, 2025, pp. 514–517.

TABLE II: Structural testing results, comparing fault coverage, runtime, and overhead metrics under user-defined constraints.

Design	Constraints			Number of Simulated Patterns	Fault Coverage	Area Overhead	Power Overhead	Test Time Overhead
	Test Time	Area	Power					
Manual Baseline								
MIPS_16	5 ms	15%	15%	500	86.91%	11.4%	14.1%	4.38 ms
SASC	0.01 ms	25%	500%	47	83.04%	23.33%	451.3%	4.9 μs
MD5	0.1 ms	4%	175%	481	95.02%	3.17%	163.7%	82 μs
WB_DMA	0.02 ms	0.02%	150%	146	12.14%	2.37%	149.3%	49.1 μs
AES_192	2 ms	5%	15%	342	98.82%	3.01%	7.9%	1.65 ms
MLP_MODEL	2.5×10^{-4} ms	200%	15%	10	61.22%	156.03%	9780%	5.00×10^{-5} ms
LLM + DSE Flow								
MIPS_16	5 ms	15%	15%	796	86.69%	11.32%	14.7%	3.49 ms
SASC	0.01 ms	25%	500%	40	83.13%	24.61%	499.8%	4.24 μs
MD5	0.1 ms	4%	175%	464	95.02%	3.17%	163.7%	62.1 μs
WB_DMA	0.02 ms	0.02%	150%	100	12.79%	2.32%	144.1%	48 μs
AES_192	2 ms	5%	15%	343	98.82%	3.01%	10.5%	1.65 ms
MLP_MODEL	2.5×10^{-4} ms	200%	15%	12	63.43%	381.39%	9780%	4.40×10^{-5} ms
LLM + TPE Flow								
MIPS_16	5 ms	15%	15%	797	86.7%	11.1%	5.25%	6.9 ms
SASC	0.01 ms	25%	500%	47	83.04%	23.33%	454.69%	4.98 μs
MD5	0.1 ms	4%	175%	471	94.0%	2.24%	76.63%	60 μs
WB_DMA	0.02 ms	0.02%	150%	146	12.6%	2.13%	108.24%	32 μs
AES_192	2 ms	5%	15%	342	98.82%	3.01%	10.52%	1.65 ms
MLP_MODEL	2.5×10^{-4} ms	200%	15%	9	62.0%	231.16%	9779.13%	3.60×10^{-5} ms

[2] P. Domanski, D. Sahoo, E. Ortega, F. Firouzi, and K. Chakrabarty, "LLM-aided in-field workload generation for detecting silent data corruptions at scale," in *2025 IEEE International Test Conference (ITC)*. IEEE, 2025, pp. 121–130.

[3] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," New York, NY, USA, 2017.

[4] R. Baranowski, F. Firouzi, S. Kiamehr, C. Liu, M. Tahoori, and H.-J. Wunderlich, "On-line prediction of nbti-induced aging rates," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 589–592.

[5] F. Firouzi, S. Kiamehr, M. Tahoori, and S. Nassif, "Incorporating the impacts of workload-dependent runtime variations into timing analysis," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 1022–1025.

[6] S. Kiamehr, F. Firouzi, and M. B. Tahoori, "Input and transistor reordering for nbti and hci reduction in complex cmos gates," in *Proceedings of the great lakes symposium on VLSI*, 2012, pp. 201–206.

[7] Y. Hara-Azumi, F. Firouzi, S. Kiamehr, and M. Tahoori, "Instruction-set extension under process variation and aging effects," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 182–187.

[8] F. Firouzi, F. Ye, K. Chakrabarty, and M. B. Tahoori, "Chip health monitoring using machine learning," in *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2014, pp. 280–283.

[9] F. Firouzi, F. Ye, A. Vijayan, A. Koneru, K. Chakrabarty, and M. B. Tahoori, "Re-using bist for circuit aging monitoring," in *2015 20th IEEE European Test Symposium (ETS)*. IEEE, 2015, pp. 1–2.

[10] F. Firouzi, B. Farahani, P. Vergos, D. Sahoo, N. Bleier, and K. Chakrabarty, "Prompt, fab, flex: Agentic LLMs for flexible electronics design," in *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2025, pp. 1–9.

[11] J. Talukdar, A. Seth, S. Banerjee, F. Firouzi, and K. Chakrabarty, "Malls: Multi-agent llms for synthetic hardware vulnerability generation and detection," in *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, 2025, pp. 782–789.

[12] J. Talukdar, S. Banerjee, and F. Firouzi, "Llm-sec: Closing the security loop using llms for hardware design," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2025, pp. 1–6.

[13] S. Banerjee, J. Talukdar, and F. Firouzi, "Silicon whisperers: Improving test quality and cost in the age of generative ai," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2025, pp. 1–5.

[14] F. Firouzi *et al.*, "Chipmnd: Llms for agile chip design," in *2025 IEEE 43rd VLSI Test Symposium (VTS)*. IEEE, 2025, pp. 1–10.

[15] F. Firouzi, S. S. R. Nakkilla, C. Fu, S. Banerjee, J. Talukdar, and K. Chakrabarty, "Llm-aid: Leveraging large language models for rapid domain-specific accelerator development," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[16] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[17] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[18] S. Thakur *et al.*, "Benchmarking large language models for automated verilog rtl code generation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[19] M. DeLorenzo, A. B. Chowdhury, V. Gohil, S. Thakur, R. Karri, S. Garg, and J. Rajendran, "Make every move count: Llm-based high-quality rtl code generation using mcts," *arXiv preprint arXiv:2402.03289*, 2024.

[20] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model (2023)," *arXiv preprint arXiv:2308.05345*, 2023.

[21] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.

[22] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[23] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, “Llm4dv: Using large language models for hardware test stimuli generation,” *arXiv preprint arXiv:2310.04535*, 2023.

[24] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, “Chateda: A large language model powered autonomous agent for eda,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[25] R. Zhong *et al.*, “Llm4eda: Emerging progress in large language models for electronic design automation,” *arXiv preprint arXiv:2401.12224*, 2023.

[26] K. Chang *et al.*, “Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework,” in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[27] J. Bhandari, R. Sadhukhan, P. Krishnamurthy, F. Khorrami, and R. Karri, “Sentaur: Security enhanced trojan assessment using llms against undesirable revisions,” *arXiv preprint arXiv:2407.12352*, 2024.

[28] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, “Generating secure hardware using chatgpt resistant to cwes,” *Cryptology ePrint Archive*, 2023.

[29] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, “Llm-assisted generation of hardware assertions,” *arXiv preprint arXiv:2306.14027*, 2023.

[30] J. Chaudhuri, D. Thapar, A. Chaudhuri, F. Firouzi, and K. Chakrabarty, “Spiced: Syntactical bug and trojan pattern identification in a/ms circuits using llm-enhanced detection,” in *IEEE Physical Assurance and Inspection of Electronics (PAINE)*, 2024, pp. 1–7.

[31] ——, “Spiced+: Syntactical bug pattern identification and correction of trojans in a/ms circuits using llm-enhanced detection,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2025.

[32] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” in *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53.

[33] X. Hou *et al.*, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[34] M. Liu, N. Pinckney, B. Khailany, and H. Ren, “Verilogeval: Evaluating large language models for verilog code generation,” in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–8.

[35] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, “Verigen: A large language model for verilog code generation,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[36] P. Vijayaraghavan, L. Shi, S. Ambrogio, C. Mackin, A. Nitsure, D. Beymer, and E. Degan, “Vhdl-eval: A framework for evaluating large language models in vhdl code generation,” *arXiv preprint arXiv:2406.04379*, 2024.

[37] P. Vijayaraghavan *et al.*, “Chain-of-descriptions: Improving code llms for vhdl code generation and summarization,” in *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–10.

[38] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, “Silifuzz: Fuzzing cpus by proxy,” *arXiv preprint arXiv:2110.11519*, 2021.

[39] D. Schwachhofer, P. Domanski, S. Becker, S. Wagner, M. Sauer, D. Pflüger, and I. Polian, “Training large language models for system-level test program generation targeting non-functional properties,” in *IEEE European Test Symposium (ETS)*, 2024, pp. 1–4.

[40] ——, “Large language model-based optimization for system-level test program generation,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2024, pp. 1–6.

[41] N. Karystinos, O. Chatzopoulos, G.-M. Fragkoulis, G. Papadimitriou, D. Gizopoulos, and S. Gurumurthi, “Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation,” in *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 516–531.

[42] V. J. Reddi, M. S. Gupta, G. Holloway, G.-Y. Wei, M. D. Smith, and D. Brooks, “Voltage emergency prediction: Using signatures to reduce operating margins,” in *IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 18–29.