PLPilot: Benchmark an Automated Programming Language Design Framework Enabled by Large Language Models

Kaiyan Chang[1,3] **Kun Wang**[1,2,3] **Mengdi Wang**[1,3] **Shengwen Liang**[1] **Yinhe Han**[1]
Huawei Li[1] **Xiaowei Li**[1] **Ying Wang**[1]
[1]Institute of Computing Technology, Chinese Academy of Science
[2]Hangzhou Institute for Advanced Study
[3]University of Chinese Academy of Science
Beijing, China 100190
changkaiyan@live.com,wangkun22@mails.ucas.ac.cn
{wangmengdi17s,liangshengwen,yinhes,lihuawei,lxw,wangying2009}@ict.ac.cn

Abstract

The design of new programming languages traditionally requires expertise across syntax and semantics. Recently, large language models(LLMs) have provided unprecedented power in the code generation field, which has the potential to revolutionize the current programming language design stack, including automating writing passes and formally defining a programming language's semantics and syntax. However, there is yet no framework to leverage LLMs to support programming language design. We propose an programming language design framework enabled by large language models, which decouples every part in the programming language design process into a form acceptable by LLMs. We then propose a set of benchmarks on LLM-based programming language tasks. We evaluate this framework on eight decoupled programming language design stages, which shows great productivity improvements over manually designed languages.

1 Introduction

The design and implementation of new programming languages is a complex task requiring expertise across the full stack from syntax to semantics. Recent advances in large language models (LLMs) open up new possibilities for automating aspects of programming language creation. These advances promise to democratize programming language creation and enable new languages tailored to emerging domains.

Researchers in programming language design have long sought to automate aspects of programming language design to enhance software quality and engineer productivity. Recent work has explored using automated tools to improve program safety, performance, and portability while reducing manual effort. For example, frameworks like Soot[1] optimize Java byte code to facilitate program analysis and verification. Projects such as MLIR[2] provide unified intermediate representations to liberate developers from creating lexical and syntactic parsers. Other systems including Ansor[3], AutoTVM[4], and Pluto[5] apply rule-based methods or small machine learning models to automatically optimize code for better performance.

While these compiler automation efforts have shown promise, their reliance on narrow AI models and rule-based methods limits generalizability and fails to profoundly advance human productivity. Tasks such as program analysis still require manually crafting thousands of lines of compiler passes. Meanwhile, pioneering works on code generation LLMs like CodeX[6] and StarCoder[7] have

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

focused solely on synthesizing code snippets. The extreme importance of key compiler tasks like optimization and analysis remains untapped in Code LLM research so far.

To solve these challenges, PLPilot provides a workflow for rapidly prototyping and optimizing on programming language. Designers can provide high-level natural language prompts to specify desired language features. PLPilot utilizes LLMs to automatically generate language syntax. The system handles lower-level details of validation, analysis, and optimizing code generation, while allowing designers to focus on high-level semantics. **Our initial results pave the way for realizing this vision of democratized, AI-assisted programming language implementation.**

The contributions are listed below:

- **We demonstrate the potential of LLMs for automating programming language design.** By benchmarking LLMs on decoupled tasks across eight key stages, we show these models' aptitude for this domain. Our work points the way towards integrating LLMs into the language development workflow.
- **Formallize programming language design problem to LLM accepted form.** We formalize end-to-end language design into distinct, LLM-amenable modules with clear inputs/outputs. This decomposition enables leveraging LLMs for automated generation.
- **Open Source LLM-based Programming Language Framework Benchmarks.** We propose the first open-source, full-stack benchmark for LLM-based programming language design. This enables evaluating LLM capabilities across eight compiler pipeline stages, demonstrating the potential to automate key tasks.
- **Open Source unified LLM driven Programming Language Framework.** We propose PLPilot, a novel unified LLM-driven programming language design framework, which has the capacity to assist people to design programming language.

2 Background and Motivation

2.1 Programming Language related Large Language Model

The emergence of large language models (LLMs) has enabled the programming language community to leverage AI to alleviate manual burdens. Much research has focused on applying LLMs to code generation, with projects like Jigsaw[8], CHOOSE[9], and Synchromesh[10] demonstrating strong results. Benchmarking popular LLMs like CodeX and GPT-J highlights their capacity for generating code[11] and even mathematical code[12]. Using multi-turn training can further enhance code LLM abilities[13]. To handle complex prompts, LLMs can break down problems before generating solutions[14]. For reverse engineering, LLMs generate illustrative code examples[15]. While the above focuses on code generation, LLMs also assist in complementary tasks like generating programs to train compiler models[16] and optimizing LLVM[17] assembly code[18]. LLMs have been applied to program repair as well[19]. However, these existing works focus more on isolated stages of the programming language development process, overlooking the end-to-end automation needs of the programming language design community.

2.2 Motivation

Challenge 1: Writing Compiler Pass requires great manual effort. Expert developers must meticulously hand-engineer the intricate program transformations underpinning performance gains and safety guarantees. For example, a production-grade optimizer may contain thousands of lines of code just to perform one dataflow analysis and associated rewrites. Each new analysis requires painstaking work to track values, build data structures, and codify rewrite rules.

Challenge 2: Designing programming language requires great formal skill, prevent engineers to join to the community. The significant formal methods expertise required in programming language design creates a high barrier to participation for many software engineers. Programming languages are built upon intricate theoretical foundations spanning type theory, semantics, logic, and category theory. Designers must construct rigorous frameworks governing syntax, type systems, grammars, and semantics. This level of formality prevents many technically proficient developers from contributing to the programming languages community.

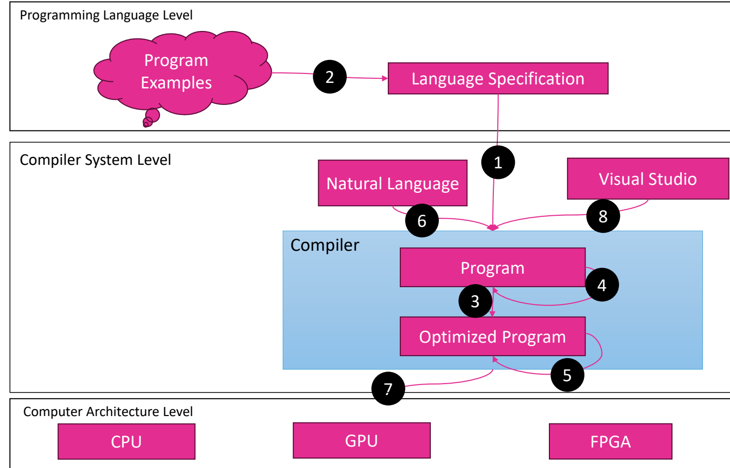

Figure 1: PLPilot Overview.

3 Decoupling the Programming Language Design Process

To make programing language's design merge into current LLM, we decouple the whole programming language design process into several part, which explicitly provide input and output to LLMs. As shown in Fig. 1, the whole programming language design stack transforms several new programming language examples to language specification then assist compiler designers to automatic optimize program. For the more detailed implementation, We require readers use our open source website `https://github.com/changkaiyan/plpilot` to access benchmark. The formal forms of this framework are shown in Appendix Tab. 1.

3.1 Programming Language Level Design

Traditionally, when designing a new programming language, developers first prototype example programs to explore the desired functionality. They then manually abstract syntax and semantics from these examples before finally implementing the compiler according to the specifications.

❶ Transform Specification to Compiler Parser Traditionally, developers must manually transform language specifications into lexer and parser inputs in a grammar definition format like `.g4`. This requires familiarity with grammar file formats. PLPilot automates this process by taking language specifications and directly generating the lexer and parser frontend code in Python. This bypasses the need for manual grammar implementation.

❷ Transform Program Examples to Specification Before creating a new programming language, designers prototype by writing example programs representing desired functionality. They then manually analyze these to derive formal syntax and semantics. PLPilot automates this process. By providing example programs as input, the system leverages an LLM to directly generate formal syntax and semantics, bypassing manual analysis.

3.2 Compiler System Level Design

To assist compiler system level design, we separate this stage into two parts to assist IDE development and compiler development.

3.2.1 Integrated Design Environment Assistant

In Integrated Design Environment such as vscode and intellij, the IDE often have program correction and natural language programming function. **❻Program synthesis** is a common technique in IDE. LLM-era applications such as cursor, often provide an interface to accept natural language description and output program. Therefore, PLPilot provide such function. Moreover, cuurent

IDEs use **❸program correction** to correct wrong syntax. PLPilot can accept a program written by programmer with flaws and remove these flaws.

3.2.2 Compiler Design Assistant

To enable agile compiler development, we leverage LLMs to automate program analysis and modification without manually implemented passes.

❹Program Analysis Classical analysis requires implementing custom passes to analyze ASTs, adding developer effort. We find LLMs can generalize for this task. By providing the analysis type and program, PLPilot's LLM outputs results without passes.

❺Program Verification Verifying programs involves inserting logic expressions. In PLPilot, LLMs can symbolically check if a program satisfies given expressions, avoiding manual verification.

❸Program Optimization Compilers optimize by analyzing ASTs and applying rewrite rules. PLPilot demonstrates LLMs can replicate this - given rules and unoptimized code, the LLM directly transforms the program without human optimization.

3.3 Computer Architecture Level

In computer architecture level, researchers often setup a **❼cost model** to measure the program execution latency. LLMs offer zero-shot alternative to learn these cost models. However, LLMs cannot directly output a program's latency. We show PLPilot can compare two programs' latency without training, by simply inputting code snippets and outputting their relative performance. This exemplifies LLMs' potential for accelerating architecture-level optimizations.

4 Implementation

PLPilot is implemented in Python with OpenAI library. It uses GPT-3.5 as the base large language model, while it can also generalize to other large language models. The inner structure of PLPilot consists a set of prompt manager. Every prompt manager consists two partitions act as system prompt and user prompt respectively. System prompts are defined to provide the input and output format and the task functions. User prompts are defined to focus on the input content. For example, in the program embedding task, the system prompt is "Users will give you a program. You are an assistant to extract its information follows the provided syntax format and output the extract information. The syntax format element enclosed by <>. The output should in a single line.", the user prompts are "Embedding syntax format" and "Input program". System prompts act as the control flow to give user constraint, while user prompts act as the data flow to give LLM the program which need to process.

5 Evaluation

Due to page limits,we have included the detailed evaluation in the appendix. The results of our evaluation affirm the proficient performance of PLPilot across the majority of programming language design tasks.

6 Conclusions and Future Work

Traditionally, creating programming languages requires extensive manual effort. In this paper, we proposed a decoupled language design process powered by large language models to automate key aspects of development. Our framework demonstrates the potential for LLMs to accelerate programming language design. By providing a unified, LLM-driven framework, our approach reduces manual effort and makes language design more accessible.However, the current PLPilot system has limitations in accurately generating complete language specifications and implementations. Going forward, we aim to enhance PLPilot's capabilities by expanding the diversity of programming language case studies. This includes conducting more case studies on both hardware and software domain-specific languages to further improve the system.

References

[1] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[2] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[3] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[4] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[5] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[6] OpenAI. Openai codex, 2023. `https://openai.com/blog/openai-codex`, Last accessed on 2023-09-29.

[7] Leandro von Werra, Loubna Ben Al. Starcoder: A state-of-the-art llm for code, 2023. `https://huggingface.co/blog/starcoder`, Last accessed on 2023-09-29.

[8] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[9] Dominik Sobania, Martin Briesch, and Franz Rothlauf. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pages 1019–1027, 2022.

[10] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

[11] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[12] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[13] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[14] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*, 2023.

[15] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*, pages 37–39, 2022.

[16] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. Benchdirect: A directed language model for compiler benchmarks. *arXiv preprint arXiv:2303.01557*, 2023.

[17] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[18] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[19] Nigar M Shafiq Surameery and Mohammed Y Shakor. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, 3(01):17–22, 2023.

A Structure

Tab. 1 shows formal expression of LLM driven compiler tasks.

Table 1: Formal expression of LLM driven compiler tasks.

Formal Expression	**Stage Description**
$LLM(Spec) \rightarrow Comp : prog_0 \rightarrow prog_1$	1-Compiler Design
$LLM(prog_0, prog_1, \cdots) \rightarrow Spec$	2-Programming Language Design
$LLM(prog_0) \rightarrow prog_1$	3-Program Optimization
$LLM(prog_0) \rightarrow attribute$	4-Program Analysis
$LLM(prog, attribute) \rightarrow yes/no$	5-Program Verification
$LLM(prompt) \rightarrow prog$	6-Natural Language Compile
$LLM(prog_0, prog_1, \cdots) \rightarrow index$	7-Program Cost Comparasion
$LLM(prog_0) \rightarrow prog_1$	8-Program Correction

B Benchmarks

Tab. 2, 7, 5, 8, 4, 6, 3, 9 show the current benchmarks under different stages under PLPilot.

Table 2: Compiler Design Benchmark (Specification To Compiler).

Workload	**Description**
C	A tiny C language specification, an interpreter.
Python	A subset Python language specification, an interpreter.
Pytorch	A subset Pytorch DSL specification.
State Machine	A customized state machine manipulate language.
SQL	A subset SQL specification, compile to C.
Graph	A customized Graph computing language.
Java	A subset Java language specification, an interpreter.
LLVM	A subset C language specification compile to llvm.
RISC-V	A subset RISC-V specification , output an interpreter.
CPU-ISA	A customized CPU-ISA, an interpreter.

C Case Study

Tab. 10 shows input and output of every stage in the whole programming language design framework. Tab. 2, 7, 5, 6, 4,9, 8, 3 show several case studies in PLPilot, which provide a comprehensive view.

Table 3: Programming Language Design Benchmark (Program To Specification).

Workload	Description
C	A subset C language specification.
Python	A subset Python language specification.
Pytorch	A subset Pytorch DSL specification.
State Machine	A customized state machine manipulate language in python.
SQL	A subset SQL specification.
Graph	A customized Graph computing language.
Java	A subset Java language specification.
LLVM	A subset LLVM IR.
RISC-V	A subset RISC-V specification.
CPU-ISA	A customized CPU-ISA.

Table 4: Program Optimization Benchmark.

Workload	Description
Matmul-C	matrix multiply in C.
FC+bias	Fully connect layer add bias in C.
Vector mul	Vector multiply in Python.
Matmul-Cuda	A cuda implementation of matmul.
Convolution	Convolution in C.
Attention	Implement attention operator in C.
Graph	A graph aggregate in Python.
Tensorize	A gemm for vector-based TPU in C.
Matmul-thread	A matrix multiply in multi-CPU system.
Image crop	Image crop on cuda devices.

Table 5: Program Analysis Benchmark.

Workload	Description
C-DC	find dead code.
C-CE	find common expression.
C-Array	find array size beyond scape.
C-OverSize	find number size beyond scape.
BitMismatch	find wrong bit size(FPGA).
Memleak	Find memory leakage.
Pointernull	Find null pointer.
Wrongpointer	Find wrong pointer type.
CUDA-thread	Find thread overflow(CUDA).
CUDA-sync	Find sync error.(CUDA)

Table 6: Program Verification Benchmark.

Workload	Description
Undef var	Undefined variable in C++.
Type error	Undefine type error in C++.
Loop overflow	Out of bound error in C++.
Number over	Out of variable number overflow
Number bound	Two number add result <a number.
If bound	Write a logic with /\and ->to check.
Number bound 2	Two numbers has precondition and postcondition.
Mixed bound	A mixed constraint.
Memory bound	Dynamic memory should less than a value.
Type error 2	Customize type mismatch in C++.

Table 7: Natural Language Benchmark.

Workload	Description
Linux button driver	Synthesis button driver based on Linux driver DSL.
Linux LED driver	Synthesis LED driver based on Linux driver DSL.
CPU on FPGA	A simple CPU on FPGA.
Vecmul on FPGA	Apply matmul on FPGA
Graph DSL	Synthesis graph DSL to python.
Parser	Normal expression analyze in python.
DFS	DFS in python.
BFS	BFS in python.
Draw	Call library in python.
Deep learning	Synthesis deep learning DSL in python.

Table 8: Program Cost Benchmark.

Workload	Description
Matmul-C	matrix multiply in C.
FC+bias	Fully connect layer add bias in C.
Vector mul	Vector multiply in Python.
Matmul-Cuda	A cuda implementation of matmul.
Convolution	Convolution in C.
Attention	Implement attention operator in C.
Graph	A graph aggregate in Python.
Tensorize	A gemm for vector-based TPU.
Matmul-thread	A matrix multiply in multi-CPU system.
Image crop	Image crop on cuda devices.

Table 9: Program Correction Benchmark.

Workload	Description
VNC	Variable name correction
GC	Grammar correction
LBC	Logic bug correction
API	API use correction.
REC	Code reconstruction.
VNC-DSL	Variable name correction in DSL
GC-DSL	Grammar correction in DSL
LBC-DSL	Logic bug correction in DSL
API-DSL	API use correction in DSL.
REC-DSL	Code reconstruction in DSL.

Program Example

```
for(i = 1..10){
  for(j = 1..10){
    A[i][j]=B[j];
  }
}
```

Specification

```
program = { statement } ;
statement = assignment | loop ;
assignment = identifier , "=" , expression ;
loop = "for" , identifier , "in" , range , "{" ,
program , "}" ;
range = number , ".." , number ;
expression = identifier | number ;
identifier = letter , { letter | digit } ;
number = digit , { digit } ;
letter = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" ;
digit = "0" | "1" | ... | "9" ;
```

Figure 2: A Case of PLPilot-based translating program example to language specification.

Table 10: Detailed Stage Design

Id	Type	Task	Description	Input	Output	LLM Capacity
1	Compiler	Compiler AutoDesign	From programming language specification to compiler	EBNF, Test Case	Compiler components	LLM abstract capacity
2	Programming Language	Programming language syntax, semantic design	Extract the syntax, semantic and type system of program	Program example	EBNF, Big-step Semantic	PL syntax abstract capacity
3	Program Optimization	Reduce program latency	Change program structure to reduce program latency	Raw program	Optimized program	Optimize capacity
4	Program Analysis	Analyze program semantic attribute	Analyze program to its specific attribute	Raw program	If has common sub-expression, inactive variables Etc.	Inductive capacity
5	Program Verification	Verify if program follow the natural language specification	Verify if program can match the verifying statement	Program, verification statement	Match or Mismatch	Fuzzy logic verification capacity
6	Program Synthesis	Natural Language Compiling	Translate natural language to target programming language	Natural language description	program	GPT abstract understanding capacity
7	Program cost comparasion	Evaluate program performance	Evaluate programs' relative latency and performance measurement	Program 0, Program 1	Cost rank	Comparasion capacity
8	Program correction	Language syntax and semantic correction	Correct syntax and semantic error in a program	Raw program	Right program	Formal understanding capacity

Specification	Compiler

state_transition = "state" state_id transition_operator action_id ">" state_id
state_machine = state_transition { state_transition }
	class StateTransition:
	    def __init__(*self*, *state_id*, *transition_operator*, *action_id*, *next_state_id*):
	        self.state_id = state_id
	        self.transition_operator = transition_operator
	        self.action_id = action_id
	        self.next_state_id = next_state_id
	
	class StateMachine:
	    def __init__(*self*, *transitions*):
	        self.transitions = transitions
	
	    def generate_code(*self*):
	        code = ""
	        for transition *in* self.transitions:
	            code += f"if state == '{transition.state_id}' and action == '{transition.action_id}':\n"
	            code += f" state = '{transition.next_state_id}'\n"
	        return code

Figure 3: A Case of PLPilot-based translating language specification to compiler design.

Program	Optimized Program (Unroll)	Optimized Program (locality)

for(i=0;i<10;i++){
    for(j=0;j<10;j++){
        for(k=0;k<20;k++){
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
	for(i=0;i<10;i++){
	    for(k=0;k<20;k++){
	        for(j=0;j<10;j+=4){
	            C[i][j] += A[i][k]*B[k][j];
	            C[i][j+1] += A[i][k]*B[k][j+1];
	            C[i][j+2] += A[i][k]*B[k][j+2];
	            C[i][j+3] += A[i][k]*B[k][j+3];
	        }
	    }
	}
		for(i=0;i<10;i++){
		    for(k=0;k<20;k++){
		        for(j=0;j<10;j++){
		            C[i][j] += A[i][k]*B[k][j];
		        }
		    }
		}

Figure 4: A Case of PLPilot-based program optimization.

Program	Corrected Version

int mian(int argc, char**argv){
  int sum;
  for(int i=0;i<argc;i++){
   sum+=argv[i];
  }
  return 0;
}
	int main(int *argc*, char** *argv*) {
	  int sum = 0;
	  for (int i = 1; i < argc; i++) {
	    sum += atoi(argv[i]);
	  }
	  return 0;
	}

Figure 5: A Case of PLPilot-based program correction.

Program 0	Program 1	Cost Compare

for(i=0;i<10;i++){
  for(k=0;k<20;k++){
    for(j=0;j<10;j+=4){
      C[i][j] += A[i][k]*B[k][j];
      C[i][j+1] += A[i][k]*B[k][j+1];
      C[i][j+2] += A[i][k]*B[k][j+2];
      C[i][j+3] += A[i][k]*B[k][j+3];
    }
  }
}
	for(i=0;i<10;i++){
	  for(j=0;j<10;j++){
	    for(k=0;k<20;k++){
	      C[i][j] += A[i][k]*B[k][j];
	    }
	  }
	}
		Based on the given program latency,
		it seems that the second program in p
		rogram would be better.
		The reason for this is that the second
		program has a loop that increments j
		by 4,
		allowing for more efficient memory a
		ccess.

Figure 6: A Case of PLPilot-based program cost comparasion.

Program	Analyzed Result (Dead Code)

for(int i=0;i<100;i++){
  for(int j=0;j<10;j++){
  A[i]=B[i+1];
  B[i]=C[i-1];
}
}
	The reason for this is that the inner for lo
	op does not seem to be doing anything
	with the variable j.
	This means that the loop will simply itera
	te through 10
	times without actually performing any ac
	tions.
	
	As a result, the line B[i]=C[i-
	1]; will be executed 100 times,
	regardless of the value of j.
	This suggests that the inner loop is unne
	cessary and could be removed without a
	ffecting the functionality of the program.

Figure 7: A Case of PLPilot-based program analysis.

Program needs to be verified

```
int func(int a){
   int b;
if(a > 0){
 b = a;
}
<< b < 0 >>
}
```

Verification Result

```
#include <iostream>

bool isSatisfied = false;

int func(int a) {
   int b;
   if (a > 0) {
      b = a;
   }
   if (b < 0) {
      isSatisfied = true;
   }
   return b;
}
```

```
int main() {
   int a = 5;
   int result = func(a);
   std::cout << "Result:
" << result << std::endl;
   if (isSatisfied) {
      std::cout <<
"true" << std::endl;
   }
   return 0;
}
```

Figure 8: A Case of PLPilot-based program verification.

Input: give me an example Linux button driver

```
/* Example Linux button driver */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>

#define BUTTON_PIN 17

static irqreturn_t button_isr(int irq,
void *data)
{
   printk(KERN_INFO "Button
pressed\n");
   return IRQ_HANDLED;
}
```

```
static int __init button_init(void)
{
   int ret;

   ret = gpio_request(BUTTON_PIN,
"button");
   if (ret) {
      printk(KERN_ERR "Failed to
request GPIO pin %d\n", BUTTON_PIN);
      return ret;
   }

   ret =
gpio_direction_input(BUTTON_PIN);
   if (ret) {
      printk(KERN_ERR "Failed to set
GPIO pin %d as input\n", BUTTON_PIN);
      gpio_free(BUTTON_PIN);
      return ret;
   }

   ret = gpio_to_irq(BUTTON_PIN);
   if (ret < 0) {
      printk(KERN_ERR "Failed to get
IRQ for GPIO pin %d\n", BUTTON_PIN);
      gpio_free(BUTTON_PIN);
      return ret;
   }
```

```
 ret = request_irq(ret, button_isr,
IRQF_TRIGGER_RISING, "button",
NULL);
   if (ret) {
      printk(KERN_ERR "Failed to
request IRQ for GPIO pin %d\n",
BUTTON_PIN);
      gpio_free(BUTTON_PIN);
      return ret;
   }

   printk(KERN_INFO "Button driver
initialized\n");

   return 0;
}
static void __exit button_exit(void)
{
   free_irq(gpio_to_irq(BUTTON_PIN),
NULL);
   gpio_free(BUTTON_PIN);
   printk(KERN_INFO "Button driver
exited\n");
}

module_init(button_init);
module_exit(button_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Example
Linux button driver");
```

Figure 9: A Case of PLPilot-based program synthesis.

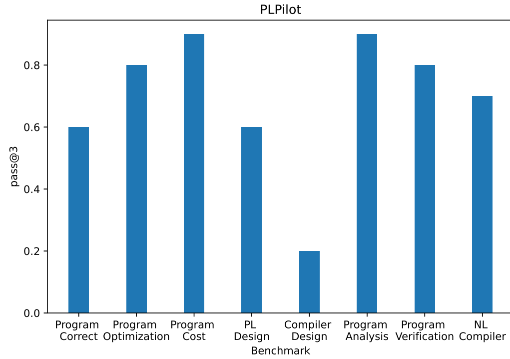

Figure 10: A comprehensive evaluation on PLPilot.

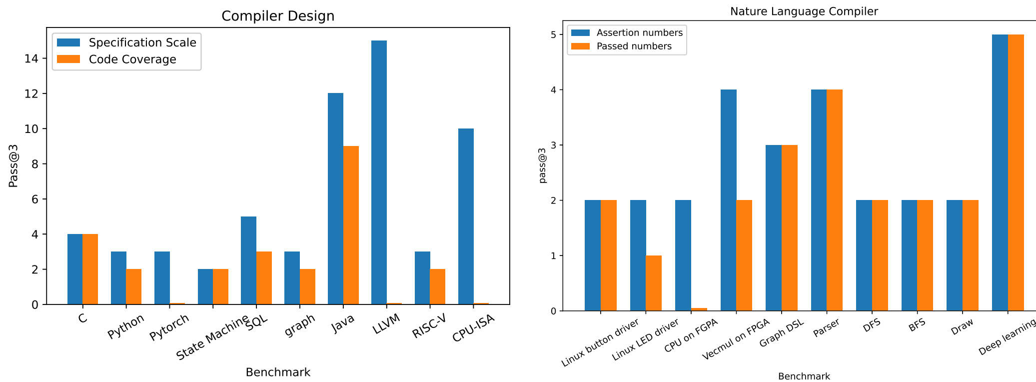

Figure 11: Evaluation on Compiler Design Task (From Specification to Compiler).

Figure 12: Evaluation on natural language program synthesis task (Fron natural language to program).

D Detailed Evaluation Result

Fig.10, 11, 13, 16,14,18,15,17,12 show several evaluation results. Fig. 10 shows that The results show that for most of the tasks in programming language design, PLPilot has a positive evaluation results. Fig. 11 shows the ability of LLM for compiler design.The blue columns represent the number of specifications, while the orange columns represent the number of specifications covered by the compiler code generated by LLM. Fig. 13 shows the ability of LLM for programming language design, EBNF , Sematic and type system are used to evaluate the quality of programming languages. Only the number of errors in the benchmark where the programming languages design failed is listed here.Fig.16 shows the ability of LLM for program analysis.The blue columns represent the number of semantic information in the program , while the orange columns represent the number of semantic information that LLM can analyze correctly. Fig.14 shows the ability of LLM to compare program performance.The results of the LLM output are compared with the ground truth which can evaluate the ability of the large model as a cost model.Fig.18 shows the ability of LLM for program correction. The blue columns represent the total number of errors ,while the orange columns represent the number corrected by the LLM.Fig.15 shows the ability of LLM for program verification. The blue columns represent the total number of validation statements, while the orange columns represent the number of validation statements correctly verified by LLM. Fig.17 shows the ability of LLM for program optimization.The speed-up ratio of LLM-optimized programs reflects the effectiveness of LLM in program optimization. Fig.12 shows the ability of LLM for natural language programming. Blue columns represent the number of sub-tasks mentioned in the prompt, while orange columns represent the number of sub-tasks that can be completed by the code generated by LLM.

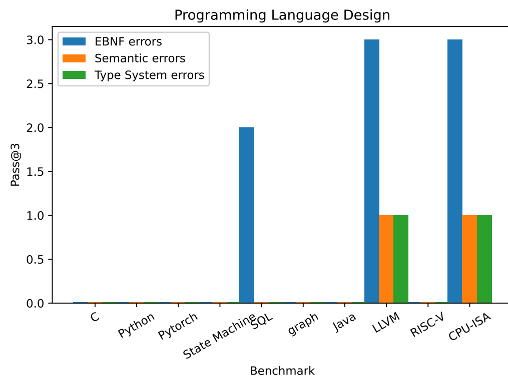

Figure 13: Evaluation on programming language specification generation.

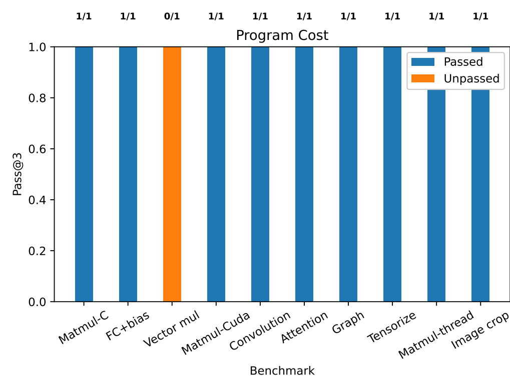

Figure 14: Evaluation on program cost comparison.

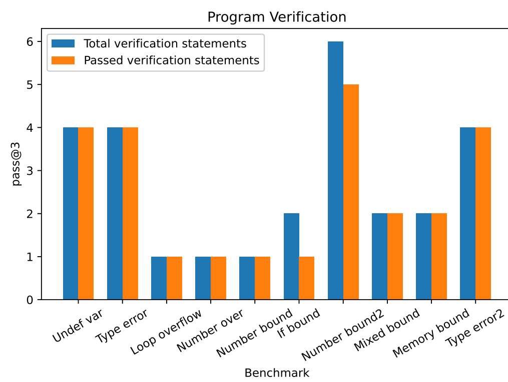

Figure 15: Evaluation on program verification.

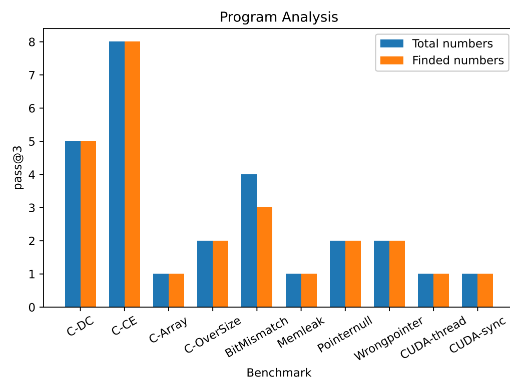

Figure 16: Evaluation on program analysis.

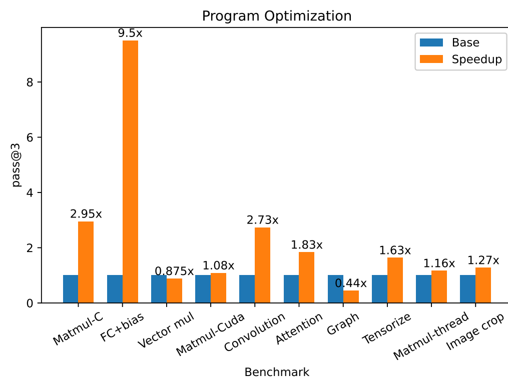

Figure 17: Evaluation on program optimization.

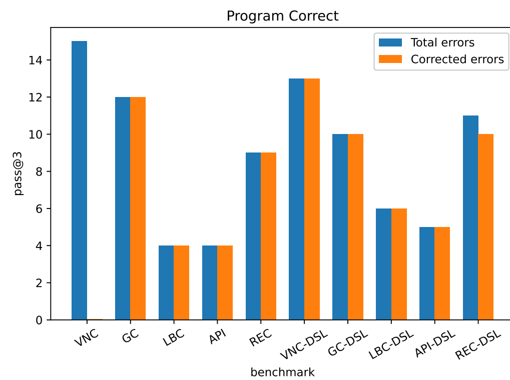

Figure 18: Evaluation on program correction.