VerilogReader: LLM-Aided Hardware Test Generation

Ruiyang Ma[1], Yuxin Yang[1], Ziqian Liu[2], Jiaxi Zhang[1], Min Li[3], Junhua Huang[3], Guojie Luo[1]
[1]*School of Computer Science, Peking University;* [2]*School of Information, Renmin University of China;* [3]*Noah's Ark Lab, Huawei*
ruiyang@stu.pku.edu.cn, {yxyang, zhangjiaxi, gluo}@pku.edu.cn, liuziqian@ruc.edu.cn, minli.amoy@gmail.com, huang.hjh@outlook.com

2024 IEEE LLM Aided Design Workshop (LAD) | 979-8-3503-7608-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/LAD62341.2024.10691801

***Abstract*—Test generation has been a critical and labor-intensive process in hardware design verification. Recently, the emergence of Large Language Model (LLM) with their advanced understanding and inference capabilities, has introduced a novel approach. In this work, we investigate the integration of LLM into the Coverage Directed Test Generation (CDG) process, where the LLM functions as a Verilog Reader. It accurately grasps the code logic, thereby generating stimuli that can reach unexplored code branches. We compare our framework with random testing, using our self-designed Verilog benchmark suite. Experiments demonstrate that our framework outperforms random testing on designs within the LLM's comprehension scope. Our work also proposes prompt engineering optimizations to augment LLM's understanding scope and accuracy.**

***Index Terms*—Automatic Test Generation, LLM, Verilog**

I. Introduction

As hardware complexity surges, the importance of hardware verification in the development process intensifies. Undetected hardware bugs can result in substantial repercussions and considerable economic losses. To address the risk of design flaws in hardware, engineers employ two primary verification methodologies: *formal verification* and *dynamic verification*.

Formal methods employs mathematical techniques to prove or disprove the correctness of a system with respect to a certain formal specification or property [1]. On the other hand, dynamic verification, generates diverse test cases to simulate the Design Under Test (DUT), offering more flexibility and scalability than formal verification [2]. Coverage targets, including code and functional coverage, serve as benchmarks for determining the thoroughness of tests. The attainment of these targets necessitates high-quality test inputs, which imposes a considerable labor burden on verification engineers.

To reduce the need for human intervention, Coverage Directed Test Generation (CDG) has emerged as a pivotal technique in automatic hardware test generation [3]–[7]. This method leverages heuristic approaches to explore the input space, with coverage states serving as basic feedback for the generation of new test cases. In situations with hard-to-reach coverpoints, supplementary circuit structural information (*e.g.,* control/data flow graph, module connectivity graph) are utilized to guide directed test generation [4], [7], [8].

Recently, the impressive capabilities of LLM in comprehension and inference have been highlighted. Previous studies have shown LLM's versatility in multiple hardware tasks, such as RTL writing [9], [10], assertion generation [11], [12] and bug fixing [13]. The advanced competencies of LLM present a compelling opportunity for their deployment in the field of hardware test generation. Zhang *et al.* have pioneered the initial step towards verifying the functional points of DUT [14]. A description of functional coverpoints is provided, following which the LLM generates input sequences. Their experimental results demonstrate a significant improvement in performance over random testing on various DUTs. Their research substantiates the capability of LLM to comprehend the high-level description of input principles and functional testpoints in the task of hardware verification.

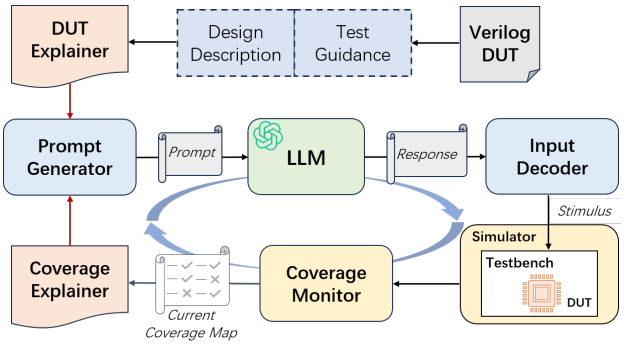

Fig. 1: LLM-Aided Hardware Test Generation Workflow.

While our research adopts a distinct perspective. Complementing with previous work, we have pioneered the use of LLM to specifically improve the hardware code coverage, which is a more fundamental testing target and is intrinsically linked to the Verilog code itself. This approach necessitates the shift in LLM's focus from the high-level functional testplan descriptions to the in-depth understanding of basic Verilog code logic and coverage status. That is, we repositioned the LLM as a *VerilogReader*, facilitating its role as a hardware verifier to read codes and write test cases for uncovered lines or branches, consequently reducing the manual effort required for code analysis and test generation.

In summary, our paper makes the following contributions:

- We open-source a framework that integrates LLM into the CDG process[1]. For the first time, LLM is used as a *VerilogReader* to understand Verilog code and coverage, aiming to generate tests for code coverage closure.
- We propose *Coverage Explainer* and *DUT Explainer* to enrich the prompt, thereby enhancing LLM's comprehension of the design and our testing intentions. These modules also augment the extensibility of our framework.
- We create a benchmark suite including 24 Verilog designs of simple, medium, and complex levels. Our experiments show that our framework outperforms random testing on simple- and medium-level DUTs. We also delineate the maximum Verilog reading capabilities of current LLM.

This work was partly supported by the National Natural Science Foundation of China (Grant No. 62090021) and the National Key R&D Program of China (Grant No. 2022YFB4500500). Corresponding author: Guojie Luo.

[1]https://github.com/magicYang1573/llm-hardware-test-generation

II. Approach

A. Basic Framework

Our study integrates LLM into the Coverage Directed Test Generation (CDG) process, as depicted in Figure 1. In each iteration, the LLM generates multi-cycle inputs in JSON format. These inputs are subsequently decoded by the *Input Decoder* into hardware stimuli. Upon completion of simulation, the *Coverage Monitor* provides current code coverage information to LLM, guiding the generation of extra input stimuli.

To generate test inputs, the LLM requires a comprehensive understanding of the Verilog DUT and the current coverage status. Given that these data are initially in non-natural-language formats, they must be transformed into a format conducive to the LLM. To this end, we have introduced two explainer modules. The *Coverage Explainer* module reformats the original simulator coverage report into a more LLM-readable format, while the *DUT Explainer* module enriches the DUT code with a natural language description or guidance. These modules collectively enhances the LLM's comprehension of test intentions and the DUT's functionality. Following this, the *Prompt Generator* integrates these outputs to create the final prompt.

B. Prompt Generator

To encourage a step-by-step thought process in the LLM, the *Prompt Generator* facilitates two rounds of question-and-answer sessions in each iteration of the CDG process, thereby generating the hardware input stimulus, as depicted in Figure 2. In the first round, the LLM is informed of our objective to generate tests for unexplored code lines, incorporating details about the DUT from the *DUT Explainer* and the current coverage data from the *Coverage Explainer*. The LLM responds in natural language, typically mirroring its cognitive process. In the second round, we instruct the LLM to reformulate its initial response into a standardized JSON format for subsequent input decoding.

C. Coverage Explainer

To enhance LLM's comprehension of the current DUT coverage, we introduce the *Coverage Explainer* module, which translates the intricate coverage report into a more comprehensible format. As shown in Figure 3(a), the original Verilator coverage report format includes each coverpoint represented by a unique identifier string and a hit count. This format is cryptic and poses readability challenges for both human users and LLM.

A simple alternative involves using Verilator-provided `verilator_coverage` tool to create an annotated coverage report, as depicted in Figure 3(b). This format, which correlates coverage status with DUT source code, is more interpretable. The left-side number in each code line indicates the hit count of the line.

Despite the improvements of annotated coverage format, it still presents challenges for LLM, as LLM must identify uncovered lines, thereby increasing the complexity. To mitigate this, we suggest an advanced LLM-readable coverage report,

Task Prompt 1:
Your task involves a Verilog DUT that is in initial stage of testing. The assignment requires you to generate a binary input sequence to cover these uncovered lines with mark *'TO BE COVERED'*.

DUT and Coverage Prompt:

```
// DUT Description: 2-to-1 multiplexer, selects between a / b based on sel
//Test Guidance: test all possible states of sel signal
module top (input a, input b, input sel, output reg out);
    always@(*) begin
        if (sel) out = a;
        else out = b;      //TO BE COVERED
    end
endmodule
```

Answer 1:
The line that is marked 'TO BE COVERED' is the line where the output ***out*** equals ***b***. This line is executed when the ***sel*** input is zero. The value of ***a*** and ***b*** does not matter in this case.

Task Prompt 2:
DUT has the following input signals:
Input Signal: a, b, sel; Width: 1 bit
Please translate your answer with the following Json format:
{a: x, b: x, sel: x}

Answer 2:
{a: 0, b: 0, sel: 0}

Fig. 2: Example of prompts and LLM answers.

```
C 'ftop.vl3n13pagev_branch/topoifS3hTOP.top' 5
C 'ftop.vl3n14pagev_branch/topoelseS4hTOP.top' 0
```

(a) Verilator original coverage report

```
    module top (input a, input b, input sel, output reg out);
      always@(*) begin
000005    if (sel) out = a;
000000    else out = b;
      end
    endmodule
```

(b) Verilator annotated coverage report

```
    module top (input a, input b, input sel, output reg out);
      always@(*) begin
        if (sel) out = a;
        else out = b;       // TO BE COVERED
      end
    endmodule
```

(c) LLM-readable coverage report

Fig. 3: Comparison of three coverage report formats.

specifically designed for our test generation task, as depicted in Figure 3(c). This report introduces a *'TO BE COVERED'* flag for lines that remain uncovered. The application of natural language to flag only the uncovered lines could facilitate a more straightforward inference process for LLM.

D. DUT Explainer

To augment LLM's comprehension of the DUT, we introduce the *DUT Explainer* module. Given that the LLM's comprehension of Verilog code for test generation tasks is not fully optimized, this module aims to provide additional digestible information about the DUT, thereby facilitating

more efficient test generation. The DUT Explainer module is designed to serve two main functions.

Design Description, provides the LLM with a natural language explanation of the DUT's functionalities and internal logic, mitigating the LLM's incapacity to interpret Verilog code. This description can be acquired either by the LLM or manually. When acquired by the LLM, the test generation task is split into two stages: DUT understanding and input logic inference, thus alleviating LLM's workload in each phase.

Test Guidance, enriches the LLM with supplementary information for creating tests for specific DUT. This could involve fundamental test logic rules or advice for some hard-to-cover points. For instance, when generating tests for a Finite State Machine (FSM) circuit, LLM is guided to first consider the transition to each state and then discern conditions to address any uncovered points within that state. Additionally, it could be endowed with some knowledge on reaching challenging states, thereby reducing the analytical burden on the LLM.

III. Evaluation

We evaluate our framework on our synthetic benchmark suite, detailed in Section III-A. For each design, we use Pyverilog [15] to extract input signals and automatically generate testbench interface with our framework. Verilator [16] serves as our simulator. The language models used in our experiments include OpenAI's GPT-4 and GPT-4-Turbo-0125 [17].

A. Benchmark Suite

We created 24 Verilog designs in our benchmark suite and assigns three difficulty levels for these designs.

1) Simple: This level involves 10 basic combinational logic circuits (`s01-s10`), such as multiplexer and ALU. The direct influence of inputs on the coverage path within the same cycle offers a straightforward inference scenario for LLM. These designs are used to assess LLM's understanding of Verilog syntax, including constructs like `always`, `case`, `assign`, *etc.*

2) Medium: This level consists of 8 sequential logic circuits (`m01-m08`), such as FSMs, counters and arbiters. The coverage path of the current cycle is influenced by inputs from several preceding cycles. These designs aim to demonstrate LLM's cross-cycle inference capabilities in test generation tasks.

3) Complex: This level encompasses 6 large-scale FSM circuits (`c01-c06`), ranging from 16 to 128 states, and two transition branches per state. It serves as a benchmark category to evaluate the upper limit of the current LLM's comprehensive ability in hardware test generation tasks.

B. Comparison of Coverage Explanations

In Section II-C, we present an LLM-readable coverage report, designed to enhance LLM's comprehension of current coverage status. To validate the utility of our coverage explanation method, we contrast it with the original and annotated coverage reports from Verilator.

The experiments were carried out on medium-level DUTs using GPT-4 as the language model. The comparison metric was the total length of input stimulus (measured in clock cycles) required to achieve full line coverage. Given the stochastic behavior of LLM, each experiment was replicated five times. The results are represented as box (25%ile) and whisker (75%ile) plots, along with median lines for each DUT, as shown in Figure 4. The figure clearly indicates that the original unreadable coverage report poses the greatest challenge for LLM, whereas our LLM-readable coverage report demonstrates superior performance compared to the other two Verilator-provided reports.

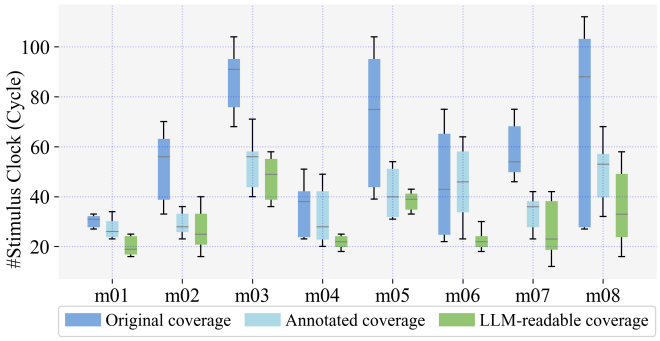

Fig. 4: Comparison of coverage explanations.

C. Comparison against Random Testing

In order to evaluate the efficacy of LLM for hardware test generation, we contrast our framework with random testing.

We conducted experiments on simple- and medium-level DUTs, utilizing GPT-4 and GPT-4-Turbo as language models. We also performed five trials for each experiment. As illustrated in Figure 5 (log scale), LLM achieved 100% coverage using significantly fewer inputs than random testing. The limitations of random testing became especially apparent in sequential designs with elusive branches, often failing to achieve full coverage within one-minute timeframe. In contrast, LLM could expediently reach these branches with their capacity for circuit logic analysis. Interestingly, despite GPT-4-Turbo's purported superiority, it demonstrated a similar capability to GPT-4 in hardware test generation tasks in our experiments.

D. DUT Explanation Optimization

In Section II-D, we introduce two optimization methods in *DUT Explainer* module that aim to improve LLM's understanding of hardware design. Beyond providing LLM with the original Verilog code, we can supplement this with *Design Description* or *Test Guidance*. The former is generated by GPT-4 in our experiment, while the latter is manually written. These resources can be accessed in our open-source project.

We carried out experiments on medium-level DUTs using GPT-4, with each experiment conducted five times. The results, presented in Figure 6, indicate that the inclusion of a LLM-generated *Design Description* in the prompt improved LLM's understanding of the design during test generation. However, the impact of *Test Guidance* was not uniformly beneficial. In designs like m05 and m06, the guidance inadvertently reduced the diversity of LLM-generated input, causing an over-reliance on our guidance and consequently stifling its capacity for self-exploration.

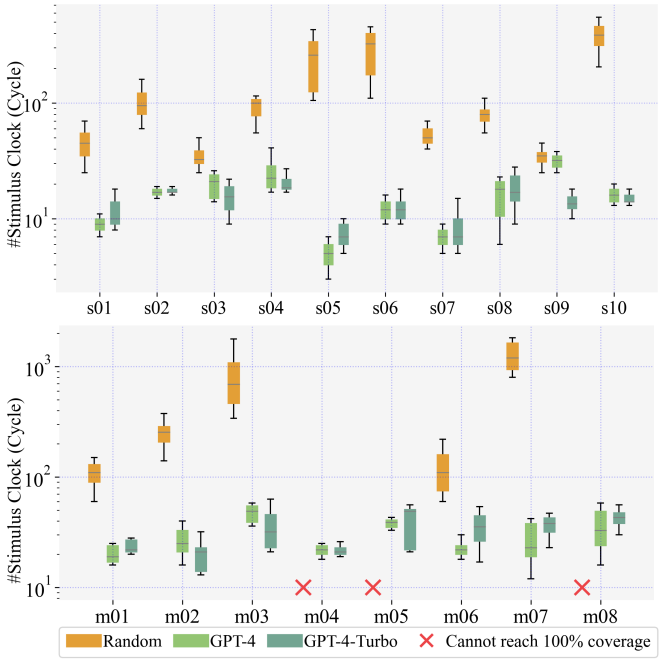

Fig. 5: Comparison of LLM-aided test generation and random testing.

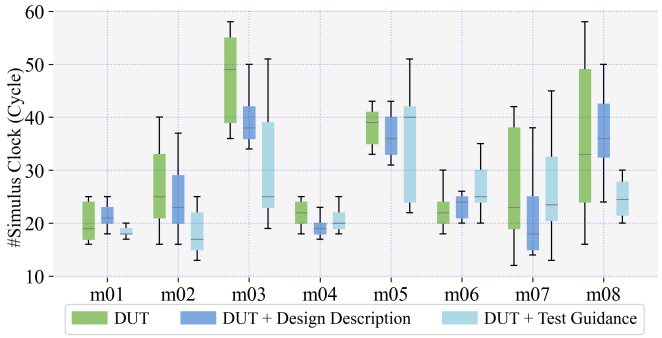

Fig. 6: Effect of design description and test guidance.

E. LLM Reading Scalability

In our previous experiment, we evaluated LLM's proficiency in generating tests for simple- and medium-level hardware designs, with the most complex designs consisting of around 100 lines of Verilog code. To explore the upper bounds of current LLM's capabilities for test generation, we introduced a complex level in our benchmark and employed FSMs with varying numbers of states as DUTs. Given that the largest design exceeded 500 lines of code and surpassed GPT-4's input length limitation, we chose GPT-4-Turbo for this experiment, conducting three trials and calculating the average.

Figure 7 illustrates the outcome of the experiment. It is evident that as the DUT scalability escalates, the quality of test generation precipitously declines. For an FSM with 16 states, nearly 100% line coverage was achieved after 20 iterations of LLM calls. However, for larger FSM designs with over 64 states, the coverage cannot exceed 50%. This reveals the LLM's inadequacies in directly processing large Verilog designs and performing intricate inferences for test generation.

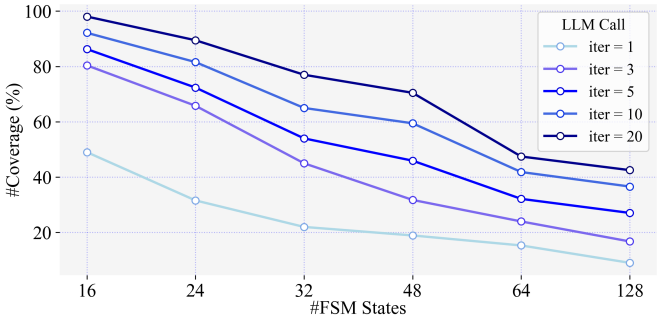

Fig. 7: LLM's performance in test generation for large-scale FSMs.

IV. Discussion

While LLM demonstrates competence in understanding simple- and medium-level DUTs, their performance diminishes with complex-level benchmarks and industry-scale hardware designs. The aspiration to employ LLM in an end-to-end manner for such designs is challenging. A substantial journey lies ahead before LLM can surpass a human hardware expert, especially in the context of Verilog code comprehension and its subsequent application in diverse EDA tasks.

One potential solution to enable the application of LLM in real-world hardware verification is to enhance our *DUT Explainer*. By providing a more comprehensive high-level abstraction of the design and the verification intentions, we can guide LLM to view test generation tasks from a more macroscopic perspective. Our LLM-aided framework offers the opportunity for users to seamlessly incorporate help information during the hardware CDG process. LLM could facilitate the translation of these guidance information from natural language into actual hardware stimuli, thereby alleviating the workload of hardware verification engineers.

Furthermore, future research could focus on merging LLM with other structural AI techniques. Verilog's highly structured nature, characterized by a multitude of concurrent always blocks and module hierarchies, presents a significant challenge for LLM's decipherment. However, these structures may be more easily understood by a Graph Neural Network (GNN) [8], [18], [19]. Therefore, the combination of LLM for regional semantic interpretation and GNN for structural interpretation could present a promising strategy to enhance the scalability of AI hardware understanding capabilities.

V. Conclusion

Our research primarily investigates the application of LLM in understanding Verilog designs and generating test inputs to achieve code coverage closure. We have constructed a suite of benchmarks comprising basic combinational and sequential circuits to assess our framework's efficacy. To enhance LLM's comprehension of a given DUT and the test generation task, we have introduced Coverage Explainer and DUT Explainer to enrich the prompt. Experimental results demonstrate that the LLM is capable of generating inputs and achieves full code coverage for DUTs of simple and medium complexity in our benchmarks. Future research could focus on enhancing the abstraction level of guidance information provided to LLM, or integrating LLM with GNN to capture both semantic and structural information of DUTs.

REFERENCES

[1] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *TODAES*, vol. 4, no. 2, pp. 123–193, 1999.

[2] W. K. Lam, *Hardware design verification: simulation and formal method-based approaches*. Prentice Hall PTR, 2005.

[3] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *DAC*, Jun 2003, pp. 286–291.

[4] M. Li, K. Gent, and M. S. Hsiao, "Design validation of RTL circuits using evolutionary swarm intelligence," in *ITC*, Nov 2012, pp. 1–8.

[5] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, "Accelerating coverage directed test generation for functional verification: A neural network-based framework," in *GLSVLSI*, May 2018, pp. 207–212.

[6] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *ICCAD*, 2018, pp. 1–8.

[7] S. Canakci, L. Delshadtehrani, and F. Eris, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *DAC*, 2021, pp. 529–534.

[8] S. Vasudevan, W. J. Jiang, D. Bieber, R. Singh, C. R. Ho, C. Sutton *et al.*, "Learning semantic representations to verify hardware designs," in *NeurIPS*, vol. 34, 2021, pp. 23 491–23 504.

[9] 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.

[10] 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," *TODAES*, vol. 29, no. 3, pp. 1–31, 2023.

[11] 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.

[12] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "AssertLLM: Generating and evaluating hardware verification assertions from design specifications via multi-LLMs," *arXiv preprint arXiv:2402.00386*, 2024.

[13] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[14] 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.

[15] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL," in *ARC*. Springer International Publishing, Apr 2015, pp. 451–460.

[16] W. Snyder, "Verilator," https://www.veripool.org/wiki/verilator.

[17] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[18] Z. Shi, H. Pan, S. Khan, M. Li, Y. Liu, J. Huang, H.-L. Zhen, M. Yuan, Z. Chu, and Q. Xu, "Deepgate2: Functionality-aware circuit representation learning," in *ICCAD*, 2023, pp. 1–9.

[19] B. Jin, G. Liu, C. Han, M. Jiang, H. Ji, and J. Han, "Large language models on graphs: A comprehensive survey," *arXiv preprint arXiv:2312.02783*, 2023.