HLSDebugger: Identification and Correction of Logic Bugs in HLS Code with LLM Solutions

Jing Wang, Shang Liu, Yao Lu, Zhiyao Xie*

Hong Kong University of Science and Technology

{jwangjw, sliudx, yludf}@connect.ust.hk, eezhiyao@ust.hk

Abstract—High-level synthesis (HLS) accelerates hardware design by enabling the automatic translation of high-level descriptions into efficient hardware implementations. However, debugging HLS code is a challenging and labor-intensive task, especially for novice circuit designers or software engineers without sufficient hardware domain knowledge. The recent emergence of Large Language Models (LLMs) is promising in automating the HLS debugging process. Despite the great potential, three key challenges persist when applying LLMs to HLS logic debugging: 1) High-quality circuit data for training LLMs is scarce, posing a significant challenge. 2) Debugging logic bugs in hardware is inherently more complex than identifying software bugs with existing golden test cases. 3) The absence of reliable test cases requires multi-tasking solutions, performing both bug identification and correction. In this work, we propose a customized solution named HLSDebugger[1], to address the challenges. HLSDebugger first generates and releases a large labeled dataset with 300K data samples, targeting HLS logic bugs. The HLSDebugger model adopts an encoder-decoder structure, performing bug location identification, bug type prediction, and bug correction with the same model. HLSDebugger significantly outperforms advanced LLMs like GPT-4 in bug identification and by more than $3\times$ in bug correction. It makes a substantial advancement in the exploration of automated debugging of HLS code.

Index Terms—high-level synthesis, functional verification, bug identification, large language model

I. INTRODUCTION

High-Level Synthesis (HLS) has revolutionized the hardware design process by allowing designers to define hardware functionality using high-level programming languages, such as C++ or SystemC. Such a high-level abstraction of circuits accelerates the design process and thus enables rapid prototyping and agile development of hardware. However, as with any development process, debugging is a critical component, which is labor-intensive and requires knowledge in both software and hardware domains. In real-world situations, many pre-silicon logic bugs will be inadvertently introduced by hardware designers when crafting the HLS version of a design. Such logic bugs can elude HLS and static analysis tools (e.g., Infer) [1], causing unintended functionalities.

In recent years, the emergence of Large Language Models (LLMs) has provided unprecedented opportunities for automating code debugging tasks. Despite extensive debugging explorations in the software domain [2]–[7], LLM solutions are almost unexplored in HLS, one of the most widely adopted agile design methodologies in the hardware domain. We believe LLM-assisted HLS debugging will help further boost the hardware design efficiency and narrow the productivity gap between software and hardware domains. Targeting LLM-assisted HLS debugging, particularly for *logic bug* identification and correction, we have identified three key challenges that need to be addressed.
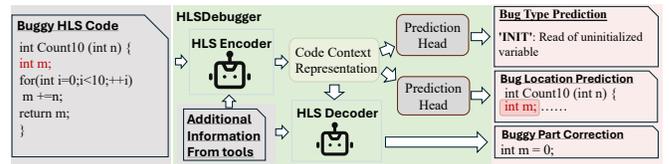
Challenge 1. Circuit Data Scarcity. The proprietary nature of circuit data and its intrinsic value in the semiconductor industry have
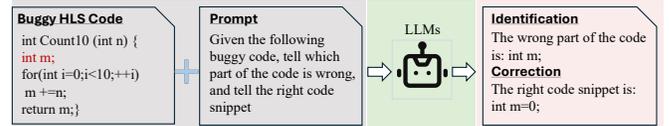
(a) Traditional LLM-based solutions for debugging tasks generate bug identification and correction sequentially in the output text.

(b) Our customized bug identification solution, HLSDebugger, directly generates bug location and provides bug corrections without requiring a carefully crafted prompt. The HLSDebugger can also utilize information (e.g, Bug Location Indication) from HLS tools to assist the bug correction process.

Fig. 1: Comparison between traditional LLM solutions on debugging tasks and our customized debugging structure on HLS code. Our solution in (b) is faster, more accurate, and offers improved performance robustness.

caused a long-lasting barrier to circuit data availability. As a result, collecting sufficient designs in HLS code format is exceptionally difficult, thus preventing the training of LLM solutions for HLS debugging. The work of [1] introduces a benchmark for LLM-based identification of logic bugs in HLS. Targeting the data scarcity challenge, in this work, we generate and release a $25\times$ larger-scale dataset with around 300K samples to support LLM training. This is a labeled dataset that supports supervised fine-tuning of LLMs, with each data sample including buggy code, corrected code, and detailed bug information.

Challenge 2. Debugging Logic Bugs. Many prior works on hardware and software debugging [2], [5], [6], [8], [9] focused on syntax bugs, which are easier to detect and correct based on corresponding compilers. However, logic bugs, where the code functionality diverges from the desired behavior, are much more challenging to handle. Compilers are not guaranteed to detect most logic bugs, necessitating more sophisticated functionality-aware methods to capture logic bug information. In this work, we target 8 types of known logic bugs that commonly exist in problematic HLS code, as suggested in the work of [1]. Their detailed information will be listed in Section V-A.

In the HLS domain, crafting reliable test cases for every HLS code sample has been labor-intensive, requiring significant domain expertise. The growing complexity of modern hardware makes hand-crafting enough test cases increasingly challenging, especially at such an early design stage. In this work, we focus on logic bugs of HLS code, without assuming any additional input such as test cases. As a result, both the identification and correction of bugs by LLMs are required, as we will introduce in the next subsection.

---

*Corresponding Author.

[1]Open-sourced at https://github.com/hkust-zhiyao/HLSDebugger

Works	Domain	Logic Error	New Training Dataset	New Model	Not Require Test Cases	Multi-task		
						Bug Location	Bug Correction	Bug Info
RTLFixer [8]	RTL						✓	
HDLdebugger [9]			Closed-source	Closed-source			✓	✓
LLM4SecHW [10]		✓	Closed-source	Closed-source	✓	✓	✓	✓
FixAgent [4]	Software	✓			✓	✓	✓	✓
D4C [5]		✓					✓	
AgentFL [3]		✓				✓		
ChatDBG [6]		✓					✓	✓
CREF [2]		✓					✓	✓
LDB [7]		✓					✓	✓
HLSDebugger	HLS	✓	Open-source	Open-source	✓	✓	✓	✓

TABLE I: Comparison between HLSDebugger and related LLM-based *debugging* works, all in other domains.

Challenge 3. Multi-Tasking in HLS Debugging. Debugging HLS code without additional information (e.g., without test cases) typically requires a locate-then-correct process, making this process inherently multi-tasking. But this two-step process is challenging and can lead to error accumulation [5]. For example, when asked to identify the bug in the given code in the first step, traditional generative LLMs might wrongly produce an unrelated code snippet that is actually not even found in the original code. Such mistakes will make correcting bugs in the second step impossible. In addition, the traditional generative objective of LLMs (i.e., making generated output exactly match the label) is not sufficient for such multi-tasking debugging tasks.

In this paper, we propose a new solution named HLSDebugger, which tackles the aforementioned challenges. To the best of our knowledge, HLSDebugger is the first fine-tuned LLM solution performing both the identification and correction of logic bugs in HLS code. HLSDebugger first generates and releases a large labeled dataset targeting HLS logic bugs. The HLSDebugger model adopts an encoder-decoder structure, performing bug location identification, bug type prediction, and bug correction with the same model.

Dataset Generation. We first address the circuit data gap by generating a large open-source dataset to train customized open-source LLMs on HLS debugging tasks. This data set contains around 300K data samples[2] (each buggy sample will be used as a training sample), which is approximately 25 times the size in [1]. It is also more than ten times larger than representative open-source hardware code datasets [11], [12] for other LLM-assisted hardware design tasks (e.g., RTL code generation). We generate the dataset by first collecting unlabeled data with mostly correct code, then subsequently inserting common bugs to generate corresponding buggy code.

Encoder-Decoder Debugging Structure. To better adapt the LLM structure for this multi-tasking debugging application, we propose to utilize the encoder-decoder LLM structure with a customized loss function. As Figure 1(b) summarizes, the LLM encoder performs bug identification, both pinpointing bug location within the original code context and predicting the bug type. The decoder then incorporates the intermediate representation from the encoder to generate the bug correction. Experiments demonstrate that integrating bug location identification and correction in a unified encoder-decoder structure boosts the model performance in both tasks.

Debugger Training Scheme. Furthermore, we propose a new explicit training scheme for the customized encoder-decoder structure. This training scheme combines both *bug identification loss* and *bug correction loss* to train both the encoder and decoder parts of HLSDebugger simultaneously.

Our contributions in HLSDebugger can be summarized as follows.

- To the best of our knowledge, HLSDebugger provides the first fine-tuned LLM solution for both identification and correction of logic bugs in HLS code.
- We generated and released an open-source, high-quality dataset for HLS debugging. Our dataset demonstrates effectiveness in improving existing open-sourced LLMs to outperform GPT-4 by more than $2\times$ in bug correction.
- We proposed a customized encoder-decoder structure for HLS logic bug identification and correction. After fine-tuned with our dataset, it significantly outperforms the commercial GPT-4 by more than $3\times$ in bug correction.
- We meticulously designed a customized training scheme for the encoder-decoder structure. This training scheme enables HLSDebugger's multi-tasking debugging process.

II. RELATED WORK

We introduce related LLM for debugging solutions, covering software, RTL, and HLS in Section II-A, II-B, and II-C, respectively. Table I summarizes these related debugging works.

A. LLMs for Software Debugging

In the software domain, benchmarks such as QuixBugs [13] and CodeFlows [14] have been proposed for assessing model debugging capabilities. Most software debugging works [2], [5]–[7] do not undertake explicit bug location identification tasks. Instead, they assume test cases (e.g., unit tests) are available for each potentially buggy software code. As a result, they can directly utilize the information from test cases as known bug indicators to guide bug correction. For similar reasons, most software debugging works do not strictly distinguish logic and syntax bugs.

In this work, we focus on a more challenging scenario where the logic bugs can not be identified directly by the compiler, and no test cases are provided [1]. LLMs are tasked with identifying and correcting bugs at the same time.

B. LLMs for RTL Debugging

In the hardware domain, most works focus on RTL instead of HLS code. The works of [8]–[10], [15], [16] target debugging of RTL. Works [8], [9] build robust databases to support LLM debugging using RAG, which needs error information from test cases. LM4SecHW [10] created a dataset from GitHub for fine-tuning LLMs by gathering commit information tagged with 'bug' and 'enhancement' from open-source hardware repositories. This dataset enhances LLM's capabilities in identifying and correcting

---

[2]Each data sample consists of a training instance containing a buggy code sample that begins with a module definition and concludes with the "end module" keyword, accompanied by corresponding bug indication and correction labels.
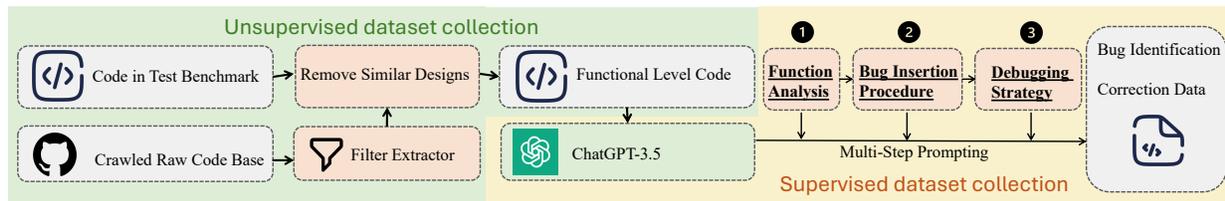
Fig. 2: Our proposed dataset generation flow in HLSDebugger. It consists of two parts: 1) unsupervised dataset collection and 2) supervised bug dataset generation. The second supervised dataset generation process consists of three sub-steps: ❶ **Function analysis**, ❷ **Bug insertion procedure**, ❸ **Debugging strategy analysis**.

function-related bugs, flaws, or possible improvements. However, such "version-control" data from GitHub does not ensure that a subsequent commit actually corrects bugs in prior code.

In our work, we are facing a more challenging scenario where the location of the bug is not available. We compared different LLMs in HLS debugging without providing location information and observed a significant drop in debugging performance. To address this challenge, we propose the HLSDebugger that simultaneously tackles both the location problem and the bug-fixing problem.

C. LLM for HLS Debugging & Generation

Compared to RTL and software development, there is a lack of mature debugging tools and methodologies tailored for HLS. HLS operates at a higher level of abstraction, which can hide details that are crucial for hardware design but are not immediately visible in the source code, making it harder to pinpoint issues related to hardware implementation. Even though there are works like C2HLSC [17] to explore HLS tasks related to code generation, LLMs' solution on HLS debugging tasks remains untouched.

Wan et al. [1] introduced an HLS dataset with around 12K data samples. This dataset is more appropriate for benchmarking various LLM solutions in HLS code debugging. This work does not contribute new models for HLS debugging. HLSPilot [18] targets HLS generation tasks, emphasizing performance bottlenecks and co-design for CPU and FPGA. It prompts GPT to create a framework but does not offer open-source models.

III. PROBLEM FORMULATION

We first formulate the HLS debugging problem. Given a potentially buggy code sample C_{buggy} as input, we denote the actual bug (i.e., buggy snippet) in the input code sample as s_{buggy}, with $s_{\text{buggy}} \subseteq C_{\text{buggy}}$. This debugging process includes two tasks: 1) identify the bug snippet s_{buggy} in the code sample C_{buggy}; 2) generate a corrected code sample C_{correct}. The expected outputs for each task during debugging are described below.

- **Bug Identification**. Assume there are altogether N tokens in the given code sample C_{buggy}. The label of bug identification can be represented as a binary list $\{I_1, I_2, ..., I_N\}$, where each I_i is a binary indicator denoting whether the ith token of C_{buggy} is part of the buggy snippet s_{buggy}. This can also be expressed as $s_{\text{buggy}} = \{C_{\text{buggy}}[I_i] \mid I_i == 1\}$.
- **Bug Correction.** After identifying the buggy tokens, the LLM needs to generate the corrected code sample by replacing the buggy snippet s_{buggy} in input code sample C_{buggy}. The correct code sample is denoted as C_{correct}. We denote the corresponding corrected bug snippet as s_{correct}, such that $s_{\text{correct}} \subseteq C_{\text{correct}}$.

Our goal is to develop an LLM solution, denoted as LLM, to maximize the accuracy of the process below:

$$LLM(C_{\text{buggy}}) \rightarrow \{I_1, I_2, ..., I_N\}, C_{\text{correct}}$$

IV. METHODOLOGY

This Section consists of three sub-sections corresponding to our major contributions: 1) Dataset generation; 2) Encoder-decoder structure; 3) Encoder-decoder training scheme.

A. Dataset Generation

To address the data scarcity problem for fine-tuning LLMs on HLS debugging, we proposed a dataset generation procedure and generated a large labeled dataset with approximately 300K data samples. This supervised dataset includes buggy HLS code sample C_{buggy} and detailed bug labels. The bug label provides information for both bug identification and correction: buggy snippet s_{buggy}, its bug type T, and corresponding correct code snippet s_{correct}.

As Figure 2 shows, the dataset generation process consists of two main parts: 1) Unsupervised dataset collection; and 2) Supervised bug dataset generation. We first collected an unsupervised dataset with correct HLS code from GitHub and GPT-3.5 [19] generations. Based on this initial unlabeled dataset, we generate the supervised bug dataset by inserting bugs to the original code through the CoT process by GPT-3.5 [19]. As introduced, the final supervised dataset provides buggy code C_{buggy}, correct code $C_{correct}$, and bug information for each data sample.

Unsupervised Dataset Collection. First, we crawled a substantial number of HLS codes from GitHub[3], and then we applied a filtering mechanism to remove noisy notation and markdown files. These crawled data constitute the initial unsupervised dataset with around 30K code samples[4]. To further augment the dataset, we converted many RTL codes from another RTL dataset proposed by [11] into HLS codes using GPT-3.5 [19][5]. In this augmentation step, we further generated around 10K additional code samples. We subsequently prompted GPT-3.5 [19] to refine the potential flaws or errors in both crawled and augmented code samples (Each code sample is denoted as C_{correct}).

Supervised Bug Dataset Generation. Based on the collected unsupervised dataset from the first step, we perform bug insertion to build our desired supervised dataset for fine-tuning LLMs. This bug insertion process begins by pinpointing potential bugs that can be inserted into each original correct code sample C_{correct}. After the analysis, we create various buggy code samples C_{buggy} by substituting originally correct snippet s_{correct} in C_{correct} with buggy snippet s_{buggy}.

As introduced, this supervised dataset will include buggy HLS code and detailed bug labels. We prompt the GPT-3.5 [19] model with

---

[3]The GitHub projects are collected under the HLS Topic (https://github.com/topics/hls with around 100 projects.)

[4]We split the crawled context into individual code samples, each starting with an "module" and ending with an "end module" keyword.

[5]This augment process is mainly intended to provide a larger code context for training open-source LLMs, thus functional matching between the original RTL code and converted HLS codes is not strictly enforced.
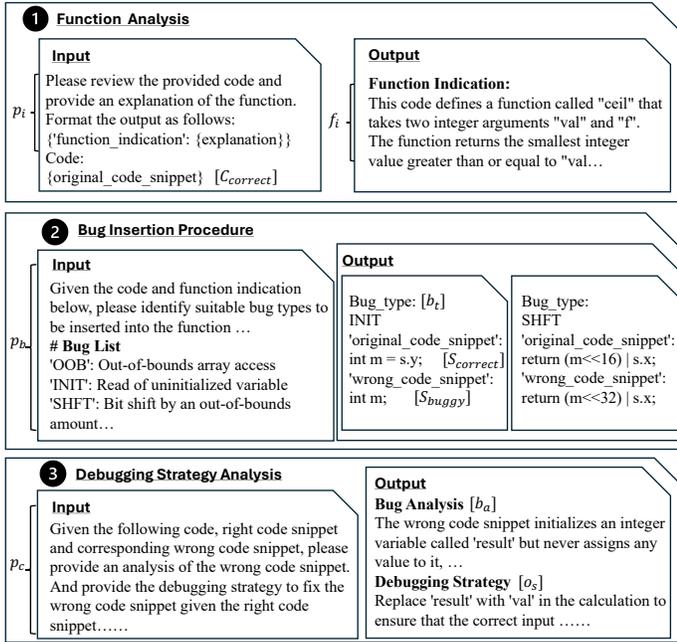
Fig. 3: Our supervised dataset creation process involves three steps: (a) **Function analysis:** LLM reviews and refines code to generate functionality indication f_i. (b) **Bug insertion procedure:** LLM selects and inserts bugs based on a provided list and function indication, generating a dataset for bug identification and correction. (c) **Debugging strategy analysis:** A subset of data is used to generate bug analysis and debugging strategies, enhancing LLM's bug correction capabilities.

CoT to generate the supervised bug dataset through three sub-steps: 1) function analysis; 2) bug insertion procedure; and 3) debugging strategy analysis. Each generation step takes the output of its previous step as its input.

Figure 3 (a) shows the step 1 named function analysis. We first design an instruction prompt p_i and concatenate the correct code sample $C_{correct}$ after it. We then feed the content $(p_i || C_{correct})$ into the GPT-3.5 [19], which we denote as $\mathbb{G}_1$. $\mathbb{G}_1$ is instructed to generate the function indication f_i:

$$f_i = \mathbb{G}_1(p_i || C_{correct})$$

The output f_i will be the input of the next step. As Figure 3 shows, the functionality indication f_i involves the variable definitions, inputs, outputs, and operations.

Figure 3 (b) shows the step 2 named bug insertion procedure. GPT-3.5 ($\mathbb{G}_2$) [19] is prompted by p_b to select relevant bug types suitable for the given refined code $C_{correct}$ from 18 types of logic bugs, with details given in Section V-A. The bug list will be directly concatenated into the prompt p_b. Each resulting bug sample contains the bug type b_t, original code snippet (correct code snippet) $s_{correct}$, and buggy code snippet s_{buggy}. This process can be formulated as follows:

$$\{s_{correct}, s_{buggy}, b_t\} = \mathbb{G}_2(p_b || C_{correct} || f_i)$$

We create the buggy code sample C_{buggy} simply by replacing the original correct snippet $s_{correct}$ with the buggy snippet s_{buggy}. The main supervised dataset is generated through these first two steps, including 300K data samples.
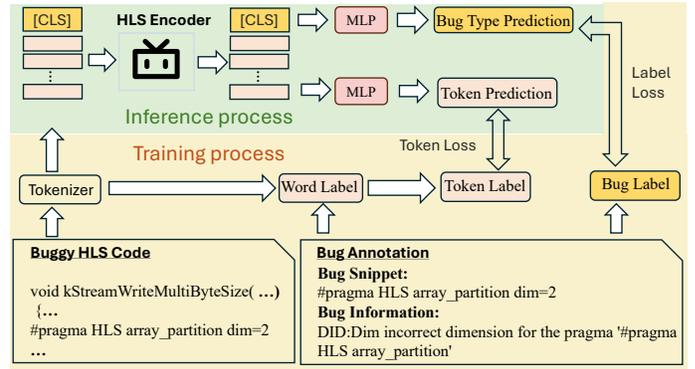

Fig. 4: The inference and training schematic for HLS Encoder. Given the tokenized HLS buggy code, we begin by concatenating the [CLS] token at the beginning. The [CLS] token represents the code context, which will later be used by the MLP head to predict the bug type.

Figure 3 (c) shows the step 3 named the debugging strategy analysis. We further create a small dataset, including bug analysis b_a and debugging strategy o_s. For the generation, we combine the specific prompt p_c, buggy code snippet s_{buggy}, which pinpoints the exact bug, and the corresponding correct code snippet $s_{correct}$.

$$b_a, o_s = \mathbb{G}_3(p_c || C_{buggy} || s_{correct} || s_{buggy})$$

The bug analysis o_s explains the cause of the bug in the buggy snippet s_{buggy}, indicating its difference from the correct functionality. The debugging strategy o_s provides modification suggestions based on the bug analysis b_a and the buggy sample C_{buggy} and correct sample $C_{correct}$.

The final supervised dataset includes both the main dataset $\{s_{correct}, s_{buggy}, b_t\}$ from step 2 and additional debugging strategy information from $\{b_a, o_s\}$ step 3, as formulated below.

$$\{s_{correct}, s_{buggy}, b_t\}, \{b_a, o_s\} = \mathbb{G}_3 | \mathbb{G}_2 | \mathbb{G}_3(p_i || C_{correct}) \quad (1)$$

B. Encoder-Decoder Structure of HLSDebugger

In this section, we introduce the encoder-decoder structure of HLSDebugger, illustrated in Figure 1(b). Our preliminary studies indicate that large language models (LLMs) frequently misidentify buggy code locations, leading to incorrect corrections. We define this issue as error accumulation, which presents a significant bottleneck in the traditional single decoder-only LLM approach during the localization-then-correction process. Additionally, even when they identify the correct snippets, they frequently fail to produce appropriate corrections. This might be because the LLMs still fail to fully understand the context and bug reason. To address this, we propose an encoder-decoder architecture. This approach allows us to train the encoder explicitly to identify bug locations more accurately. Additionally, the encoder provides a soft representation of the bug location, rather than a potentially misleading explicit string. This soft representation, integrated with the decoder through cross-attention, includes more contextual information beneficial for accurate bug correction, thereby enhancing the overall performance of the system.

Based on this observation, we propose to adopt an encoder-decoder structure for HLSDebugger, which consists of an HLS Encoder and an HLS Decoder, as shown in Figure 5. The HLS Encoder is tasked to identify the location and type of bugs. It also provides contextual
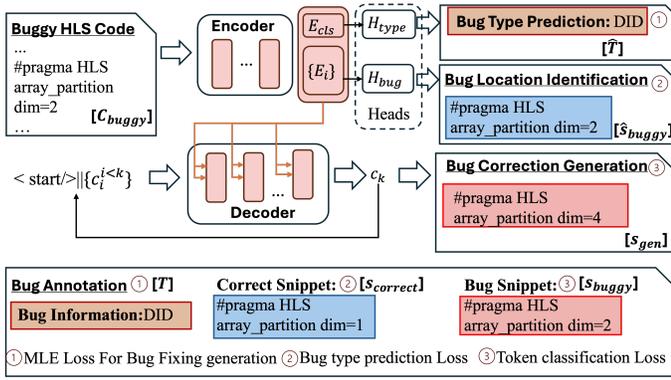
Fig. 5: The interaction between the HLS encoder and HLS decoder within HLSDebugger, along with the ensemble combined loss. The total loss function is the sum of the bug type prediction loss, bug location loss, and bug correction loss.

Fig. 6: Combination of *encoder objective* and *decoder objective* for the encoder-decoder training scheme.

information to support the HLS Decoder by directing their attention to the critical buggy snippet within the context.

HLS Encoder. Figure 4 showcases the process of the HLSEncoder generating the context representation and using the prediction head for bug type prediction and bug location identification. The HLSEncoder uses a stack of transformer blocks, each consisting of self-attention layers and feed-forward networks. The self-attention mechanism allows the encoder to capture the context and relationships between different parts of the input sequence. The representation generated by the HLSEncoder is a high-dimensional vector, which encapsulates its semantic meaning. This representation is then used as input to two distinct heads: one for predicting the type of bug and another for pinpointing its location. Assume there are N tokens in the buggy code C_{buggy}. The input into the encoder is the concatenation of the N-token buggy code C_{buggy} after a special token named $[CLS]$, denoted as $([CLS]||C_{\text{buggy}})$. This special leading token $[CLS]$ represents the whole code functionality as well as the bug information. The encoder will generate an embedding vector for each of the N tokens, denoted as $\{E_1, E_2, ..., E_N\}$, as well as a leading embedding E_{cls} for the special token $[CLS]$. This encoding process is formulated below.

$$E_{\text{cls}},\ E_1,\ ...,\ E_N = \text{Encoder}([\text{CLS}]||C_{\text{buggy}}) \qquad (2)$$

After encoding the potentially buggy code C_{buggy}, these context embeddings $E_{\text{cls}}, \{E_1, E_2, ..., E_N\}$ will be fed into the *prediction heads* ($\mathbb{H}_{\text{type}}, \mathbb{H}_{\text{bug}}$). Each prediction head is based on a basic multilayer perceptron (MLP). Prediction heads $\mathbb{H}_{\text{type}}$ and $\mathbb{H}_{\text{bug}}$ will predict the bug type (prediction denoted as $\hat{T}$) of the overall code and identifying whether each of the N tokens is buggy (prediction denoted as $\hat{I}_1, ..., \hat{I}_N$), respectively:

$$\hat{I}_1, ..., \hat{I}_N = \mathbb{H}_{\text{bug}}(E_1, ..., E_N)$$
$$\hat{T} = \mathbb{H}_{\text{type}}(E_{\text{cls}})$$

HLS Decoder. The HLS Decoder is tasked to generate the bug correction C_{gen}, which is similar to the HLS Encoder as a set of transformer blocks, but it also includes cross-attention layers that allow it to 'attend' to the encoded input representation. The final layer of the decoder generates sequences of corrected buggy snippets. Figure 5 shows the dynamics between the encoder and decoder. At each time, Decoder will generate the probability prediction for the next token c_k in all vocabularies, based on all previous tokens before
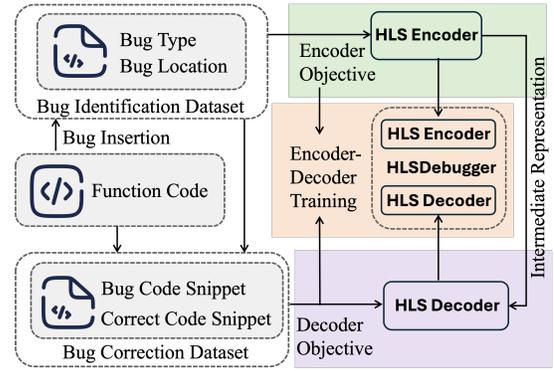
c_k, which can be denoted as $\{c_i^{\{i<k\}}\}$.

$$c_k = \text{Decoder}(<\text{start/}>||\{c_i^{i<k}\}, \{E_{\text{cls}}, E_1, ..., E_N\}) \qquad (3)$$

c_k represents the k_{th} token of the correct code snippet sequence s_{gen}. The input into Decoder $<\text{start/}> ||\{c_i^{i<k}\}$ is the concatenation of the start token $<\text{start/}>$ and previous generated sequence. The other input $\{E_{\text{cls}}, E_1, ..., E_N\}$ is the output embeddings from the encoder. It is provided through the decoder's cross-attention layers [20]. In the cross-attention layers, the encoder outputs $\{E_{\text{cls}}, E_1, E_2, ..., E_N\}$ will be taken as key (K) and value (V) embeddings, while the intermediate result from each previous decoder layer will be treated as a query (Q). The interaction between decoder intermediate results and encoder output embeddings in cross-attention layers enables the decoder to selectively attend to relevant tokens and improve the quality of bug correction.

The decoder and encoder are trained together to learn the interaction with each other with a customized loss, as we will introduce in the following sub-section.

C. Training Scheme for Encoder-Decoder Structure

Most decoder-only LLMs for generative tasks adopt the basic maximum likelihood estimation (MLE) loss for their fine-tuning process. This cannot fulfill our requirements for fine-tuning encoder-decoder structured HLSDebugger on multiple tasks. We fill the gap by proposing a customized training scheme that simultaneously trains both the encoder and decoder with distinct loss functions, as illustrated in Algorithm 1.

Encoder Training Loss. As introduced in Section IV-B, the input to the encoder is the buggy code sample C_{buggy}, and the prediction based on encoder outputs is the bug type prediction $\hat{T}$ and the binary bug classification for each token $\hat{I}_1, ..., \hat{I}_N$. The encoder's objective is to minimize the combined loss of bug type prediction and bug location identification.

The bug type prediction loss is the cross-entropy loss (TypeLoss in Algorithm 1) for multi-label classification on $\hat{T}$, denoted as L_{type}. The token classification loss for bug location identification is the simple binary cross-entropy loss (BugLoss in Algorithm 1) on $\{\hat{I}_1, ..., \hat{I}_N\}$, denoted as L_{bug}. Additionally, the token distribution is unbalanced since most tokens are labeled with *no bug* (i.e., false in token classification). To account for the imbalanced distribution, we assign a scaling factor of α_t to true labels and α_f to false labels when calculating L_{bug}. For the combinational loss of bug type and token classification, we further assign a factor α_{type} to bug type loss L_{type}

Algorithm 1: Training for Encoder-Decoder Structure

---

1 **Intialize Training**: training epochs: N, training dataset: D, set iteration index $iter = 0$, optimizer: **Optm**;
2 **Set Loss Factor**: Bug type: α_{type}, bug location: α_{bug}, buggy token factor: α_t, correct token factor: α_f, bug correction generation α_{decoder} ;
3 **TypeLoss** $\leftarrow$ LogitCrossEntropyLoss;
4 **BugLoss** $\leftarrow$ LogitCrossEntropyLoss(α_t, α_f);
5 **GenLoss** $\leftarrow$ LogitMaximumLikelihoodEstimation;
6 **while** $iter < N$ **do**
7   **for** *batch B in D* **do**
8       $C_{\text{buggy}}, \{I_1, I_2, ..., I_N\}, T, s_{\text{correct}} \leftarrow B$;
9       $E_{\text{cls}}, \{E_1, E_2, ..., E_N\} = \text{Encoder}([\text{CLS}]||C_{\text{buggy}})$;
10      $\{\hat{I}_1, ..., \hat{I}_N\} = \mathbb{H}_{\text{bug}}(\{E_1, E_2, ..., E_N\})$;
11      $\hat{T} = \mathbb{H}_{\text{type}}(E_{\text{cls}})$;
12      $s_{\text{gen}} \leftarrow \text{Decoder}(E_{<\text{start/}>}, \{E_{\text{cls}}, E_1, ..., E_N\})$;
13      $L_{\text{type}} = \textbf{TypeLoss}(T, \hat{T})$;
14      $L_{\text{bug}} = \textbf{BugLoss}(\{I_1, ..., I_N\}, \{\hat{I}_1, ..., \hat{I}_N\})$;
15      $L_{\text{encoder}} = \alpha_{\text{type}} \cdot L_{\text{type}} + \alpha_{\text{bug}} \cdot L_{\text{bug}}$
16      $L_{\text{decoder}} = \textbf{GenLoss}(s_{\text{correct}}, s_{\text{gen}})$;
17      $L_{\text{all}} = \alpha_{\text{encoder}} \cdot L_{\text{encoder}} + \alpha_{\text{decoder}} \cdot L_{\text{decoder}}$;
18      $L_{\text{all}} . \text{backward}()$;
19   **Optm**. $\text{step}()$;

---

and a factor α_{bug} to token classification loss L_{bug}. The loss function of the encoder is defined below:

$$L_{\text{encoder}} = \alpha_{\text{type}} \cdot L_{\text{type}} + \alpha_{\text{bug}} \cdot L_{\text{bug}} \quad (4)$$

Decoder Training Loss. During training, the correct code snippet s_{correct} in the desired code sample C_{correct} serves as the ground truth for the decoder. s_{correct} is a sequence of tokens with a total length of K. The decoder's output, s_{gen}, is compared against s_{correct} to calculate the loss. Denote the sequence of tokens before k_{th} in s_{correct} as $\{c_i^{<k}\}$, the overall MLE loss is:

$$L_{\text{decoder}} = -\sum_{k=1}^{K} \log \text{Decoder}(<\text{start/}>||\{c_i^{<k}\}) \quad (5)$$

In this equation, the output from the HLS Decoder is the probability of the k_{th} token across all vocabularies. The loss function aims to maximize the log-likelihood of the correct snippet sequence s_{correct}, encouraging the decoder to predict the correct token at each step in sequence generation.

Combined Loss for Ecoder-Decoder. Finally, we combine the loss terms of both encoder and decoder, scaling them with hyper-parameters α_{encoder} and α_{decoder}. The final loss is shown below.

$$L_{\text{all}} = \alpha_{\text{encoder}} \cdot L_{\text{encoder}} + \alpha_{\text{decoder}} \cdot L_{\text{decoder}} \quad (6)$$

During the training process, we noticed that bug correction by the decoder is more challenging than identifying bug types and locations by the encoder. Therefore, we attribute a higher loss scaling factor α_{decoder} to the decoder loss L_{decoder}. Based on the combined loss, the encoder and decoder in the HLSDebugger are trained simultaneously to learn a better interaction through cross-attention layers.

V. EXPERIMENTAL SETUP

Before introducing results, in this section, we will introduce the details about the setup of our experiment, including the testing benchmark and tasks (Sec V-A), the implementation details of our HLSDebugger model (Sec V-B), the evaluation metrics (Sec V-C), and the selection of baseline models (Sec V-D).

A. Experimental Benchmark and Task

Evaluation Benchmark. To evaluate HLSDebugger and other LLM solutions, we utilize the benchmark[6] from [1], which comprises 8 different types of logic bugs. All of the types of bugs and a broad spectrum of designs are covered in the selected test samples. The 8 types of bugs are listed as follows [1]:

1) `OOB` - Out-of-bounds array access
2) `INIT` - Read of uninitialized variable
3) `SHFT` - Bit shift by an out-of-bounds amount
4) `INF` - An infinite loop arising from an incorrect loop termination
5) `USE` - Unintended sign extension
6) `MLU` - Errors in manual loop unrolling
7) `ZERO` - Variable initialized to zero instead of nonzero initializer
8) `BUF` - Copying from the wrong half of a split buffer

The bug location identification performance is evaluated in token-wise, line-wise, and code-wise granularities. The bug correction generation accuracy is evaluated by matching the ground truth correct code snippet with the generated outputs. Detailed metrics are introduced in the section (V-C).

B. HLSDebugger Implementation in Experiment

HLSDebugger Structure Details. We developed the HLSDebugger based on a pre-trained CodeT5-Large [22] model, with 24 transformer layers for both encoder and decoder (the dimensions of the embeddings are 1024, and the head dimension is 16). The number of layers for our MLP prediction heads ($\mathbb{H}_{\text{bug}}$, $\mathbb{H}_{\text{type}}$) is set to 3.

HLSDebugger Training Details. To avoid potential information leakage (i.e., overlap of data samples between training dataset and testing benchmark [1]), we utilize the Rouge-L [23] metric to calculate the similarity between codes in our unsupervised dataset and that in the testing benchmark [1]. We eliminate code samples that exhibit high similarity to the testing benchmark [1]. This operation removes approximately 2K code samples with a Rouge-L similarity greater than 0.5. The finalized number of training code samples in our dataset is around 300K. For training HLS Debugger, the factors $\alpha_{\text{type}}, \alpha_{\text{bug}}, \alpha_{\text{gen}}$ in the Algorithm 1 for are: 0.2, 2, 10. The scaling factors (α_t, α_f) for token identification loss (BugLoss in Algorithm 1) are set as 0.05, 1. We also fine-tuned an Encoder with only bug identification loss, denoted as HLSEncoder, for an ablation study. The factors are set with the same proportion (α_{type}, $\alpha_{\text{bug}} = 0.2$, 2) but larger scale for better convergence.

C. Evaluation Metrics

In this subsection, we introduce the metrics used to evaluate the performance of LLMs for both bug identification and correction.

Bug Identification Metrics. We evaluate the bug identification accuracy of all models at three different granularities: 1) Token-wise classification. Similar to the encoder training process, token-wise evaluation treats bug location as a token classification problem. 2) Line-wise string classification. The line-wise evaluation assesses the match between predicted and actual locations at the line level. 3) Code-wise evaluation based on top-k line predictions. It evaluates whether the bug in a code sample is correctly identified. It is based

---

[6]To avoid a long overall inference time of LLMs during decoding, we randomly selected 2K samples from the benchmark in [1] for the evaluation of all models. Our experiments validate that all model testing accuracies converge very well after more than 500 samples are tested (i.e., change in average test accuracy is negligible when adding more test samples).

Model	Token-wise accuracy				Line-wise accuracy				Code-wise accuracy	
	Precision	Recall	f1	RoC	Precision	Recall	f1	RoC	Top-1	Top-5
GPT-3.5 [19]	10.1%	29.1%	0.136	0.513	10.6%	15.4%	0.121	0.554	16.3%	53.4%
GPT-4 [19]	12.2%	37.1%	0.168	0.554	13.3%	17.5%	0.145	0.566	19.7%	54.7%
Gemma-7B [21]	7.2%	21.0%	0.097	0.447	7.6%	22.3%	0.104	0.451	15.5%	60.7%
Gemma-7B UFT	7.3%	18.8%	0.092	0.476	7.5%	19.6%	0.096	0.475	26.1%	67.3%
Gemma-7B SFT	22.2%	47.7%	0.279	0.648	20.0%	46.4%	0.255	0.632	37.1%	69.9%
HLSEncoder	46.1%	79.8%	0.539	0.913	50.5%	75.4%	0.564	0.899	76.7%	86.9%
HLSDebugger	54.3%	83.5%	0.614	0.935	57.7%	78.4%	0.627	0.923	79.5%	87.6%

TABLE II: Evaluation of bug identification accuracies on different granularities.

on the correctness of the top-1 and top-5 line predictions within each code sample. All evaluations utilize binary classification metrics such as Precision, Recall, F1 score, and the area-under-curve (AUC) of the ROC curve [24].

The precision measures the ratio of correctly identified bug locations among all predicted locations, and recall evaluates the proportion of correctly identified bug locations against actual bug locations. It's worth noting the overall number of buggy tokens/lines is far less than non-buggy tokens/lines. In other words, the overall dataset is skewed by class bias. Therefore, it is important to report both the precision and recall of each model for a fair evaluation.

Bug Correction Generation Metrics. To evaluate the performance of LLMs in generating bug corrections, we evaluate whether LLM-generated code snippets align with the desired functionality. Since LLM decoders are generative models, the outputs they produce are in a flexible format, with some contents that are not code but describe the underlying structure and semantics of the code. Thus, evaluation metrics that are too strict may underestimate the real performance of LLMs. To better evaluate the quality of the LLMs' ability on bug correction generation, we employ a 'strict substring match' evaluation metric. This metric compares the processed[7] output from the LLMs with the correct label code snippet, trying to capture a strict match[8].

D. Baseline Selection

We use GPT-3.5 and GPT-4 [19], the most widely used commercial LLM solutions, as commercial baseline models for comparison. For open-source LLMs, we experimented on various models, including Gemma-7B, Mistral-7B [25], DeepSeek [26], and Code-Llama [27]. Additionally, to evaluate the impact of bug correction loss on the bug identification performance, we evaluate the performance of HLSEncoder (the Encoder fine-tuned with only bug identification loss) as the ablation study.

VI. EXPERIMENTAL RESULT

A. Bug Identification Performance

Table II presents the comparison of bug identification performance at token-level, line-level, and code-level, respectively. Results show that HLSDebugger significantly outperforms both the open-source LLM (Gemma-7B) and the commercial GPTs [19] across all three granularities. Even with the significant improvement of the supervised dataset on improving Gemma-7B, HLSDebugger still achieves superior performance than Gemma-7B-SFT, demonstrating the effectiveness of our encoder-based bug identification solution and debugging training scheme.

Specifically, for the token prediction ('Token-wise accuracy' in Table II), HLSDebugger shows a significant improvement over GPT-4 [19]. Its precision of 54.3% is more than four times higher, and the recall of 83.5% is more than double that of GPT-4 [19]. The high recall indicates that the HLSDebugger can capture most of the bugs in the code. The HLSEncoder achieves a recall rate that is nearly comparable to that of the HLSDebugger (79.8% versus 83.5%), suggesting that HLSEncoder has a similar capability to HLSDebugger in identifying bugs. However, HLSEncoder has a lower precision (46.1% compared to 54.3%), indicating that HLSDebugger is more effective at accurately understanding detailed information.

For line-wise prediction ('Line-wise accuracy' in Table II), experiments show a similar trend as in token-wise evaluation. In practice, locating the specific buggy line of code is sufficient for identifying the problem. HLSDebugger still achieves over two times higher recall and precision than GPT-4 [19], indicating greater robustness and consistency. At the higher level (line-wise) identification, the HLSEncoder shows lower performance than HLSDebugger, but significantly higher than Gemma-7B SFT, which aligns with the results on the 'token-wise accuracy'.
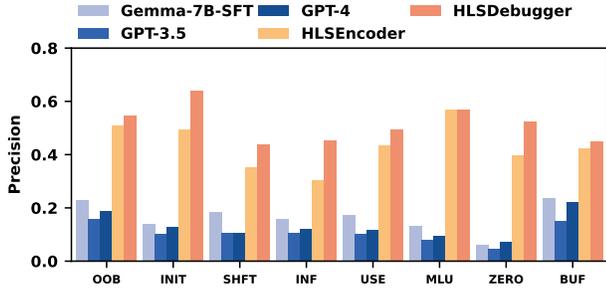
For code-wise evaluation based on top-k code line predictions ('Code-wise accuracy' in Table II), the HLSDebugger notably excels, particularly with the top-1 code line. As defined in Section V-C, top-1 accuracy measures whether the highest probability code line prediction is correct, while top-5 accuracy requires only one of the top-5 line predictions to be correct. Consequently, top-5 accuracy is expected to be substantially higher than top-1. Notably, the accuracy for top-1 and top-5 lines of HLSDebugger reaches 79.5% and 87.6% for code-level evaluation, indicating that our model can successfully identify bugs in most buggy code samples with 1 to 5 trials.
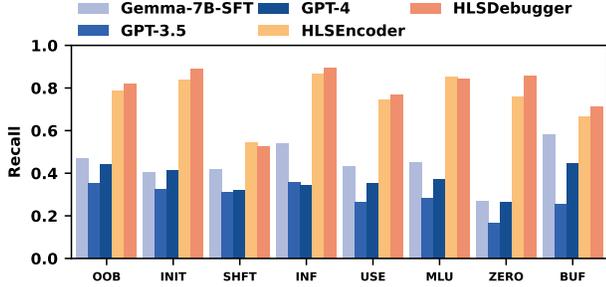
B. Bug Correction Performance

In addition to purely identifying bugs, an even more challenging task is to correct the bugs in a given code sample C_{buggy}. No

Model	Bug Correction Accuracy	Model Type
GPT-3.5 [19]	8.5%	Commercial Model
GPT-4 [19]	10.5%	
Gemma-7B [21]	7.5%	Open-sourced model Not fine-tuned
Gemma-7B UFT	6.3%	Fine-tuned on our dataset
Gemma-7B SFT	15.1%	
HLSEncoder + GPT-4	18.9%	Commercial model + our encoder
HLSEncoder + Gemma-7B SFT	23.7%	SFT Gemma 7B + our encoder
HLSDebugger	**37.6%**	Our solution

TABLE III: Bug correction accuracy comparison. HLSDebugger outperforms GPT-4 by more than 3×.

---

[7]The string will undergo processing to eliminate spaces and special characters.
[8]The label (correct snippet) will be used to strictly match with output of the LLMs within a limited length. Here we limit the LLMs' generation length to a maximum 3 times of the length of the snippet label.

(a) Bug identification precision of different LLMs on all bug types.

(b) Bug identification recall of different LLMs on all bug types.

Fig. 7: Bug identification performance of different LLM solutions on all bug types, including HLSEncoder for ablation study.

information about bug locations is provided. Table III shows the performance of LLM solutions in bug correction tasks. Despite Gemma-7B-UFT's enhanced bug identification (Section VI-A), fine-tuning with an unsupervised dataset reduces its performance to 6.3% on bug correction, compared to 7.5% with Gemma-7B. HLSDebugger accurately corrected bugs in 37.6% of given buggy code samples, more than 3 times better than commercial tool GPT-4's 10.5%. The supervised fine-tuning improves the correction accuracy of Gemma-7B from 3.5% to 15.1% (Gemma-7B SFT). The HLSEncoder further improves the performance of Gemma-7B SFT by 7.6%, ranking 2nd with 23.7%. Additionally, the HLSEncoder improves the performance of GPT-4 [19] by 8.4%, ranking 3rd with 18.9%. The above results demonstrate the effectiveness of both our proposed dataset and HLSEncoder. The gap between HLSDebugger and HLSEncoder+Gemma-7B-SFT (37.6% v.s. 23.7%) implies the advantage of the encoder-decoder structure that combines bug identification with correct tasks.

Overall, we admit that all existing LLMs, including HLSDebugger with the best 37% accuracy, are not yet ready for practical debugging applications. But the obvious $3\times$ superior performance of HLSDebugger over GPT-4 implies room for improvement for specific debugging tasks with customized models. With a larger and higher-quality training dataset, as well as a larger-scale and more LLM structure, we expect a better performance of LLM-aided HLS debugging.

VII. DISCUSSION

In this section, we present more detailed results of HLSDebugger on each individual bug type. We also report detailed results of GTP-4 [19] and Gemma-7B-SFT, the two strongest baselines.

We further provide detailed results on each of the 8 types of bugs. The detailed results cover two parts: 1) Token-wise bug identification, 2) Bug correction accuracy.
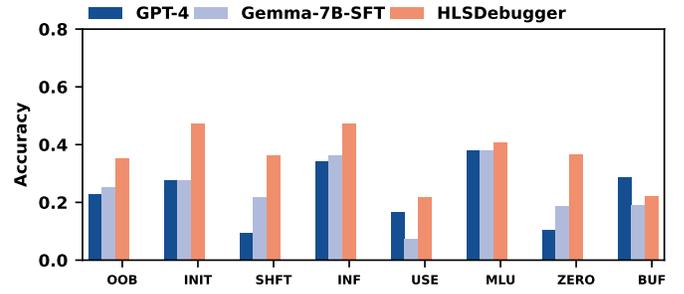

Fig. 8: Bug correction accuracy of three models on different bug types. The blue bars correspond to normal bugs, while the orange bars correspond to pragma-related bug types.

Model	Bug Correction Accuracy
GPT-3.5 [19]	36.0%
GPT-4 [19]	43.6%
Gemma-7B-UFT [21]	14.7%
Gemma-7B-SFT [21]	45.4%
HLSDebugger	**49.1%**

TABLE IV: Bug correction accuracy given ground-truth bug location.

Token-wise bug-identification. Figure 7 shows detailed results on identification (a) precision and (b) recall for each individual bug type. The accuracy of HLSDebugger significantly exceeds GPT-4 [19] and open-source models across all bug types. This superior performance is consistent with the overall results in Table II.

Bug Correction Accuracy. Figure 8 shows a detailed bar plot of the bug correction accuracy of various LLM solutions and our HLSDebugger on each specific bug type. We can observe that all three models show similar trends across different bug types. Gemma-7B-SFT slightly outperforms GPT-4 [19] in bug corrections. HLSDebugger demonstrates superior performance to both GPT-4 [19] and Gemma-7B-SFT across the majority of the 8 bug types.

Bug Correction Accuracy Given Location. In addition to bug identification, another contribution of HLSDebugger is bug correction. We add a new experiment in Table IV to evaluate the bug correction accuracy, assuming the correct bug location is given (i.e., assuming 100% identification accuracy). Given the ground-truth bug identifications, all LLMs' performance has been improved. HLSDebugger still achieves the highest performance 49.1% in this sole bug correction task.

VIII. CONCLUSION

In conclusion, we target the problem of debugging HLS logic bugs, addressing three challenges: 1) scarcity of circuit data, 2) challenge of logic bugs, 3) multi-tasking in HLS debugging. We propose the HLSDebugger, a customized LLM-based method for HLS logic debugging tasks. HLSDebugger first generates a large-scale labeled dataset, then adopts a customized structure, and also proposes a debugging training scheme. Our solution achieved higher performance on both bug identification and correction generation compared with commercial LLM solutions.

IX. ACKNOWLEDGEMENT

REFERENCES

[1] L. J. Wan, Y. Huang, Y. Li, H. Ye, J. Wang, X. Zhang, and D. Chen, "Invited paper: Software/hardware co-design for llm and its application for design verification," in *In ASP-DAC*, 2024, pp. 435–441.

[2] B. Yang, H. Tian, W. Pian, H. Yu, H. Wang, J. Klein, T. F. Bissyandé, and S. Jin, "Cref: An llm-based conversational software repair framework for programming tutors," *arXiv preprint arXiv:2406.13972*, 2024.

[3] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao, "Agentfl: Scaling llm-based fault localization to project-level context," *arXiv preprint arXiv:2403.16362*, 2024.

[4] C. Lee, C. S. Xia, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," 2024.

[5] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning llms for fl-free program repair," *arXiv preprint arXiv:2404.08877*, 2024.

[6] K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: An ai-powered debugging assistant," *arXiv preprint arXiv:2403.16354*, 2024.

[7] L. Zhong, Z. Wang, and J. Shang, "Ldb: A large language model debugger via verifying runtime execution step-by-step," *arXiv preprint arXiv:2402.16906*, 2024.

[8] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," 2023.

[9] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[10] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[11] 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," 2023.

[12] Y. Zhang, Z. Yu *et al.*, "Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation," *arXiv preprint arXiv:2407.01910*, 2024.

[13] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multilingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017.

[14] J. Czerwonka, M. Greiler, C. Bird, L. Panjer, and T. Coatta, "Codeflow: Improving the code review process at microsoft: A discussion with jacek czerwonka, michaela greiler, christian bird, lucas panjer, and terry coatta," *Queue*, vol. 16, no. 5, oct 2018.

[15] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "Verilogreader: Llm-aided hardware test generation," 2024.

[16] S. Qiu, B. Tan, and H. Pearce, "Explaining eda synthesis errors with llms," *arXiv preprint arXiv:2404.07235*, 2024.

[17] L. Collini, S. Garg, and R. Karri, "C2hlsc: Can llms bridge the software-to-hardware design gap?" *arXiv preprint arXiv:2406.09233*, 2024.

[18] C. Xiong, C. Liu, H. Li, and X. Li, "Hlspilot: Llm-based high-level synthesis," 2024.

[19] 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.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[21] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love *et al.*, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[22] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *ArXiv*, 2021.

[23] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004.

[24] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[25] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," 2024.

[27] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," 2023.