Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models

Kangwei Xu[1], Grace Li Zhang[2], Xunzhao Yin[3], Cheng Zhuo[3], Ulf Schlichtmann[1], Bing Li[4]
[1]*Chair of Electronic Design Automation, Technical University of Munich (TUM)*, Munich, Germany
[2]*Hardware for Artificial Intelligence Group, Technical University of Darmstadt*, Darmstadt, Germany
[3]*College of Information Science and Electronic Engineering, Zhejiang University*, Hangzhou, China
[4]*Research Group of Digital Integrated Systems, University of Siegen*, Siegen, Germany
Email: {kangwei.xu, ulf.schlichtmann}@tum.de, grace.zhang@tu-darmstadt.de, {xzyin1, czhuo}@zju.edu.cn, bing.li@uni-siegen.de

***Abstract*—In High-Level Synthesis (HLS), converting a regular C/C++ program into its HLS-compatible counterpart (HLS-C) still requires tremendous manual effort. Various program scripts have been introduced to automate this process. But the resulting codes usually contain many issues that should be manually repaired by developers. Since Large Language Models (LLMs) have the ability to automate code generation, they can also be used for automated program repair in HLS. However, due to the limited training of LLMs considering hardware and software simultaneously, hallucinations may occur during program repair using LLMs, leading to compilation failures. Besides, using LLMs for iterative repair also incurs a high cost. To address these challenges, we propose an LLM-driven program repair framework that takes regular C/C++ code as input and automatically generates its corresponding HLS-C code for synthesis while minimizing human repair effort. To mitigate the hallucinations in LLMs and enhance the prompt quality, a Retrieval-Augmented Generation (RAG) paradigm is introduced to guide the LLMs toward correct repair. In addition, we use LLMs to create a static bit width optimization program to identify the optimized bit widths for variables. Moreover, LLM-driven HLS optimization strategies are introduced to add/tune pragmas in HLS-C programs for circuit optimization. Experimental results demonstrate that the proposed LLM-driven automated framework can achieve much higher repair pass rates in 24 real-world applications compared with the traditional scripts and the direct application of LLMs for program repair. The codes are open-sourced at this link: https://github.com/code-source1/catapult.**

I. Introduction

High-Level Synthesis (HLS) tools have achieved significant progress in automatically generating Hardware Description Languages (HDL) such as Verilog from general-purpose programming languages such as C/C++. This advancement allows software engineers to contribute to hardware design more actively, thereby lowering the expertise barrier in hardware design [1]–[3]. However, HLS tools only support a subset of C/C++ programs, necessitating significant manual rewriting to transform regular C/C++ programs into their HLS-compatible C/C++ counterparts (HLS-C) [4]–[7]. In other words, given C/C++ programs should be *repaired* before they can be processed by HLS tools to generate the corresponding circuits. For example, the use of pointers, recursion, and dynamic memory should be manually rewritten, as such programs are not compatible with the existing HLS tools. In addition, the integer arithmetic in CPU instructions is typically defined with 32/64 bits, whereas in circuit design, the bit widths are customized to enhance hardware efficiency [11]. An unnecessarily large bit width can lead to increased area overhead, high power consumption, and large clock periods in hardware implementation [12].

Several methods have been proposed for *program repair* in HLS taking C/C++ programs as input. [4] statically analyzes the pointers in C/C++ programs to reduce global connections. However, the challenge remains when the application accesses memory dynamically with various complex pointers. [6] provides Domain Specific Languages (DSLs) on top of C++ programs to support recursion in HLS, but control statements need to be manually added. [7] supports dynamic memory by providing data structure templates. However, only a limited set of data structures is supported, and significant refactoring is still required. [8] presents an HLS backend for generating a customized accelerator using C++ template-based, parameterized types. However, this approach requires the user to manually specify the bit widths. [9] reduces bit widths through both program profiling and bit analysis. When inputs exceed the typical range, a fallback function is triggered, which is an error-prone process.

Large Language Models (LLMs) have exhibited great potential in automating both software and hardware design, supporting engineers throughout the design flow from initial concepts, algorithms, and architectures to debugging, verification, and optimization [13]–[18]. Recent studies have demonstrated that LLMs can be used to correct syntax and logic errors in Verilog and C/C++ programs [20]–[22]. [14] proposes an iterative and conversational-based approach to generate Verilog codes using LLMs, and [20] harnesses the capabilities of LLMs as autonomous agents, incorporating human knowledge for reasoning and action planning to automate syntax error fixing for Verilog code. For C/C++ programs, [23] introduces iterative approaches that repeatedly query LLMs based on feedback from previous fix attempts to automate program repair. [24] proposes an LLM-driven program repair approach, where the buggy code is removed, and the LLM directly predicts the correct code given the prefix and suffix context.

Previous methods above use LLMs to generate codes or fix simple errors in existing codes. However, repairing regular C/C++ codes so that they can be converted into HLS-C for hardware design with HLS tools has not yet been well studied. In this paper, we propose an automated LLM-driven C/C++ program repair framework for HLS, which employs an LLM-based search and iterative repair process to eliminate HLS-incompatible errors while enhancing hardware performance. The key contributions of this paper are summarized as follows.

- We propose an automated C/C++ program repair frame-

This work is licensed under a Creative Commons Attribution International 4.0 License.
MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA
© 2024 Copyright is held by the owner/author(s).
ACM ISBN 979-8-4007-0699-8/24/09.
https://doi.org/10.1145/3670474.3685953

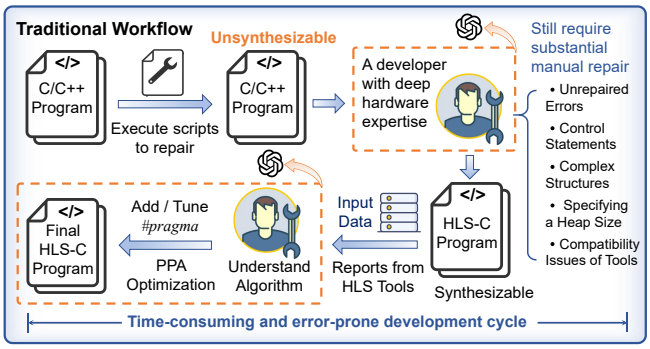

Fig. 1. Traditional workflow for repairing regular C/C++ programs in HLS.

work driven by LLMs for HLS with minimal human effort. The proposed workflow covers runtime profiling and C/C++ program repair for hardware generation, verification, and performance optimization.

- To mitigate factual errors (hallucinations) produced by LLMs, a Retrieval-Augmented Generation (RAG) paradigm is introduced to guide the LLMs toward correct repair. An external library containing correct repair templates is built with human guidance and from the manual of HLS tools. These templates are matched through an embedding retrieval mechanism to enhance the quality of input prompts for LLMs, which leads to a 23.33% increase in the repair pass rate.
- To find the optimal bit widths for variables implemented in hardware, we use the C/C++ code, the corresponding task scenarios, and the description of input data as prompts to create a bit width optimization program via LLMs. This optimization effectively reduces the bit width and achieves an average of 36.57%, 33.03%, and 29.08% reduction in area, power, and minimum clock period, respectively.
- A joint LLM-script repair mechanism is introduced to pre-repair simple errors at early stages and minimize the cost of using LLMs. With the script providing preliminary repair structures, the repair cost of using LLMs can be reduced by an average of 21.56%.
- To achieve a better power, performance, and area (PPA) design of the synthesized circuit, successfully repaired HLS-C programs are collected into a potential list for subsequent optimization. Critical code segments with large area, high power, and high latency are identified by HLS tools. LLMs are used to further optimize these critical code segments to achieve a more efficient PPA design.

The rest of this paper is organized as follows. Section II provides the background and motivation of this work. Section III explains the details of the proposed method. The experimental results are provided in Section IV. Section V concludes the paper.

II. Background and Motivation

Regular C/C++ programs are usually designed to be executed by CPUs; HLS tools often cannot compile them correctly because only a subset of the C/C++ programs, i.e., HLS-compatible C/C++, can be mapped to hardware design directly.

TABLE I
Examples of HLS-incompatible types.

Incompatibility Type	A Typical Symptom (# Error / Warning:)	Corresponding Repair
Pointer	Unsupported feature 'pointers-to-pointers' (CIN-243)	Convert to array access
Dynamic Array	Unsupported feature 'variable length array' (CIN-15)	Specify the array size
Recursion	Unsupported feature 'recursion' (CIN-15)	Replace with a loop
Bit Width	Using ac_int<> need consider bit accurate (CIN-46)	Optimize the bit width
Boolean Operation	Incrementing a bool value is deprecated (CRD-708)	Convert to integer type
Incomplete Statement	Expected a statement (CRD-127)	Complete the statement
Unsupported Struct	Unsupported structs create deadlocks. (HIER-10)	Add explicit constructors
Exception-Handling	Out-of-bounds read/write access (CRD-175)	Add a bound check

Traditionally, making regular C/C++ programs synthesizable requires developers to possess interdisciplinary expert knowledge in both hardware and software, often involving substantial manual rewriting. It is crucial to eliminate or at least reduce such manual effort to allow designers to focus on logic and performance optimization. Table I summarizes several types of HLS incompatibility types with their corresponding error symptoms and repair edits [27]. Key incompatibility types are summarized as follows:

- *Pointer:* Pointers are strictly forbidden in HLS tools, except for statically analyzable ones. Thus, developers need to manually convert pointer access to array access.
- *Dynamic array:* HLS tools do not support dynamic arrays because hardware designs cannot manage data structures with unbounded size. Hence, dynamic array functions such as *malloc()* and *free()* must be replaced with pre-allocated arrays.
- *Recursion:* This design style requires dynamic storage in a stack of execution states and is not supported by HLS tools. Manual transformation into loops is required.
- *Bit Width:* Regular C/C++ programs running on CPUs use standard data types, such as 32-bit for integers, which leads to wasted on-chip resources. The optimal bit widths for variables are needed to be determined to achieve more efficient circuits.
- *Boolean Operation:* HLS tools do not support assignment operators (+=, −=) or increment operators (++, −−) for Boolean values. An appropriate integer type for these operators needs to be used to replace the Boolean type.
- *Incomplete Statement:* HLS tools generate a violation when the case statement does not completely cover the range of values for the object used in conditional expressions. Considering the uncovered values to complete the statement is needed.
- *Unsupported Structures:* HLS tools do not support virtual functions in C++, necessitating the use of strategy or template methods to achieve polymorphic behavior in hardware designs.
- *Exception-handling:* Violations are reported when there are out-of-bound array reads/writes and illegal shifts. Additionally, whereas CPU programs use stack popping to handle exceptions, on ASIC, exception-handling modules need to be built to deliver the termination message to other modules.

Fig. 1 illustrates the traditional workflow, where developers rely on scripts to repair the HLS incompatible programs. However, these scripts can only address basic errors, leaving complex or unforeseen issues unresolved. Therefore, manual repair by a developer with deep hardware expertise becomes essential. After the incompatibility errors are fixed, the developer then applies EDA tools to further optimize the design.

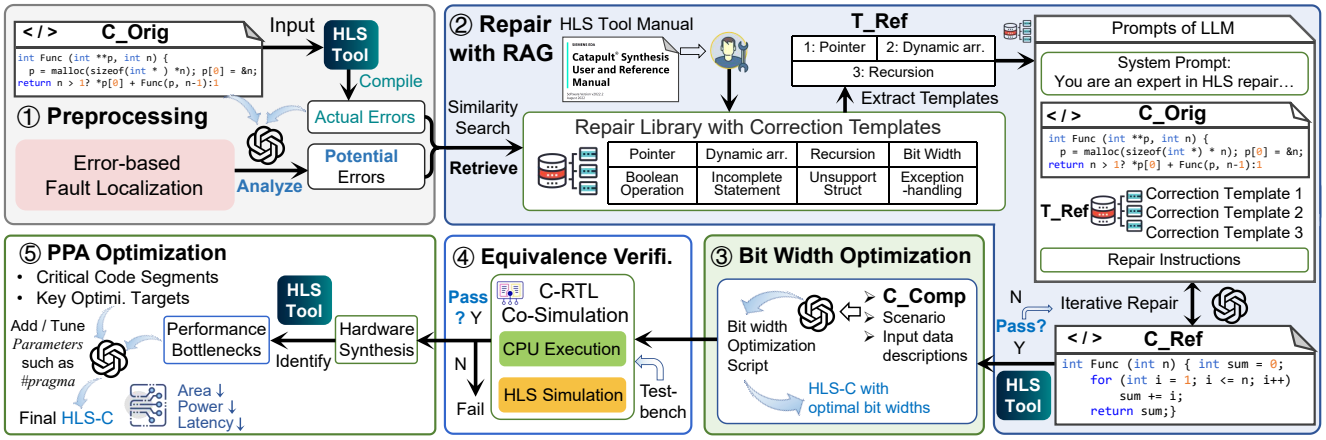

Fig. 2. The proposed LLM-driven automatic C/C++ program repair framework for HLS.

Several techniques have been proposed to address the incompatible issues of regular C/C++ programs in HLS tools [6]–[8]. However, these techniques can only partially resolve the aforementioned errors in regular C/C++ programs. Developers still need to continually apply various scripts implementing these individual techniques to enhance the hardware synthesizability of C/C++ programs, thus leading to error-prone and time-consuming development cycles.

To automate program repair, recent studies have reported promising results of directly applying LLMs [21]–[24] to repair C/C++ programs. But this direct application of LLMs for HLS still faces several main challenges: 1) The majority of research on LLM-based program repair focuses on fixing syntax and logical errors [20] [23], with a notable lack of studies on the automatic structural repair of C/C++ programs for HLS using LLMs; 2) LLMs are prone to producing hallucinations, especially in the interdisciplinary field such as HLS that requires both software and hardware knowledge. It's thus necessary to integrate correct repair guidance into the input prompts of LLMs to improve output response; 3) Using LLMs is costly, and existing approaches have not considered cost-effectiveness in applying LLMs to iteratively repair simple errors.

Contrary to the previous work, the proposed framework is designed to comprehend repair guidelines and combine the relevant design tools/scripts with LLMs to repair C/C++ programs for HLS. It takes regular C/C++ programs as input and automatically generates corresponding HLS-C designs while ensuring correct syntax and logical functionality and minimizing the cost of using LLMs.

III. LLM-Driven C/C++ Program Repair for High-Level Synthesis

A. Overview of the Proposed Framework

As shown in Fig. 2, the proposed LLM-driven automated C/C++ program framework for HLS consists of five stages:

1) Preprocessing: An original C/C++ program C_Orig is compiled using the HLS tool, and actual errors are reported. Since the compiler may not be able to detect all errors in a single compilation, the common HLS-incompatible errors, along with the complete program, are fed into LLM to detect other potential errors. An example of the preprocessing stage for detecting and analyzing HLS-incompatible errors is demonstrated in Fig. 10 in Appendix II. After preprocessing, the stage moves to repair with RAG.

2) Repair with RAG: A repair library containing correction templates for HLS is manually established, which is built with human guidance and from the official documentation of HLS tools. The Retrieval-Augmented Generation (RAG) technique is employed to search similar repair templates T_Ref from this library, which can be used to enhance the quality of LLM's prompts while generating a more accurate program C_Ref. If the compilation of C_Ref fails, the error message will be fed back to the LLM for iterative repair. Otherwise, the successfully compiled program is saved as C_Comp.

3) Bit Width Optimization: To optimize the hardware synthesized by the HLS tool, the existing dataset corresponding to the C/C++ program is used to determine the optimal bit width of variables in C_Comp. Then, the repaired C++ program, corresponding task scenarios, and the descriptions of input data from the dataset are fed into the LLM, which automatically generates a script that is able to assign the optimal bit width by calculating the maximum and minimum values of the variables.

4) Equivalence Verification: After synthesizing the HLS-C program into a corresponding RTL code, the C-RTL co-simulation is performed for equivalence verification. Based on the original C++ test benchmarks, the HLS tool automatically compares the RTL simulation results with the C_Orig results to verify the correctness of the synthesized RTL design.

5) PPA Optimization of Circuits: The successfully repaired HLS-C programs are collected into a potential list for further optimization. Critical code segments with large area, high power, and high latency are identified by the HLS tool. The LLM is then used to optimize these key segments by adding/tuning pragmas with the guidance of the HLS official manual. This framework integrates automated interaction between LLM and HLS tools, thereby greatly improving the reliability and efficiency of the repair and optimization process.

An example of repairing a regular C++-based breadth-first search (BFS) algorithm is provided in Appendix I to illustrate the proposed framework. While stages *1)* and *4)* above are straightforward, the key stages *2)*, *3)*, and *5)* above are explained in detail in the following subsections.

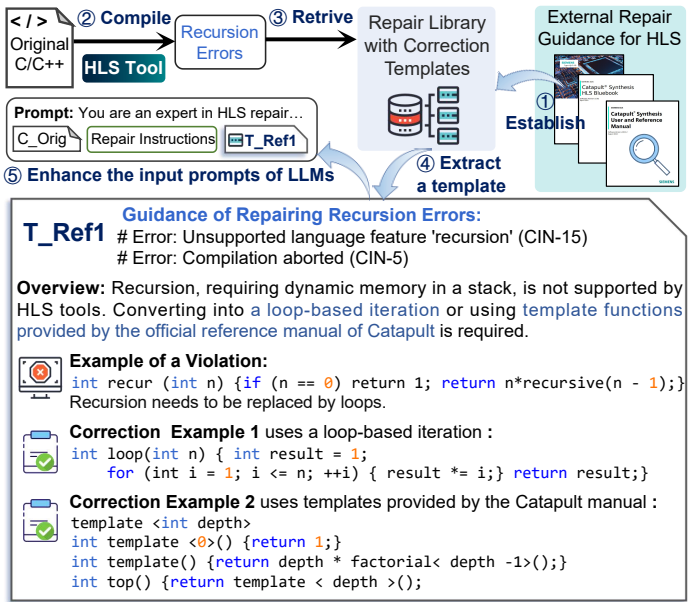

Fig. 3. Example of the LLM using RAG to repair recursion errors.

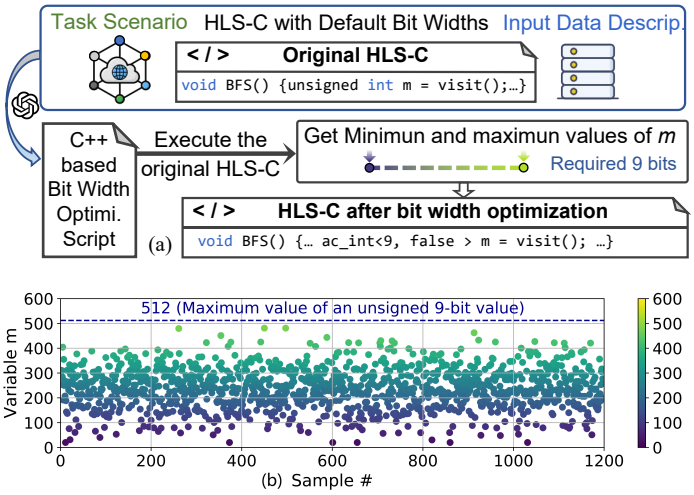

Fig. 4. (a) Example of the bit width optimization scheme; (b) Distribution of 1200 samples of the variable 'm' from the real-world BFS task. This example illustrates that the C++-based script optimizes the bit width of the variable 'm', which needs only 9-bit instead of the default 32-bit of the *int* type.

B. Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique for enhancing the LLM's capability by incorporating external expert guidance through a retriever. Employing RAG to the LLM for repairing HLS incompatibility errors involves the following stages: ① An external repair library containing correction templates is created. Each template integrates the error message from the compiler log, a violation example, corresponding guidance context, and correction examples extracted from the manual of the HLS tool. ② After obtaining the compiler error log, a similarity search mechanism, such as a sentence transformer [28], is employed to retrieve the corresponding correction template from the repair library. ③ Once the correction template with the highest similarity is retrieved, it is extracted and used as part of the prompts provided to the LLM.

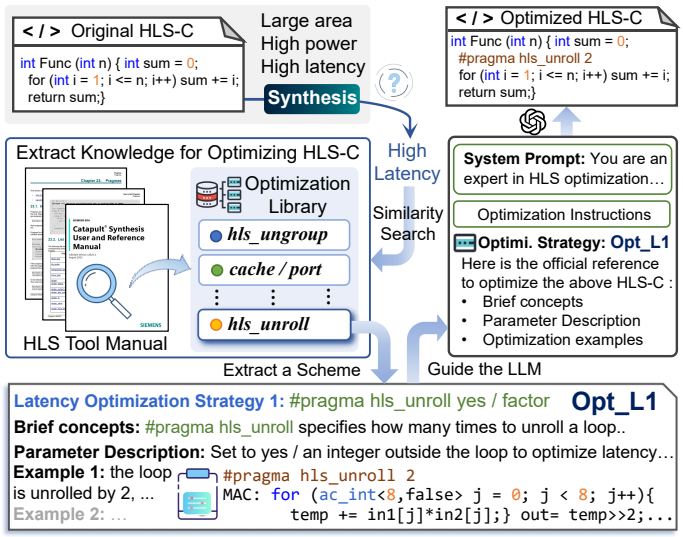

Fig. 5. LLM-driven automatic optimization scheme.

Fig. 3 illustrates the workflow of employing RAG for repairing the HLS-incompatible '*recursion*' error. Initially, when the compiler detects a '*recursion*' error, it generates an error log. This log is then used to retrieve the most appropriate correction template in the HLS repair library by the sentence transformer [28]. Once the correction template with the highest similarity is retrieved, it is used as part of the prompts for the LLM. This correction template (T_Ref1 in Fig. 3) includes HLS guidance on how to repair recursion, examples of a violation, and corresponding repaired versions. By integrating this external guidance into the input prompts of the LLM, the output responses are improved significantly. Another example of RAG-based program repair for HLS via the LLM, including the prompts and responses, is demonstrated in Fig. 11 in Appendix II.

C. Bit Width Optimization

In traditional HLS development, developers often define variables as basic data types in C/C++ programs, which often contain higher than the bit width required for the actual task, thus resulting in resource waste. In Fig. 4(a), to reduce bit width for efficient hardware designs, we propose an LLM-driven bit width optimization scheme, which can find the minimum and maximum value of a variable based on a range analysis. This technique sends the C/C++ code, corresponding task scenarios, and the descriptions of input data to the LLM as prompts. The LLM then automatically creates a C++-based bit width optimization program for this task. This created program identifies variable declaration nodes, finds the maximum and minimum values of variables, and automatically adjusts the default data type with the determined optimal bit widths.

Fig. 4(b) illustrates the profiling of the variable value 'm' in a regular C++ program, which follows a normal distribution. The optimization program identifies that 'm' has a minimum value of 0 and a maximum value of 481, so only 9-bit is needed instead of 32-bit. The framework then implements integer variables with determined bit widths with the ac_uint or ac_int

TABLE II
COMPARISON OF THE PROPOSED FRAMEWORK WITH THE TRADITIONAL SCRIPTS AND THE GPT-4 TURBO BASELINE

Type	Benchmark	Traditional [6] [7]	Baseline (Pass Rate %) Compi.	Baseline (Pass Rate %) Simu.	Proposed (Pass Rate %) Compi.	Proposed (Pass Rate %) Simu.	Type	Benchmark	Traditional [6] [7]	Baseline (Pass Rate %) Compi.	Baseline (Pass Rate %) Simu.	Proposed (Pass Rate %) Compi.	Proposed (Pass Rate %) Simu.
T1: Pointer	1: Double Pointer	×/×	60	53.33	73.33	73.33	T5: Boolean Operation	13: Support Vector Mac.	×/×	46.67	46.67	93.33	93.33
	2: Hash Table	✓/✓	93.33	93.33	100	100		14: Fourier Transform	×/×	40	33.33	80	80
	3: Deep Neural Network	×/×	60	60	86.67	80		15: Color Correction	×/×	46.67	46.67	66.67	66.67
T2: Dynamic Array	4: Linear Programming	✓/×	93.33	86.67	100	100	T6: Incomplete Statement	16: Fibonacci Sequence	✓/✓	86.67	80	100	100
	5: Binary Tree	×/×	66.67	60	80	80		17: Cyclic Rotation	×/×	40	40	86.67	73.33
	6: K-Nearest Neighbor	×/×	60	46.67	73.33	66.67		18: AES	×/×	33.33	26.67	60	60
T3: Recursion	7: Linked List	✓/✓	73.33	73.33	93.33	93.33	T7: Unsupport. Struct	19: Data Stream	×/×	66.67	53	73.33	66.67
	8: Depth-First Search	×/×	46.67	40	80	66.67		20: Longest Increa. Path	×/×	60	60	86.67	86.67
	9: Breadth-First Search	×/×	60	60	73.33	73.33		21: Max Points on Line	×/×	46.67	40	80	73.33
T4: Bit Width	10: Edge Detection	×/×	53.33	33.33	73.33	60	T8: Exception-Handling	22: QR Decomposition	×/×	66.67	60	80	80
	11: Greedy Algorithm	×/×	46.67	46.67	86.67	80		23: Dump Filter	×/×	33.33	33.33	66.67	66.67
	12: Bubble Sort	✓/✓	100	100	100	100		24: Turbo Encoder	×/×	73.33	66.67	93.33	93.33

The pass rate is calculated from the results of 15 rounds of HLS simulation.

types provided by the HLS tool. The detailed demonstration of the bit width optimization scheme, including prompts and responses, is shown in Fig. 12 in Appendix II.

D. Joint LLM-Script Repair

Although LLMs have been effective in automatic program repair, using LLMs can be costly, as users are charged by the number of input and output tokens. For example, the GPT-4 Turbo model, one of the most advanced LLMs, charges $0.01 per 1K input tokens and $0.03 per 1K output tokens [16]. A cost-effective LLM-based repair technique should guide and steer the model toward the correct repair with the lowest possible cost, i.e., with the lowest possible number of tokens.

To minimize the repair cost of using LLMs, we first use traditional HLS repair scripts, which run efficiently on the local CPU to repair the HLS-incompatible errors in the regular C/C++ program. The LLM then conducts comprehensive repairs only on the program that has been processed by these scripts. By pre-repairing simple errors using the traditional scripts at the early stage, the repair cost of using LLMs can be effectively reduced.

E. PPA Optimization

Power, Performance, and Area (PPA) are critical metrics in hardware design [26]. To facilitate the automatic HLS optimization, the critical code segments with performance bottlenecks are identified using design analyzers in the HLS tool. Strategies such as adding or modifying pragmas are then applied to these critical code segments to optimize them.

The optimization strategies are identified by an HLS optimization strategy library, which is built by us according to the manual of the HLS tool [27]. As shown in Fig. 5, the optimization strategy information Opt_L1 consists of a brief concept, parameter descriptions, and optimization examples. The most suitable optimization strategies will be matched via a similarity search, such as a sentence transformer [28], and then integrated into the prompts, utilizing the LLM's in-context learning capabilities to generate an optimized program. For example, loop unrolling (*#pragma hls_unroll*) can be added by the LLM to transform a serial iteration of a loop into a parallel execution, which helps reduce circuit latency. This pragma needs to be set to yes or a value less than or equal to the maximum number of loop iterations. In addition, when memory interfaces become limiting factors, the LLM can optimize memory read and write operations by adding dual-port capabilities or increasing cache mechanisms in the HLS-C program. Another example of the area optimization for the hardware design, including prompts and responses, is demonstrated in Fig. 13 in Appendix II.

After optimizing the HLS-C program, circuit optimization strategies such as retiming are further adopted to minimize the clock period and the number of sequential components of the synthesized circuit.

IV. EXPERIMENTAL RESULTS

We demonstrate the results of the proposed LLM-driven framework for HLS across 24 real-world applications in terms of repair pass rate, cost of using LLM, and optimized hardware performance. These benchmarks are from related work [6], [7], [9], [10] with HLS-incompatible errors.

During the evaluation, GPT-4 Turbo Model was used as the LLM via OpenAI APIs [29]. All experiments were conducted with the Catapult HLS Tool on an Intel(R) Xeon(R) Silver 4314 2.40 GHz CPU. The logic circuits in implementing HLS-C programs were synthesized with Synopsys Design Compiler using the Nangate 45nm open-cell library. To demonstrate the repair capability of the proposed framework, each experiment for a particular application was repeated n instances (n = 15). In each instance, the LLM was queried five times to repair the program based on the errors reported by the HLS tool. We calculate the expectation repair pass rate as Pass Rate (%) $= m/n$, where m is the number of successfully repaired instances and n is the number of all the instances.

Table II compares the repair pass rates of the proposed framework with traditional scripts [6] [7] and the baseline to use GPT-4 Turbo directly to repair the benchmarks. The first and second columns of the left and right subtables show the incompatible error types (T1-T8) and their corresponding benchmarks (1-24). The third column shows the compilation

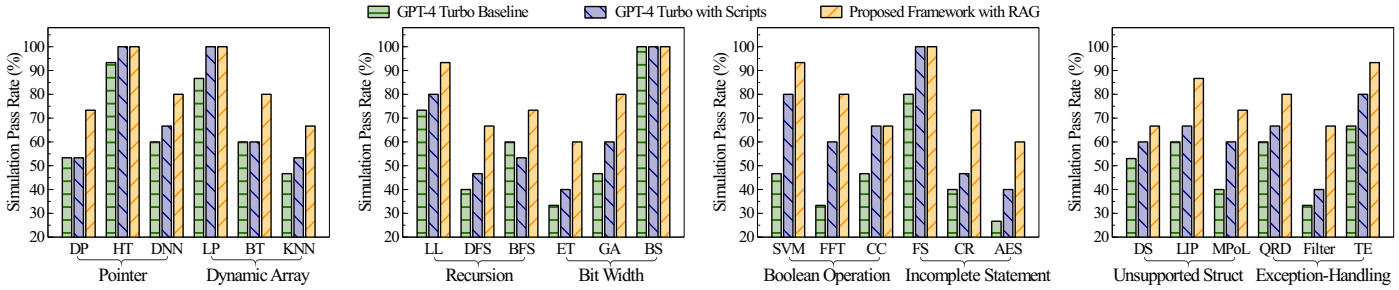

Fig. 6. Comparison of HLS simulation pass rate of the proposed method with GPT-4 Turbo baseline and GPT-4 Turbo with scripts on 24 real-world applications, shown on the x-axes of the figures.

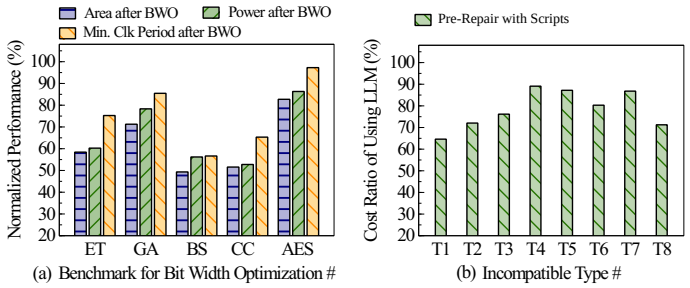

Fig. 7. (a) Ratios of area, power consumption, and minimum clock period using the proposed bit width optimization compared with 32-bit *int* implementation; (b) Cost ratios of the proposed joint LLM-script repair compared with using LLM-driven repair alone.

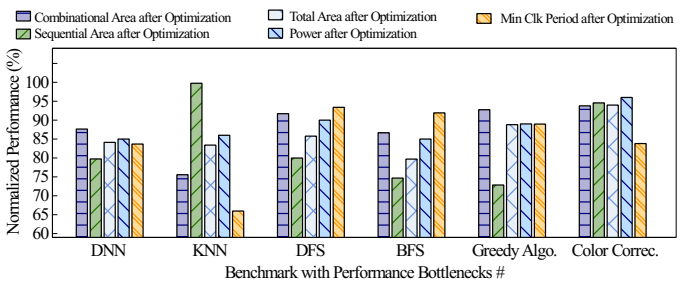

Fig. 8. Ratios of area, power, and minimum clock period after applying LLM-driven optimization.

and simulation pass rates using traditional scripts to repair the programs, with only a few examples passing the simulation test. The fourth, fifth, sixth, and seventh columns represent the compilation and simulation pass rates of the GPT-4 Turbo baseline and our proposed framework, respectively. According to these columns, the proposed LLM-driven framework can improve the pass rate effectively compared with the repair with traditional scripts and the direct application of the LLM.

Fig. 6 compares the simulation pass rate of the proposed framework with those using GPT-4 Turbo directly and GPT joint scripts (ablation experiments without RAG) for program repair, respectively. According to this comparison, the proposed LLM-driven framework outperforms the baseline by achieving an average 23.33% and 13.89% increase in repair pass rate, respectively.

To demonstrate the advantages of the bit width optimization (BWO) scheme, we compare the proposed scheme with the default 32-bit *int* type in terms of area, power, and minimum clock period. We use the actual scenarios and the descriptions of input data from the dataset as prompts to create a static bit width optimization program via the LLM. Fig 7(a) shows the performance ratios of the logic circuit after using bit width optimization. In this comparison, the proposed scheme automatically identified the optimized bit widths for integers, leading to an average reduction of 36.57%, 33.03%, and 29.08% in area, power, and minimum clock period, respectively.

To verify the cost reduction of using LLM, we compared the cost of the proposed joint LLM-Script repair mechanism with that of using only LLM. Among all successful repair programs, we counted the average numbers of input and output tokens associated with the LLM separately. The total costs of using LLMs are calculated from these tokens and their corresponding costs. The normalized average costs to repair each error type [29] are shown in Fig. 7(b). According to this comparison, pre-repairing simple errors with scripts locally can significantly reduce the overall repair costs of only using the LLM by 21.56% on average.

To demonstrate the effectiveness of the proposed LLM-driven PPA optimization strategy on logic circuits, we compared the area, power, and minimum clock period of six benchmarks with performance bottlenecks before and after using such optimization. As shown in Fig. 8, the proposed optimization strategy further achieves an average reduction of 14.02%, 11.50%, and 17.94% in area, power, and minimum clock period, respectively, resulting in more efficient designs.

V. Conclusion

In this paper, we have proposed an LLM-driven program repair framework to solve the incompatible issues of regular C/C++ programs in HLS. A retrieval-augmented generation paradigm is introduced to guide the LLM toward correct repair. An LLM-driven bit width optimization scheme is then applied to identify the optimized bit widths for variables. A joint LLM-script repair mechanism is further used to reduce the cost of using the LLM. The critical code segments are extracted and optimized to generate more efficient HLS designs. Experimental results demonstrate that the proposed LLM-driven repair framework can achieve much higher repair pass rates on 24 real-world applications compared with the traditional scripts and the direct applications of the LLM.

Acknowledgement

This work is funded by the Deutsche Forschungsgemeinschaft *(DFG, German Research Foundation)* – Project-ID 504518248 and supported by TUM International Graduate School of Science and Engineering *(IGSSE)*.

APPENDIX I: AN EXAMPLE OF THE PROPOSED LLM-DRIVEN PROGRAM REPAIR FRAMEWORK FOR HLS

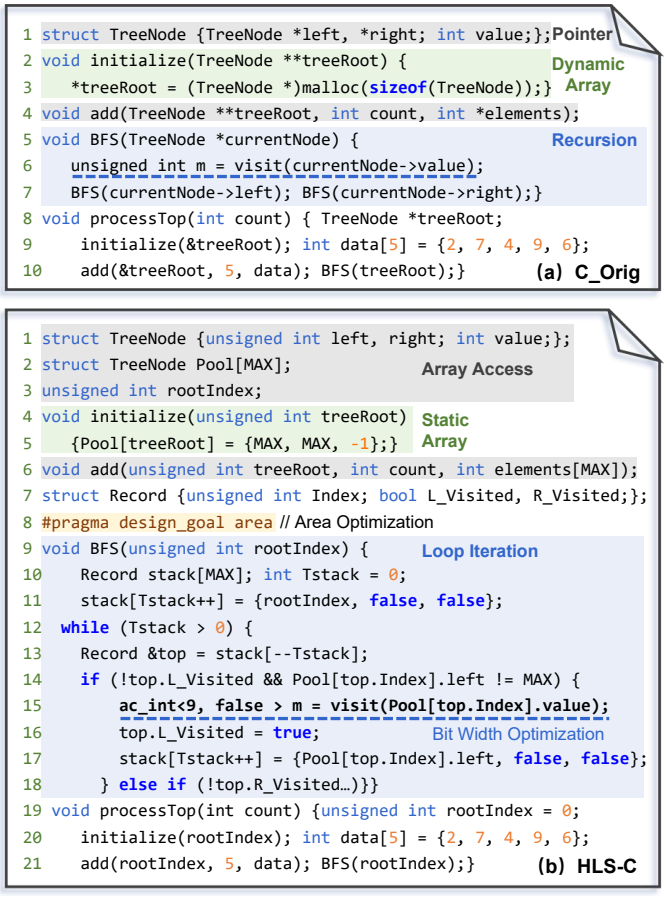

Fig. 9. Working example (Breadth-First Search) of the proposed LLM-driven repair framework. (a) Original C++ program using pointers, dynamic arrays, and recursion; (b) Repaired HLS-C program.

Consider a part of the Breadth-First Search program (in a total of 139 lines in the original code) shown in Fig. 9. This program involves creating a binary tree, initializing the tree structure, adding nodes, and then performing a Breadth-First Search (BFS) to visit and process each node.

In Section IV, we conduct a comprehensive evaluation of all eight types of errors. Here, to clearly illustrate the repair process, we use this example containing three typical incompatible errors, i.e., '*pointer*', '*dynamic array*' and '*recursion*'. The detailed repair process is as follows:

1) Preprocessing: The original C++ program, C_Orig, is compiled using the Catapult HLS tool, and the first compilation reports two incompatible errors: '*pointer*' and '*dynamic array*'. Subsequently, both the common HLS-incompatible error types along with the complete program are fed into the LLM to detect other potential errors, during which the LLM identifies an additional incompatible error, '*recursion*'. Traditional scripts are used to repair the above three errors, but the compilation fails again. The process then moves to the stage of matching error correction templates.

2) Repair with RAG: In this stage, a repair library is built with human guidance and HLS official documentation, which is used to match detected errors with their corresponding correction templates. The matched templates, T_Ref, are then used to enhance the quality of prompts provided to the LLM, guiding the LLM to repair the C_Orig, thereby generating a more accurate HLS-C program. As shown in Fig. 9, LLM implements three main repairs on the original program C_Orig:

① Modify Pointer Access to Array Access: In C_Orig, tree nodes are managed through pointers directly manipulating memory locations (Line 1 in Fig. 9(a): `TreeNode *left, *right`). The HLS-C version modified by the LLM replaces these pointer-based node accesses with array indices (Line 1-3 in Fig. 9(b)). This modification uses a pre-allocated array (Pool) of TreeNode structures, where each node can be accessed via its index rather than pointers.

② Rewrite Dynamic Array into Static Array: In C_Orig, `malloc` is used to create a new TreeNode with dynamic memory allocation (Line 2 and 3 in Fig. 9(a): `malloc(sizeof())`), which is not supported by HLS tools because the synthesized circuit cannot manage data structures with unbounded size. The LLM manages all tree nodes through a predefined array Pool (Line 4 and 5 in Fig. 9(b): `Pool[treeRoot] = {MAX, MAX, -1};`), which avoids dynamic memory allocation.

③ Convert Recursion to Iteration: As shown in lines 5-7 of Fig. 9(a), the original function `BFS` employs recursion to call itself for processing the left and right child nodes of the current node. The LLM replaces the recursion by using an explicit stack data structure, as repaired in lines 9-18 of Fig. 9(b). This stack simulates the automatic management of the function call stack used in recursion. After these repairs, the HLS tool compiles this program again. If the compilation still fails, iterative repair continues until a successful compilation is achieved.

3) Bit Width Optimization: To optimize the design of hardware implementations, the repaired C++ program, along with task scenarios and corresponding descriptions of input data, are fed into the LLM, which automatically generates a C++-based bit width optimization script to find the maximum and minimum values of variables. As shown in line 6 of Fig. 9(a) and the corresponding line 15 in Fig. 9(b), if a variable '`m`' is declared as a 32-bit integer and the optimization script finds that its minimum value is 0 and its maximum is 481, then only 9-bit are needed instead of 32-bit. Then, the previously generated script assigns the optimal bit width by replacing `unsigned int m` with '`ac_int<9, false> m`' in C_Orig.

4) Equivalence Verification: After synthesizing the C++ code to obtain the corresponding hardware RTL model, C-RTL co-simulation is then performed to verify the correctness of the synthesized RTL design.

5) PPA Optimization: Key code segments with large area in the top function `BFS` are identified by the HLS tool. The proposed LLM-driven optimization strategy is then applied to reduce the area of design by adding the pragma such as '`#pragma design_goal area`' at the top of the function `BFS`. Retiming is further used to minimize the clock period the number of sequential components in implementing the synthesized circuit to achieve a more efficient PPA design.

APPENDIX II: EXAMPLES OF PROMPTS AND OUTPUT RESPONSES OF THE LLM AT DIFFERENT STAGES

System Prompt: You are an expert in identifying the High-Level Synthesis (HLS) incompatible errors in the regular C/C++ program.

Problem Description Prompt:

- You are now to preprocess a regular C++ code. Two actual errors are reported by the HLS Tool, which are pointers and dynamic arrays.
- **Your task** is to **detect all other potential HLS-incompatible errors**.
- The other common types of HLS-incompatible errors are summarized as follows. For **each incompatible error**, I provide you **a brief overview** and **the common struct**: (1) Exception…; (2) Recursion…; (3)… (Omitted for brevity due to space limitations)

Here is a typical breadth-first search (BFS) C++ code with HLS-incompatible errors. This code involves creating a binary tree, initializing the tree structure, adding nodes, and performing a BFS to visit and process each node. The complete code is given as follows:

```
struct TreeNode {TreeNode *left, *right; int value;};
void initialize(TreeNode **treeRoot) { //Pointers and DAs
   *treeRoot = (TreeNode *)malloc(sizeof(TreeNode));}
void add(TreeNode **treeRoot, int count, int *elements);
void BFS(TreeNode *currentNode) {
   int m = visit(currentNode->value);
   BFS(currentNode->left); BFS(currentNode->right);}
…(Omitted for brevity due to space limitations)
```

< / > C_Orig

- Please analyze the code carefully and give all potential errors except for pointers and dynamic arrays returned by the HLS tool, no other words.

LLM's Output Response:

Recursion in the BFS function: `BFS(currentNode->left);` …

Fig. 10. Preprocessing: Identify potential HLS-incompatible errors in the regular C/C++ program via the LLM

System Prompt: You are an expert in repairing the High-Level Synthesis (HLS) incompatible errors in the regular C/C++ program.

Problem Description Prompt:

- The **complete C++ code** for HLS is as follows: < / > C_Orig
- Here are the **compilation errors** returned by the compiler of a HLS tool:
  - # Error: No definition for routine 'malloc' (CIN-16)
  - # Error: Unsupported feature 'pointers-to-pointers' (CIN-243)
- Here is the **potential error** detected by LLMs: Recursion
- **Your task** is to repair all the above HLS-incompatible errors in the C++ code **with the guidance of the correction templates T_Ref** extracted from **the manual of the HLS tool**…(Omitted due to space limitations)
- The correction templates to guide repair are shown as follows:

T_Ref1 Guidance of Repairing Pointer Errors:
Overview: Pointers are forbidden in HLS, except… Because it is challenging to give each memory location an identifier… Converting to array access is needed…
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

T_Ref2 Guidance of Repairing Dynamic Array Errors:
Overview: Hardware cannot manage dynamic arrays with unbounded size. Such malloc() and free() functions must be replaced with pre-allocated static arrays.…
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

T_Ref3 Guidance of Repairing Recursion Errors:
Overview: Recursion, requiring dynamic memory in a stack…Converting into a loop-based iteration or using template functions is required…
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

- Please analyze and repair the code carefully. Ensure all modifications do not affect the functionality of the original C++ code… Please provide a completely corrected HLS-compatible C++ code, no other words.

LLM's Output Response:

```
struct TreeNode {unsigned int left, right; int value;};
struct TreeNode Pool[MAX]; ... // Static Array & Array Access
void initialize(unsigned int treeRoot)...
void add(unsigned int treeRoot, int count, int elements[MAX]);
void BFS(unsigned int rootIndex) { // Recursion to Iteration
     Record stack[MAX]; int Tstack = 0;
     stack[Tstack++] = {rootIndex, false, false};...}}}
…(Omitted for brevity due to space limitations)
```

< / > HLS-C

Fig. 11. Repair with RAG: LLM repairs the HLS-incompatible errors according to external guidance from the HLS tool manual.

REFERENCES

[1] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, Jason Cong, "Towards a Comprehensive Benchmark for High-Level Synthesis Targeted to FPGAs," Advances in Neural Information Processing Systems (NeurIPS), 2023.

[2] Qi Sun, Tinghuan Chen, Siting Liu, Jianli Chen, Hao Yu, and Bei Yu, "Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design," ACM Transactions on Design Automation of Electronic Systems (TODAES), 2022.

[3] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong, "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators," ACM Transactions on Design Automation of Electronic Systems (TODAES), 2022.

[4] Nadesh Ramanathan, George A. Constantinides and John Wickerson, "Precise pointer analysis in high-level synthesis," IEEE/ACM International Conference on Field-Programmable Logic and Applications (FPL), 2020.

[5] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, Miryung Kim, "Refactoring for Heterogeneous Computing with FPGA," IEEE/ACM International Conference on Software Engineering (ICSE), 2020.

[6] David B. Thomas, "Synthesisable recursion for C++ HLS tools," IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016.

[7] Zeping Xue, David B. Thomas, "SynADT: Dynamic data structures in high level synthesis," IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), 2016.

[8] David B. Thomas, "Templatised Soft Floating-Point for High-Level Synthesis," IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), 2019.

[9] Hsuan Hsiao, Jason H. Anderson, "Sensei: An area-reduction advisor for FPGA high-level synthesis," IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.

[10] "Leetcode Problem Set," Accessed: 2023. [Online]. Available: https://leetcode.com/problemset/.

[11] "Siemens EDA Catapult Synthesis User and Reference Manual," Accessed: 2023. [Online]. Available: https://support.sw.siemens.com/.

[12] Kangwei Xu, Grace Li Zhang, Ulf Schlichtmann, Bing Li, "Logic Design of Neural Networks for High-Throughput and Low-Power Applications," IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.

[13] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, Yingyan Lin, "GPT4AIGChip: Towards Next-Generation AI Accelerator Design Automation via Large Language Models," IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2023.

[14] Jason Blocklove, Siddharth Garg, Ramesh Karri, Hammond Pearce, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," IEEE Workshop on Machine Learning for CAD (MLCAD), 2023.

[15] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, Ramesh Karri, "AutoChip: Automating HDL Generation Using LLM Feedback," arXiv preprint: 2311.04887, 2023.

[16] "GPT-4 pricing for OpenAI API," Accessed: 2024. [Online]. Available: https://openai.com/api/pricing/.

[17] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023.

[18] Yao Lu, Shang Liu, Qijun Zhang, Zhiyao Xie, "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model,"

System Prompt: You are an expert in optimizing the bit width of variables in the High-Level Synthesis compatible C/C++ (HLS-C) programs.

Problem Description Prompt:

- The **complete HLS-C code** is as follows:

```
</> bfs.cpp  unsigned int m = visit(Pool[top.Index].value); ...
```

Scenario: This algorithm is designed to manage the books in a library. The first task of this algorithm is to count the number of borrowings of each type of book in the past three years to decide whether to continue purchasing this type of book…

Input data description: Identifier No. & Current quantity of each book Data type & format: a *.csv* file, *n* rows and *m* columns with explanations…

- **Your task** is to identify variable declarations in HLS-C and write a C++-based bit width optimization script, which can automatically replace the default data type with the data type provided by the HLS tool.
- The **description and usage of each data type** from **the manual of the Catapult HLS tool** is as follows**:**

B_Ref1 **Usage Guidance of Unsigned integer 'ac_uint':**
The HLS-C unsigned integer data types are declared as:
`ac_int<W, false> x;` where: W = Bit width, 0 <= x <= 2^{W-1}
`#include <ac_int.h>` should be added as a header file for HLS-C
Example before adjust.: int…; **Example** after adjust.: ac_uint...;

- Please refer to the usage guidance of data types and write a C++-based bit width optimization script, which can automatically run the HLS-C program, find the maximum and minimum values of variables, and adjust to the specified bit widths…(Omitted due to space limitations)

LLM's Output Response:

```
#include "bfs.cpp"  </> C++-based bit width optimization script
void initialize_data_csv(){}
int calculate_min_bit_width(){}
void replace_with_HLS_width(){}
int main() {   std::ifstream bfsFile("bfs.cpp");
   std::ofstream modifiedBfsFile("bfs_bwo.cpp");
   initialize_data_csv("book.csv"); BFS();
   int bit_width = calculate_min_bit_width ();
   replace_with_HLS_width();...} (Omitted due to space limitations)
```

Fig. 12. Bit width optimization: A script is created via the LLM to adjust the default data type with the determined bit widths.

System Prompt: You are an expert in power, performance, and area (PPA) optimization of the synthesized circuit in High-Level Synthesis (HLS).

Problem Description Prompt:

- The **complete HLS-C code** and the **.tcl script** for HLS are as follows:

```
struct TreeNode {unsigned int left, right; int value;};
struct TreeNode Pool[MAX]; ...                    </> HLS-C
void initialize(unsigned int treeRoot)...
void add(unsigned int treeRoot, int count, int elements[MAX]);
void BFS(unsigned int rootIndex) {
     Record stack[MAX]; int Tstack = 0;
     stack[Tstack++] = {rootIndex, false, false};...}}}
...(Omitted for brevity due to space limitations)
```

```
solution file add {kernel_path} -type C++ ...        run.tcl
go new\n go analyze\n
go compile...
```

- Here is the **performance bottleneck** from the tool report: Large Area.
- **Your task** is to optimize both the HLS-C program and the Tcl commands **with the guidance of the optimization strategies** extracted from **the manual of the Catapult HLS tool.**
- The **optimization strategies** to optimize the area are shown as follows:

Area Optimization Strategy 1: #pragma design_goal area **Opt_A1**
Brief concepts: Determine if the tool should optimize for smallest area ...
Parameter Description: Set design_goal to 'area' to minimize the circuit area...
Example 1:

```
#pragma design_goal area
void test (ac_int<16,false> in, ac_int<32,false> &out){}...
```

Area Optimization Strategy 2: ARRAY_INDEX_OPTIMIZATION **Opt_A2**
Brief concepts: Determine if additional optimizations are performed on arrays that are not a power of two. These optimizations include better support for word width, interleave, and memory access merging.
Parameter Description: Set this command to true to minimize the area...
Example 1: directive set ARRAY_INDEX_OPTIMIZATION true in run.tcl

Area Optimization Strategy 3: #pragma area_goal n... **Opt_A3**
Brief concepts: Specify a size goal for area optimization; the tool will stop optimizing the area when the total area is less than / equal to the area_goal.
Parameter Description: Valid setting is a positive integer or floating-point value.
Example 1: `#pragma area_goal 1250.5`

```
     void test (ac_int<7,true> in, ac_int<16,true> &out){}...
```

`.....`(Other strategies are omitted due to space limitations)

- Please refer to the usage guidance of area optimization strategies and add/tune the parameters to optimize the HLS-C program carefully...

LLM's Output Response:

```
struct TreeNode {unsigned int left, right; int value;};
struct TreeNode Pool[MAX]; ...          </> Optimized HLS-C
void initialize(unsigned int treeRoot)...
void add(unsigned int treeRoot, int count, int elements[MAX]);
#pragma design_goal area
void BFS(unsigned int rootIndex) {
     Record stack[MAX]; int Tstack = 0;
     stack[Tstack++] = {rootIndex, false, false};...}}}
...
```

```
...# Branch the solution and set directive value to true
go new\n go analyze\n
directive set ARRAY_INDEX_OPTIMIZATION true          run.tcl
go compile... (Omitted for brevity due to space limitations)
```

Fig. 13. PPA Optimization via the LLM.

IEEE/ACM Asia and South Pacific Design Automation Conference, 2024.

[19] Long Ouyang, Jeff Wu, Xu Jiang, Ryan Lowe et al., "Training language models to follow instructions with human feedback," Advances in Neural Information Processing Systems (NeurIPS), 2022.

[20] Yun-Da Tsai, Mingjie Liu, Haoxing Ren, "Automatically Fixing RTL Syntax Errors with Large Language Model," IEEE/ACM Design Automation Conference (DAC), 2024.

[21] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," IEEE Symposium on Security and Privacy, 2023

[22] Harshit Joshi, Jose Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, Gust Verbruggen, "Repair is nearly generation: multilingual program repair with LLMs," Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI), 2023.

[23] He Ye, Martin Monperrus, "ITER: Iterative Neural Repair for Multi-Location Patches," IEEE/ACM International Conference on Software Engineering (ICSE), 2024.

[24] Chunqiu Steven Xia, Yuxiang Wei, Lingming Zhang, "Automated program repair in the era of large pre-trained language models," IEEE/ACM International Conference on Software Engineering (ICSE), 2023.

[25] Nicholas V. Giamblanco, Jason H. Anderson, "A Dynamic Memory Allocation Library for High-Level Synthesis," IEEE/ACM International Conference on Field-Programmable Logic and Applications (FPL), 2019.

[26] Tinghuan Chen, Grace Li Zhang, Bei Yu, Bing Li, Ulf Schlichtmann, "Machine Learning in Advanced IC Design: A Methodological Survey," IEEE Design & Test, 2023.

[27] "Siemens EDA Catapult High-Level Synthesis Tools," Accessed: 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/.

[28] Nils Reimers, Iryna Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," ACL Empirical Methods in Natural Language Processing (EMNLP), 2019.

[29] "GPT-4 Turbo through OpenAI API," Accessed: 2024. [Online]. Available: https://platform.openai.com/.