Compiler-R1: Towards Agentic Compiler Auto-tuning with Reinforcement Learning

Haolin Pan[1,2,3]*, **Hongyu Lin**[2,3]*, **Haoran Luo**[4,†], **Yang Liu**[2,3]
Kaichun Yao[2], **Libo Zhang**[2], **Mingjie Xing**[2,3,†], **Yanjun Wu**[2,3]
[1]Hangzhou Institute for Advanced Study, UCAS, China
[2]Institute of Software Chinese Academy of Sciences, China
[3]University of Chinese Academy of Sciences, China
[4]Nanyang Technological University, Singapore
{hongyu2021,mingjie}@iscas.ac.cn, haoran.luo@ieee.org
panhaolin21@mails.ucas.ac.cn

Abstract

Compiler auto-tuning optimizes pass sequences to improve performance metrics such as Intermediate Representation (IR) instruction count. Although recent advances leveraging Large Language Models (LLMs) have shown promise in automating compiler tuning, two significant challenges still remain: the absence of high-quality reasoning datasets for agents training, and limited effective interactions with the compilation environment. In this work, we introduce Compiler-R1, the first reinforcement learning (RL)-driven framework specifically augmenting LLM capabilities for compiler auto-tuning. Compiler-R1 features a curated, high-quality reasoning dataset and a novel two-stage end-to-end RL training pipeline, enabling efficient environment exploration and learning through an outcome-based reward. Extensive experiments across seven datasets demonstrate Compiler-R1 achieving an average 8.46% IR instruction count reduction compared to `opt -Oz`, showcasing the strong potential of RL-trained LLMs for compiler optimization. Our code and datasets are publicly available at `https://github.com/Panhaolin2001/Compiler-R1`.

1 Introduction

Compiler auto-tuning [2, 5, 6] focus on automatically select and order compilation passes, modular optimization or analysis steps in modern compilers like LLVM [20], to improve program performance. A typical tuning objective is reducing the Intermediate Representation (IR) instruction count while preserving program correctness, as depicted in Figure 1. However, this task presents inherent challenging due to the combinatorial explosion of possible pass sequences and the intricate interactions between individual passes.

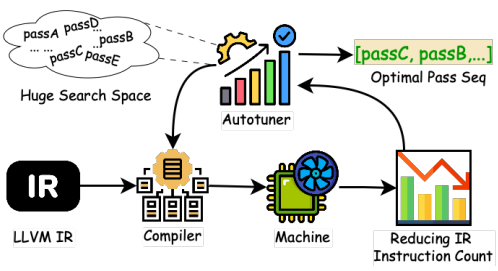

Figure 1: Overview of compiler auto-tuning task.

Recent years have witnessed significant advancements in compiler auto-tuning methods, evolving through three major generations: **Heuristic-based Methods**: Early work in iterative compilation [4, 7, 13, 1] establish fundamental search strategies

*These authors contributed equally to this work.
†Corresponding author(s).

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

such as genetic algorithm and simulated annealing. **Machine Learning (ML)-enhanced Methods**: Subsequent approaches [6, 31, 15, 26] leveraged ML models to improve tuning efficiency on unseen benchmarks. However, these traditional approaches, including both heuristic and ML methods, often suffer from inherent inefficiency and limited generalization. **Large Language Models (LLMs)-based Methods**: With the advent of LLMs, recent efforts [17, 9, 8] have explored their application to compiler auto-tuning tasks, mitigating some of the aforementioned limitations.

Although recent LLMs [11, 18] exhibit remarkable reasoning and problem-solving capabilities, directly applying them to compiler auto-tuning faces two critical challenges. First, there is an **absence of high-quality training datasets tailored for auto-tuning**. Effective interactive learning for LLM agents requires datasets that include Chain-of-Thought (CoT) reasoning, external tool integration, and environment feedback, which are currently lacking in compiler tuning domains. Second, existing methods based on Supervised Fine-Tuning (SFT) exhibit **limited interaction with the compilation environment**, restricting the model's ability to autonomously explore and adaptively learn from feedback, thus resulting in poor generalization to unseen programs.

To address these challenges, we propose **Compiler-R1**, the first reinforcement learning (RL)-based framework specifically designed to enhance LLM capabilities in compiler auto-tuning. Compiler-R1 introduces two main innovations: **(1) A high-quality auto-tuning reasoning dataset:** We curate a comprehensive dataset comprising 19,603 carefully constructed samples, encompassing diverse and representative tuning scenarios, enabling effective interactive learning for compiler optimization tasks. **(2) A two-stage end-to-end RL training framework:** We develop a novel RL framework that equips LLM agents with tool interfaces for autonomous environment exploration and leverages an outcome-based reward function to guide learning. This approach enables the agent to capture both individual pass characteristics and their compositional interactions, significantly enhancing generalization capability and tuning efficiency.

We validate Compiler-R1 using various open-source LLM backbones on seven representative benchmarks. Experimental results indicate that Compiler-R1 consistently achieves an average IR instruction count reduction of 8.46% compared to the baseline method (`opt -Oz`), demonstrating the considerable promise of RL-trained LLMs for advancing compiler auto-tuning research.

Our principal contributions are summarized as follows:

- We construct a comprehensive, high-quality auto-tuning reasoning dataset by rigorously selecting and integrating multiple authoritative compiler datasets, encompassing diverse and representative tuning scenarios.
- We propose **Compiler-R1**, the first RL-based two-stage training framework specifically designed to augment LLMs with tool-invocation and advanced reasoning capabilities tailored to compiler auto-tuning tasks.
- We demonstrate through comprehensive evaluations that **Compiler-R1** consistently outperforms other baseline methods, highlighting the substantial promise of RL-enhanced LLM approaches for advancing compiler optimization.

2 Related Work

2.1 Rule-based Reinforcement Learning with Large Language Models

Reinforcement Learning (RL) enables agents to perform sequential decision-making by interacting with environments to maximize cumulative rewards. Recent advancements in rule-based RL frameworks, such as DeepSeek-R1 [11], have notably enhanced the reasoning and decision-making capabilities of Large Language Models (LLMs). This approach has effectively been extended to various domains, including question answering [25], knowledge retrieval [19, 24], logical reasoning [29], mathematical problem-solving [28], code generation [23], and operating system [22]. Nevertheless, the application of rule-based RL to compiler auto-tuning tasks remains largely unexplored.

2.2 Compiler Auto-tuning

Compiler auto-tuning aims to automatically identify optimal pass sequences to enhance program performance. Traditional compilers like GCC and LLVM utilize fixed optimization pipelines designed

by domain experts, which often do not adapt well to specific programs or hardware. Iterative compilation methods [7, 13] employing heuristics such as genetic algorithms incur substantial overhead. Supervised learning approaches, like coreset-NVP [21], rely heavily on labeled optimal sequences, which are challenging to obtain. RL-based methods, such as AutoPhase [15], dynamically adjust optimization strategies but have limited generalization. Recently, LLM-based approaches [8] have shown potential by directly generating pass sequences from source code; however, they typically lack sufficient environmental interaction, constraining their reasoning and adaptability.

3 Methodology: Compiler-R1

We present Compiler-R1, a novel framework that introduces large language model (LLM)-driven compiler auto-tuning, incorporating both supervised learning and reinforcement learning (RL) techniques. As illustrated in Figure 2, Compiler-R1 represents the first application of reinforcement learning to LLM-based compiler optimization, enabling models to move beyond imitation and autonomously learn optimization strategies through structured interaction with a simulated compilation environment.

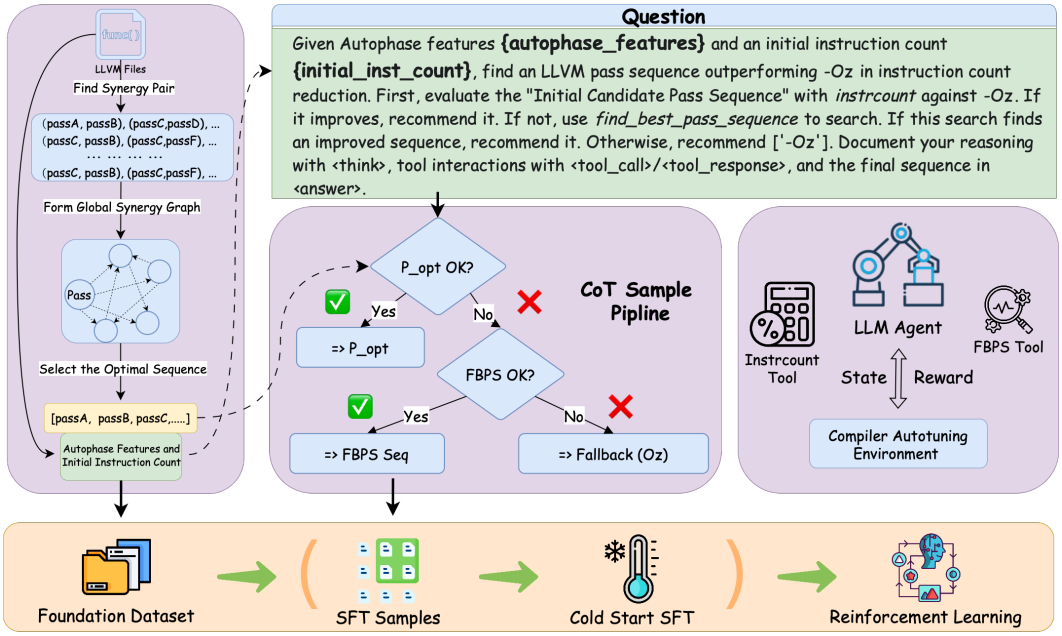

Figure 2: Compiler-R1: A two-stage LLM training framework for compiler auto-tuning. **Stage 1 (SFT):** A foundation dataset derived from synergistic pass analysis is used to create SFT samples. These samples guide the LLM through a CoT process, teaching interaction with `instrcount` and `find_best_pass_sequence` (FBPS) tools. **Stage 2 (RL):** The SFT-initialized LLM agent interacts with the compiler autotuning environment, receiving state feedback and rewards to learn an optimal policy for pass sequence generation.

3.1 Foundation Dataset Construction

To enable effective compiler auto-tuning, we construct a high-quality dataset tailored for LLM training, derived from empirical compiler tuning workflows (e.g., CFSAT [26]) and enhanced with structural insights. Our process involves the following pipelines:

Raw LLVM IR Dataset Preparation. We aggregate multiple public datasets from CompilerGym [10], applying coreset-inspired sampling [21] to ensure representational diversity. Programs with more than 10,000 IR instructions are excluded for computational feasibility. The training set comprises six uncurated datasets, while four curated benchmarks (cbench-v1, chstone-v0, mibench-v1, and npb-v0 [12, 14, 3, 16]) are served for out-of-distribution evaluation.

Synergy Pass Pair Identification. For each training program P, we derived its high-quality pass sequences S_P by collecting synergistic optimization pass pairs $(A, B) \in O \times O$ satisfying the following criteria:

$$S_P = \{(A, B) \mid (\mathrm{Count}(\mathrm{Apply}(P, B)) < \mathrm{Count}(P)) \wedge (\mathrm{Count}(\mathrm{Apply}(P, A, B)) < \mathrm{Count}(\mathrm{Apply}(P, B)))\} \tag{1}$$

where $\mathrm{Apply}(P, B)$ denotes applying pass B to program P, and $\mathrm{Count}(\cdot)$ is the IR instruction count. A pair (A, B) is considered if B alone reduces the instruction count and A further reduces it when applied before B. These synergistic pairs indicate potential beneficial interactions between passes on a per-program basis.

Global Synergy Graph Construction. We aggregate these per-program synergistic pass pairs to construct a Global Synergy Graph, where nodes represent passes and edges denote pairs (A, B) found to be synergistic for at least one program. This global perspective helps identify recurring beneficial pass interactions across the dataset.

Graph-Guided Optimal Sequence Selection. Drawing from the Global Synergy Graph, we generate 100 candidate pass sequences ($S_{cand}(P)$) per program using random walks that prioritize synergistic combinations. Each candidate $s \in S_{cand}(P)$ is evaluated by its relative improvement over the `Oz` baseline, quantified as:

$$\mathrm{OverOz}(s, P) = \frac{\mathrm{Count}(\mathrm{Apply}(P, \mathrm{Oz})) - \mathrm{Count}(\mathrm{Apply}(P, s))}{\mathrm{Count}(\mathrm{Apply}(P, \mathrm{Oz}))} \tag{2}$$

The sequence maximizing OverOz is selected as the optimal pass sequence $\mathrm{PassSeq}_{\text{opt}}(P)$. This empirical, graph-informed method, validated against a strong baseline, provides high-quality optimization examples for training. The derived $\mathrm{PassSeq}_{\text{opt}}(P)$ for each program then serves as the target solution used to construct the **Simulated LLM Thought and Action Trajectory** for the SFT stage of our training pipeline.

Feature Extraction and Representation. To address LLM context limits, we compress programs representations using AutoPhase's 56 statistical features [15], capturing structural properties such as type distributions and control-flow properties. The extracted **AutoPhase features**, along with the program's **initial instruction count**, form the core components of the "question" part of our SFT samples, representing the program to the LLM.

3.2 Training Pipeline

We propose a two-stage training pipeline designed to progressively enhance LLMs' capabilities in auto-tuning task. The first stage employs supervised fine-tuning (SFT) to establish fundamental reasoning and tool invocation skills. The second stage utilizes applies RL to refine responses based on performance feedback.

3.2.1 Stage 1: Supervised Fine-Tuning for Cold Start

The SFT stage addresses the cold-start challenge by teaching the LLM how to reason over program features and interact with external tools to construct optimized pass sequences. raining data is structured to emulate interactive problem-solving, following a "thought–action–feedback" loop that mirrors realistic tool use.

Core External Tools for SFT. Two key tools are integrated into the SFT process:

- `instrcount`: Evaluates a candidate pass sequence by computing its improvement over the standard `Oz` optimization, based on IR instruction count reduction.
- `find_best_pass_sequence`: Invoked when the initial candidate is ineffective ($\mathrm{OverOz} \leq 0$), this tool searches for a better sequence using the guided search algorithm.

SFT Sample Construction. SFT training samples are designed to simulate interactive problem-solving sessions, each comprising a structured input-output pair. The input represents the problem formulation, including the target program's AutoPhase features, a clear task instruction (e.g., "find an IR-reducing pass sequence"), and the initial IR instruction count. The output models the LLM's step-by-step reasoning and decision-making process, structured as a Chain-of-Thought (CoT) trajectory.

Simulated LLM Thought and Action Trajectory: This part serves as the model's "answer," demonstrating a step-by-step reasoning process. It begins with a `<think>` block where the LLM analyzes the optimal pass sequence $\text{PassSeq}_{\text{opt}}(P)$ and plans to verify its effectiveness using the `instrcount` tool. A corresponding `<tool_call>` is issued, followed by a simulated `<tool_response>` containing execution status and the OverOz metric. The LLM then processes this feedback in another `<think>` block to decide whether to accept the candidate or explore alternatives. This entire reasoning flow, structured with tags like `<think>`, `<tool_call>`, `<tool_response>`, and `<answer>`, forms a complete and learnable SFT sample.

Training Prompt for Compiler-R1

Act as a compiler optimization expert finding an optimal pass sequence for LLVM IR, aiming to reduce the total instruction count. The LLVM IR code is represented by statistical features. The initial statistical features are: **{autophase_features}**. Initial instruction count: **{initial_inst_count}**. Note: When calling the 'instrcount' and 'find_best_pass_sequence' tools, use the exact filepath: {filepath}
Your task is to: Provide and evaluate the Initial Candidate Pass Sequence using the `instrcount` tool to determine its instruction count improvement compared to the default `-Oz` optimization. If the initial sequence provides a positive improvement (`improvement_over_oz > 0`), recommend it as the final answer. If the initial sequence does not provide a positive improvement (`improvement_over_oz <= 0`), use the `find_best_pass_sequence` tool to search for a better sequence. If the search finds a sequence with positive improvement (`improvement_percentage > 0`), recommend that sequence. If the search tool fails to find a sequence with positive improvement, recommend the default `['-Oz']` sequence as the safest option.
Present your reasoning step-by-step using <think></think> tags and tool interactions using <tool_call></tool_call> and <tool_response></tool_response> structure, concluding with the final recommended sequence in an <answer></answer> tag.

3.2.2 Stage 2: Reinforcement Learning for Policy Optimization

After the SFT-based initialization, we further optimize the LLM's policy through reinforcement learning (RL). This stage aims to move beyond the imitative behavior learned via SFT, enabling the LLM—now acting as an agent—to autonomously discover more efficient optimization strategies through trial-and-error interactions with a simulated compilation environment. The agent's actions include generating internal reasoning traces, determining when and how to invoke external tools, and producing the final optimization sequence.

Agent-Environment Interaction. The LLM interacts with the environment by emitting token sequences that correspond to high-level actions, such as `<think>` statements, `<tool_call>` invocations (e.g., `instrcount`, `find_best_pass_sequence`), and final answers enclosed in `<answer>` tags. The environment executes these actions—simulating compilation tasks or tool invocations—and returns feedback in the form of updated state information and scalar rewards.

Reward Design. To guide the learning process, we define a composite reward function $R_{\text{final}} = w_f \cdot R_{\text{format}} + w_a \cdot R_{\text{answer}}$, where w_f and w_a control the relative importance of each component.

- *Format Reward*: R_{format} encourages well-structured and logically coherent trajectories. A full reward is given if the interaction trace (including thoughts, tool usage, and answer) adheres to a predefined valid path (e.g., directly correct guess, or fallback via tool search); otherwise, no reward is assigned:

$$R_{format} = \begin{cases} 1.5, & \text{if the format is correct} \\ 0, & \text{if the format is incorrect} \end{cases} \tag{3}$$

- *Answer Reward*: $R_{\text{answer}} = \alpha \cdot \text{OverOrig}$ quantifies the effectiveness of the final sequence $\text{PassSeq}_{\text{answer}}$ in reducing the IR instruction count compared to the original unoptimized program P_{original}, where α is a scaling factor:

$$\text{OverOrig} = \frac{\text{Count}(P_{\text{original}}) - \text{Count}(\text{Apply}(P, \text{PassSeq}_{\text{answer}}))}{\text{Count}(P_{\text{original}})} \tag{4}$$

Unlike OverOz, which measures improvement over a compiler baseline, OverOrig provides a denser and more consistent reward signal, thereby facilitating stable RL training.

Learning Algorithm. We apply on-policy RL algorithms, specifically PPO [27], GRPO, and REINFORCE++ (RPP) [30], to update the LLM's policy. These algorithms are well-suited to LLMs, handling large action spaces and long-horizon credit assignment. Training is performed iteratively using collected trajectories, progressively refining the policy to generate higher-quality optimization sequences while maintaining training stability.

4 Experiments

This section presents the experimental evaluation of Compiler-R1. We begin by describing the common experimental setup, followed by three main experiments: (1) a performance comparison between Compiler-R1 and various baselines; (2) an analysis of factors affecting task success rate, which reflects effective environment interaction and highlights differences in sampling needs between interactive and non-interactive models; and (3) a case study investigating the impact of input feature representations.

4.1 Experimental Setup

Datasets. Experiments are conducted using an aggregated LLVM IR dataset. Training is performed on six CompilerGym datasets filtered to contain programs with fewer than 10k IR instructions. Evaluation is conducted on seven test suites: `blas`, `cbench`, `chstone`, `mibench`, `npb`, `opencv`, and `tensorflow`.

Baselines. We compare Compiler-R1 against classical compiler-tuning methods, including OpenTuner [1], GA [13], TPE [4], RIO [7], CompTuner [31], BOCA [6], and AutoPhase [15] with PPO-LSTM and PPO-noLSTM variants.

Optimization Space and Robustness. All evaluated models operate within a fixed optimization space comprising 124 LLVM 10.0.0 `opt` passes and the `-Oz` preset (125 total actions). Robust evaluation protocols ensure reliability: Compiler-R1 defaults to `opt -Oz` if critical interaction failures occur (e.g., unparsable outputs or missing results). Non-interactive SFT models revert similarly upon invalid predictions. Compiler-R1 evaluates all successful interactions, whereas SFT models select the best valid sequence from N inference attempts.

Evaluation Metrics. We define the following metrics:

- **Optimization Performance (OverOz%):** Average percentage IR instruction reduction relative to `opt -Oz` (Experiment 1).
- **Success Rate (%):** Proportion of test programs (335 total) for which models successfully execute the interaction protocol (e.g., correct tool invocation, output formatting). Applicable only to interactive models (Experiment 2).
- **Input Feature Impact (OverOrig):** Average IR instruction reduction relative to the original, unoptimized program to evaluate the impact of feature representations (Experiment 3).

Models and Training. All experiments were conducted on Intel Xeon Gold 6430 servers (128 cores, 1TB RAM) with NVIDIA H100 GPUs (4×80GB HBM3).

- **Compiler-R1 (Interactive Framework):** We use Qwen2.5-Instruct models (1.5B, 3B, 7B). Training involves 800 supervised fine-tuning (SFT) samples for protocol initialization, followed by reinforcement learning (GRPO, PPO, or RPP) on 19k interactive episodes, updating over 40 steps.
- **SFT-Only Baselines:** Non-interactive Qwen models (1.5B, 3B, 7B), fine-tuned to directly predict optimal pass sequences. Experiment 1 reports their best performance from $N = 40$ inference attempts per program; Experiment 2 examines their sensitivity by varying N from 1 to 50.

Additional Ablation Baselines (Experiment 2):

- **SFT-only (Cold-start):** Qwen models (1.5B, 3B, 7B), trained only with the 800-sample interaction protocol initialization.
- **RL-only (No SFT):** Qwen-1.5B model trained exclusively via GRPO on the full RL dataset without protocol initialization.

4.2 Experiment 1: Optimization Performance Comparison

This experiment compares the average OverOz% performance of Compiler-R1 variants with SFT-only models (which directly predict pass sequences) and traditional auto-tuning baselines. SFT-only models (SFT-Qwen-1.5B/3B/7B) are evaluated using $N = 40$ inference attempts per program, reporting the best-performing result per instance. Results from traditional methods and AutoPhase variants (PPO-LSTM and PPO-no-LSTM) are also included. Table 1 summarizes the findings.

Table 1: Comparison of Optimization Performance (Average OverOz%) and Time Cost. For SFT-Qwen models, OverOz% results are from N=40 inference attempts.

Method	**blas** (%)	**cbench** (%)	**chstone** (%)	**mibench** (%)	**npb** (%)	**opencv** (%)	**tf** (%)	**Average** (%)	**Time (s)** (avg/prog)
Compiler-R1 (Interactive)									
GRPO-1.5B (Ours)	1.26	4.33	5.00	1.98	8.42	4.28	1.85	3.87	20
GRPO-3B (Ours)	3.14	3.77	2.97	3.84	15.42	3.77	2.96	5.12	22
GRPO-7B (Ours)	**5.27**	5.52	**9.08**	**6.67**	22.44	**4.52**	**5.72**	**8.46**	26
PPO-1.5B (Ours)	2.21	4.44	1.90	2.75	6.48	3.45	1.53	3.25	20
RPP-1.5B (Ours)	0.24	2.96	3.84	0.83	4.43	4.32	2.52	2.73	20
SFT-only (Direct Pass Prediction, N=40 attempts for OverOz%*)*									
SFT-Qwen-1.5B (N=40)	0.90	1.18	0.42	1.43	11.24	3.17	3.69	3.15	29
SFT-Qwen-3B (N=40)	0.78	0.62	0.88	0.68	7.00	2.62	1.15	1.96	40
SFT-Qwen-7B (N=40)	2.29	0.40	6.31	4.63	12.89	3.78	2.31	4.66	55
Traditional Autotuners									
Opentuner	1.60	1.99	6.46	3.33	**26.19**	1.76	1.29	6.09	200
GA	-1.91	1.99	6.51	0.90	25.63	1.76	1.29	5.17	561
TPE	-2.24	0.97	7.60	0.20	24.62	1.46	1.23	4.83	812
RIO	-2.02	0.24	4.98	3.47	23.87	0.79	1.23	4.65	200
CompTuner	-3.06	-0.65	4.38	-0.45	22.99	0.44	1.01	3.52	9803
BOCA	-2.36	-0.16	3.18	-0.69	22.87	1.13	1.22	3.60	2760
AutoPhase (PPO-LSTM)	-1.12	**5.60**	4.49	4.41	-4.67	-0.09	0.05	1.24	2.2
AutoPhase (PPO-noLSTM)	-4.77	-79.69	-80.90	-107.33	-76.69	-2.32	-0.76	-50.35	1.8

Compiler-R1 achieves consistent and substantial improvements across all benchmarks. Notably, GRPO-7B attains the highest average improvement of 8.46% OverOz, significantly outperforming the strongest SFT-only baseline (SFT-Qwen-7B at 4.66%) despite using fewer inference attempts and less runtime. Lower-scale variants (GRPO-1.5B and GRPO-3B) also show competitive performance, further demonstrating the efficacy of RL-based interaction.

Compared to traditional autotuners, Compiler-R1 achieves comparable or better optimization quality with markedly lower runtime. GRPO-7B completes each program in 26 seconds, outperforming OpenTuner (200s), GA (561s), and CompTuner (9800s+). This efficiency makes Compiler-R1 viable for practical deployment.

Although AutoPhase variants benefit from extremely fast inference (2s), their optimization quality is notably limited. In contrast, **Compiler-R1 balances both performance and efficiency**: even the lightweight GRPO-1.5B achieves 3.87% in just 20 seconds, surpassing SFT-Qwen-1.5B (3.15% in 29s). These results affirm Compiler-R1's strong potential in real-world compiler optimization pipelines.

4.3 Experiment 2: Task Success Rate and Interaction Efficiency

This experiment evaluates the ability of different models to follow the predefined interaction protocol, quantified by the *task success rate* on 335 test programs. This metric reflects the agent's capacity to correctly execute the `thought–tool–answer` trajectory, serving as a proxy for effective environment interaction. We examine the effects of RL algorithms, model scale, training strategy (SFT-only, RL-only, SFT+RL), and repetition penalty. Additionally, we compare the interaction efficiency of

Compiler-R1 against SFT-only models under varying inference budgets N. Success rate results are summarized in Table 2, while Figure 3 visualizes the sensitivity of SFT-only models to N.

Table 2: Impact of Configuration and Repetition Penalty on Average Task Success Rate (%) on 335 test programs. Model group abbreviations: Comp.-R1 = Compiler-R1 (SFT Cold-start + RL), SFT (Abl.) = SFT-Qwen (Ablation), RL (Abl.) = RL-only GRPO (Ablation).

	Comp.-R1					*SFT (Abl.)*			*RL (Abl.)*
Rep. Pen.	**GRPO 1.5B**	**PPO 1.5B**	**RPP 1.5B**	**GRPO 3B**	**GRPO 7B**	**SFT 1.5B**	**SFT 3B**	**SFT 7B**	**RL 1.5B**
1.05	**83.36**	65.97	71.04	45.56	51.92	26.98	4.93	36.85	22.55
1.10	79.69	60.95	60.45	42.61	**96.71**	41.08	2.72	9.50	8.92

As shown in Table 2, Compiler-R1 models consistently achieve high task success rates, with GRPO-1.5B reaching 83.36% and GRPO-7B attaining 96.71%. These results highlight the effectiveness of RL in enabling agents to reliably execute the thought-tool-answer trajectory mandated by the interaction protocol. This effective interaction—characterized by precise tool usage and sound decision-making—facilitates the robust optimization performance reported in Experiment 1 through a guided, single-trajectory search process.

Interestingly, GRPO-3B achieves a higher OverOz (5.12%) than GRPO-1.5B (3.87%) despite a much lower success rate (45.56%). This discrepancy illustrates that the success rate captures strict protocol compliance, whereas OverOz reflects end-to-end optimization effectiveness. GRPO-3B occasionally circumvents protocol checks (e.g., `instrcount` validation) by directly invoking `find_best_pass_sequence`. While this results in protocol violations and lower success scores, it can still yield high OverOz if effective sequences are discovered. In contrast, GRPO-1.5B more faithfully follows protocol, likely due to its smaller model capacity and greater adherence to SFT-initialized patterns. GRPO-7B strikes a favorable balance, combining protocol compliance and optimization quality through improved capacity and hyperparameter tuning.

SFT-only models, lacking interaction feedback, require extensive sampling to perform well. As shown in Figure 3, SFT-Qwen-7B achieves -14.78% OverOz at $N = 1$ but improves to 4.66% at $N = 40$, demonstrating their reliance on stochastic inference rather than adaptive refinement. Ablation results also confirm that neither SFT-only nor RL-only models are sufficient alone: only the two-stage SFT+RL pipeline consistently achieves high interaction reliability. Additionally, tuning repetition penalty proves crucial for success rate stability and overall interaction behavior.

In summary, Compiler-R1 exhibits strong interactive robustness across configurations and outperforms non-interactive models in both reliability and efficiency, validating the importance of protocol-aware training for compiler optimization agents.

Table 3: Comparative performance (Avg. Max OverOrig) of AutoPhase features versus raw LLVM IR as input representations for direct compiler pass sequence prediction using SFT with Qwen-1.5B models.

Decoding / N	**AutoPhase**	**LLVM IR**
Greedy (N=1)	0.4314	0.4321
Sampled (N=1)	0.3609	0.3921
Sampled (N=10)	0.4620	0.4643
Sampled (N=20)	0.4693	0.4713
Sampled (N=30)	0.4690	0.4690
Sampled (N=40)	**0.4837**	0.4795
Sampled (N=50)	0.4819	**0.4810**

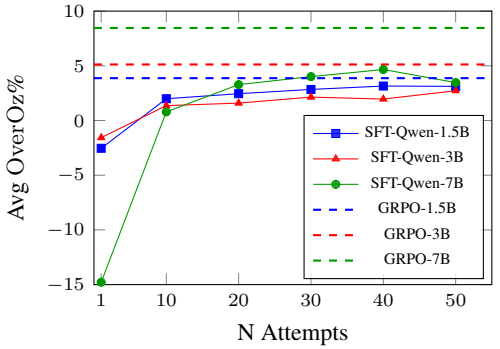

Figure 3: Average OverOz% of SFT-only models vs. N attempts. Dashed: GRPO Ref.

4.4 Experiment 3: Input Representation - AutoPhase vs. Raw LLVM IR

This experiment evaluates the impact of input representation on the effectiveness of compiler pass sequence prediction. Specifically, we compare AutoPhase features against raw LLVM IR as inputs to SFT models based on Qwen-1.5B, using $\mathrm{OverOrig}$ as the evaluation metric.

Two Qwen-1.5B models were trained on over 10,000 LLVM IR functions, with a held-out test set of 100 functions, each containing 1,800–3,000 IR tokens. One model was trained using 56-dimensional AutoPhase features; the other used tokenized raw LLVM IR. We evaluate each model under two decoding strategies:

- **Greedy Decoding ($N = 1$):** Deterministic decoding using argmax at each step.
- **Sampled Decoding ($N = 1$–50):** Nucleus sampling to generate up to 50 diverse candidate sequences per input, reporting the best $\mathrm{OverOrig}$ among them.

Table 3 presents the results, showing that AutoPhase features yield performance closely comparable to raw IR inputs.

Discussion: AutoPhase provides a compact and semantically rich input representation that performs comparably to raw LLVM IR in supervised pass sequence prediction. Under $N = 40$ sampling, the AutoPhase-based model achieves a $\mathrm{OverOrig}$ score of 0.4837, slightly higher than the 0.4795 attained using raw IR. This result suggests that AutoPhase captures sufficient structural and semantic information to guide effective optimization, while offering substantial advantages in terms of input length and interpretability.

One potential explanation is that AutoPhase abstracts away non-essential syntactic variations (e.g., variable names, function ordering), which can obscure semantic similarity in raw IR and lead to inconsistent model behavior. Moreover, performance generally improves with larger N, saturating around $N = 30$–50, which aligns with expectations under stochastic decoding. Notably, greedy decoding ($N = 1$) consistently outperforms single-shot sampling, highlighting the stability of deterministic inference for this task.

In addition, AutoPhase's fixed-size, 56-dimensional feature representation may contribute to generalization across programs of varying sizes. Because program size only influences the numerical values within this compact vector rather than the input dimensionality, the model is less affected by context length limitations that typically constrain raw-code-based methods. By relying on structural and proportional characteristics (e.g., the ratio of branch to memory instructions) rather than absolute code length, the model may learn scale-invariant patterns that transfer across different program complexities. While further empirical validation on larger program benchmarks is necessary, this feature-based representation appears to provide a practical foundation for scalable and potentially more robust learning.

In summary, AutoPhase serves as a strong, compact alternative to raw IR for LLM-based compiler tuning, particularly in scenarios constrained by context window length or requiring interpretable feature inputs.

4.5 Summary of Experimental Findings

Our experiments highlight Compiler-R1's effectiveness in compiler auto-tuning.

First, Compiler-R1—especially the GRPO-7B variant—achieves strong optimization results, with an average $\mathrm{OverOrig}$ gain of 8.46% and a task success rate of 96.71%. These results confirm the benefits of our two-stage interactive RL framework, which enables efficient environment interaction and high-quality pass sequence discovery. In contrast, SFT-only models require extensive sampling (e.g., $N = 40$) to reach comparable performance due to the lack of feedback-driven refinement.

Second, ablation studies demonstrate the necessity of two-stage training. The initial SFT phase is essential for learning interaction protocols that RL alone fails to acquire, while RL fine-tuning further improves efficiency and generalization.

Finally, AutoPhase features offer a compact and effective alternative to raw LLVM IR, delivering similar performance in sequence prediction with reduced input length and better robustness to irrelevant syntactic variations.

Together, these results position Compiler-R1's tool-augmented, two-stage LLM framework as a scalable and efficient solution for modern compiler optimization tasks.

5 Limitations and Future Work

This work examines the use of reinforcement learning to enhance LLM-based compiler auto-tuning. While the results show potential, several limitations remain and suggest directions for future study.

Instruction Count as a Proxy for Performance We use the IR instruction count as the main optimization target. Although this is a common proxy for code size reduction, it does not always correlate with runtime improvements. Cache effects, instruction-level parallelism, and hardware-specific behavior can cause shorter instruction sequences to run slower. Future work could explore integrating direct metrics, such as execution time or cycle counts, into the reward function. This would increase measurement cost but better align the optimization objective with real performance.

Semantic Information Loss in Program Representation We adopt the 56 statistical features from AutoPhase to represent programs. This compact form fits within the LLM's context but abstracts away structural and semantic details, such as control and data flow. This loss may limit the model's ability to capture complex optimization patterns. Future work could explore richer representations that preserve more semantics while keeping manageable size, such as graph encodings or embeddings.

Reliance on External Tools Compiler-R1 relies on external tools to evaluate and search for optimization pass sequences. While this design is practical and aligns with how compilers are typically used, it also means that the model primarily learns to follow a procedural protocol rather than developing a deeper understanding of why a given sequence is effective. Future work could explore curriculum learning or related staged training strategies, where the agent is exposed to progressively more complex optimization tasks. This might help improve its general reasoning ability and reduce reliance on external evaluation tools.

Cross-Version Compiler Compatibility The compiler ecosystem evolves over time, with optimization passes being added, removed, or modified between releases. The current dataset and training setup are tied to a fixed compiler version. When applied to a different version, the framework currently defaults to using the intersection of known passes, which may be conservative. Improving version adaptability remains an open problem. Possible directions include developing methods for automatic adaptation to unseen compiler versions, such as zero-shot or few-shot generalization to new passes, or incremental updates to datasets and reward functions to maintain compatibility with evolving compiler toolchains.

6 Conclusion

We present Compiler-R1, a tool-augmented framework for compiler auto-tuning with LLMs, addressing the lack of domain-specific training data and the limited adaptability of non-interactive methods. Compiler-R1 introduces a high-quality reasoning dataset combining chain-of-thought and tool-use paradigms, and adopts a two-stage training pipeline: SFT for initialization, followed by RL for outcome-driven policy optimization. Experiments show that Compiler-R1, particularly the GRPO-7B variant, achieves an average 8.46% IR instruction reduction and a 96.71% task success rate, outperforming SFT-only baselines and rivaling traditional autotuners with greater efficiency. Ablation studies highlight the importance of combining SFT and RL. Our results demonstrate the potential of LLM-based agents trained with reinforcement learning approaches in complex, interactive compiler optimization tasks.

Compiler-R1 showcases the profound potential of tool-augmented LLM agents trained via RL to revolutionize compiler auto-tuning, validating the effectiveness of an agentic LLM+RL paradigm for complex compiler auto-tuning tasks. Future research directions include leveraging LLMs to improve the interpretability of auto-tuning processes, further developing LLM-RL agents to explicitly understand and exploit synergies between compiler passes, and exploring the efficacy of interactive, turn-by-turn pass application dialogues that could simulate traditional RL auto-tuning paradigms.

7 Acknowledgements

This work is supported by the National Key R&D Program of China, Grant No.2022YFB4401402.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[2] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[5] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on profile and feedback-directed compilation*, 1998.

[6] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.

[7] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[8] 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.

[9] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[10] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.

[11] DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. 2025.

[12] Grigori Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *GCC Developers' Summit*, 2009.

[13] Unai Garciarena and Roberto Santana. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1159–1166, 2016.

[14] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[15] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[16] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195. IEEE, 2008.

[17] Davide Italiano and Chris Cummins. Finding missed code size optimizations in compilers using llms. *arXiv preprint arXiv:2501.00655*, 2024.

[18] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[19] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025.

[20] 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.

[21] Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh James Leather, et al. Learning compiler pass orders using coreset and normalized value prediction. In *International Conference on Machine Learning*, pages 20746–20762. PMLR, 2023.

[22] Hongyu Lin, Yuchen Li, Haoran Luo, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Os-r1: Agentic operating system kernel tuning with reinforcement learning, 2025.

[23] Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. *arXiv preprint arXiv:2503.18470*, 3, 2025.

[24] Haoran Luo, Haihong E, Guanting Chen, Qika Lin, Yikai Guo, Fangzhi Xu, Zemin Kuang, Meina Song, Xiaobao Wu, Yifan Zhu, and Luu Anh Tuan. Graph-r1: Towards agentic graphrag framework via end-to-end reinforcement learning, 2025.

[25] Haoran Luo, Haihong E, Yikai Guo, Qika Lin, Xiaobao Wu, Xinyu Mu, Wenhao Liu, Meina Song, Yifan Zhu, and Luu Anh Tuan. Kbqa-o1: Agentic knowledge base question answering with monte carlo tree search, 2025.

[26] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 614–627, 2025.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models.

[29] Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning, 2025.

[30] Junzi Zhang, Jongho Kim, Brendan O'Donoghue, and Stephen Boyd. Sample efficient reinforcement learning with reinforce. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10887–10895, 2021.

[31] Mingxuan Zhu, Dan Hao, and Junjie Chen. Compiler autotuning through multiple-phase learning. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–38, 2024.

Appendix

A AutoPhase Feature Set

As referenced in **Feature Extraction and Representation**, our framework utilizes the 56 statistical features from AutoPhase [15] to represent programs compactly for the LLM. These features capture various aspects of the program's static structure and instruction mix. Table 4 provides a complete list of these features.

Table 4: List of 56 AutoPhase Features Utilized.

Index	Feature Description	Index	Feature Description
0	BBs: total phi args > 5	28	Number of And insts
1	BBs: total phi args in [1,5]	29	BBs: instruction count in [15,500]
2	BBs: count with 1 predecessor	30	BBs: instruction count < 15
3	BBs: count with 1 predecessor and 1 successor	31	Number of BitCast insts
4	BBs: count with 1 predecessor and 2 successors	32	Number of Br insts
5	BBs: count with 1 successor	33	Number of Call insts
6	BBs: count with 2 predecessors	34	Number of GetElementPtr insts
7	BBs: count with 2 predecessors and 1 successor	35	Number of ICmp insts
8	BBs: count with 2 predecessors and successors	36	Number of LShr insts
9	BBs: count with 2 successors	37	Number of Load insts
10	BBs: count with >2 predecessors	38	Number of Mul insts
11	BBs: Phi node count in range (0,3] per BB	39	Number of Or insts
12	BBs: count with more than 3 Phi nodes	40	Number of PHI insts
13	BBs: count with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Occurrences of 64-bit integer constants	48	Number of Xor insts
21	Occurrences of constant 0	49	Number of ZExt insts
22	Occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (all types)
24	Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

B Synergy Pass Pair Algorithm Details

Algorithm 1 implements the synergy pass pair identification methodology described in Section 3.1. The procedure operates as follows:

C LLVM Optimization Passes

As detailed in Section 4.1, our Compiler-R1 framework and baseline models operate within an action space comprising 124 individual LLVM 10.0.0 `opt` transformation passes, augmented with the `-Oz`. This results in a total of 125 distinct actions available to the optimization agent. Table 5 enumerates these passes and the `-Oz` along with their corresponding indices used within our system. These flags can typically be obtained from the CompilerGym LLVM environment.

Algorithm 1 Finding Synergy Pass Pairs for Program

Require: Program P, set of compiler transformation passes O
Ensure: Set of chained synergy pass pairs $S \leftarrow \emptyset$

Compute V_P, the IR instruction count of the P
for each optimization pass $B \in O$ **do**
    Apply pass B to P and obtain the IR instruction count V_{P_B}
    if $V_{P_B} < V_P$ **then**
        for each optimization pass $A \in O$ **do**
            Apply pass A followed by pass B to P and obtain the IR instruction count $V_{P_{AB}}$ compared to O_0
            if $V_{P_{AB}} < V_{P_B}$ **then**
                $S \leftarrow S \cup \{(A, B)\}$
            end if
        end for
    end if
end for
return S

Table 5: List of Utilized LLVM Compiler Passes and `-Oz` with Corresponding Indices.

Index	Flag	Index	Flag	Index	Flag
0	add-discriminators	42	globalsplit	84	lower-expect
1	adce	43	guard-widening	85	lower-guard-intrinsic
2	aggressive-instcombine	44	hotcoldsplit	86	lowerinvoke
3	alignment-from-assumptions	45	ipconstprop	87	lower-matrix-intrinsics
4	always-inline	46	ipsccp	88	lowerswitch
5	argpromotion	47	indvars	89	lower-widenable-condition
6	attributor	48	irce	90	memcpyopt
7	barrier	49	infer-address-spaces	91	mergefunc
8	bdce	50	inferattrs	92	mergeicmps
9	break-crit-edges	51	inject-tli-mappings	93	mldst-motion
10	simplifycfg	52	instsimplify	94	sancov
11	callsite-splitting	53	instcombine	95	name-anon-globals
12	called-value-propagation	54	instnamer	96	nary-reassociate
13	canonicalize-aliases	55	jump-threading	97	newgvn
14	consthoist	56	lcssa	98	pgo-memop-opt
15	constmerge	57	licm	99	partial-inliner
16	constprop	58	libcalls-shrinkwrap	100	partially-inline-libcalls
17	coro-cleanup	59	load-store-vectorizer	101	post-inline-ee-instrument
18	coro-early	60	loop-data-prefetch	102	functionattrs
19	coro-elide	61	loop-deletion	103	mem2reg
20	coro-split	62	loop-distribute	104	prune-eh
21	correlated-propagation	63	loop-fusion	105	reassociate
22	cross-dso-cfi	64	loop-guard-widening	106	redundant-dbg-inst-elim
23	deadargelim	65	loop-idiom	107	rpo-functionattrs
24	dce	66	loop-instsimplify	108	rewrite-statepoints-for-gc
25	die	67	loop-interchange	109	sccp
26	dse	68	loop-load-elim	110	slp-vectorizer
27	reg2mem	69	loop-predication	111	sroa
28	div-rem-pairs	70	loop-reroll	112	scalarizer
29	early-cse-memssa	71	loop-rotate	113	separate-const-offset-from-gep
30	early-cse	72	loop-simplifycfg	114	simple-loop-unswitch
31	elim-avail-extern	73	loop-simplify	115	sink
32	ee-instrument	74	loop-sink	116	speculative-execution
33	flattencfg	75	loop-reduce	117	slsr
34	float2int	76	loop-unroll-and-jam	118	strip-dead-prototypes
35	forceattrs	77	loop-unroll	119	strip-debug-declare
36	inline	78	loop-unswitch	120	strip-nondebug
37	insert-gcov-profiling	79	loop-vectorize	121	strip
38	gvn-hoist	80	loop-versioning-licm	122	tailcallelim
39	gvn	81	loop-versioning	123	mergereturn
40	globaldce	82	loweratomic	124	`-Oz`
41	globalopt	83	lower-constant-intrinsics		

D Illustrative SFT Answer Trajectories

D.1 Answer 1: Initial Pass Sequence Outperforms Oz

```
<think>
[Initial Pass Sequence Analysis]
- Based on the provided autophase features, I'll try the following pass sequence:
["--dse","--correlated-propagation",...]
- First, I'll use the instrcount tool to check if this sequence provides improvement
    over -Oz optimization.
If improvement_over_oz > 0, I'll use this sequence as my answer.
If improvement_over_oz <= 0, I'll call find_best_pass_sequence tool to find a better seq.
</think>
<tool_call>
{"name":"instrcount","arguments":{"filepath":"example1.ll",
"optimization_flags":["--dse","--correlated-propagation",...]}}
</tool_call>
<tool_response>
{
  "status": "success",
  "improvement_over_oz": 0.06896551724137931
}
</tool_response>
<think>
[Result Analysis]
- The instrcount_tool reports an improvement_over_oz value of 0.06896551724137931.
- Since this value is positive, the pass sequence provides better optimization than -Oz.
- I'll use this pass sequence as my final answer.
</think>
<answer>
["--dse","--correlated-propagation",...]
</answer>
```

D.2 Answer 2: Initial Sequence Underperforms, FBPS Finds Better Sequence

```
<think>
[Initial Pass Sequence Analysis]
- Based on the provided autophase features, I'll try the following pass sequence:
["--jump-threading","--dse","--mem2reg",...]
- First, I'll use the instrcount tool to check if this sequence provides improvement
    over -Oz optimization.
If improvement_over_oz > 0, I'll use this sequence as my answer.
If improvement_over_oz <= 0, I'll call find_best_pass_sequence tool to find a better seq.
</think>
<tool_call>
{"name":"instrcount","arguments":{"filepath":"example2.ll",
"optimization_flags":["--jump-threading","--dse","--mem2reg",...]}}
</tool_call>
<tool_response>
{
  "status": "success",
  "improvement_over_oz": -0.04265402843601896
}
</tool_response>
<think>
[Finding Better Pass Sequence]
- The previous sequence did not provide improvement over -Oz.
- Calling find_best_pass_sequence tool to find a better sequence.
</think>
<tool_call>
{"name":"find_best_pass_sequence","arguments":{"filepath":"example2.ll"}}
</tool_call>
<tool_response>
{
  "status": "success",
  "best_pass_sequence": ["--lower-expect","--reassociate",...],
  "improvement_percentage": 1.9
}
</tool_response>
<think>
[Final Decision - Found Improved Sequence]
- Found a pass sequence with positive improvement: 1.9.
- Using this sequence as the final answer.
</think>
<answer>
["--lower-expect","--reassociate",...]
</answer>
```

D.3 Answer 3: Initial Sequence and FBPS Underperform, Fallback to Oz

```
<think>
[Initial Pass Sequence Analysis]
- Based on the provided autophase features, I'll try the following pass sequence:
["--mergereturn","--simplifycfg",...]
- First, I'll use the instrcount tool to check if this sequence provides improvement
    over -Oz optimization.
If improvement_over_oz > 0, I'll use this sequence as my answer.
If improvement_over_oz <= 0, I'll call find_best_pass_sequence tool to find a better seq.
</think>
<tool_call>
{"name":"instrcount","arguments":{"filepath":"example3.ll",
"optimization_flags":["--mergereturn","--simplifycfg",...]}}
</tool_call>
<tool_response>
{
  "status": "success",
  "improvement_over_oz": -0.02564102564102564
}
</tool_response>
<think>
[Finding Better Pass Sequence]
- The previous sequence did not provide improvement over -Oz.
- Calling find_best_pass_sequence tool to find a better sequence.
</think>
<tool_call>
{"name":"find_best_pass_sequence","arguments":{"filepath":"example3.ll"}}
</tool_call>
<tool_response>
{
  "status": "success",
  "best_pass_sequence": ["--simplifycfg", "--mem2reg",...],
  "improvement_percentage": 0.0
}
</tool_response>
<think>
[Final Decision - Fallback to -Oz]
- After attempting with find_best_pass_sequence tool, no sequence with positive
    improvement was found.
- Falling back to the standard -Oz optimization as the final answer.
</think>
<answer>
['-Oz']
</answer>
```

E Experiment 3 Prompt Templates

E.1 AutoPhase Features Model

```
PROMPT:
Act as a compiler optimization expert finding an optimal pass sequence for LLVM IR,
aiming to reduce the total instruction count.
The LLVM IR code is represented by autophase features,
the initial autophase features are: {Autophase Features}
Initial instruction count: {initial_inst_count}

ANSWER TEMPLATE:
<answer>
[Recommended pass sequence goes here]
</answer>
```

E.2 Raw LLVM IR Model

```
PROMPT:
Act as a compiler optimization expert finding an optimal pass sequence for LLVM IR,
aiming to reduce the total instruction count.
The LLVM IR code is: {ll_code}
Initial instruction count for this code: {initial_inst_count}

ANSWER TEMPLATE:
<answer>
[Recommended pass sequence goes here]
</answer>
```

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the contributions of Compiler-R1, including the construction of a high-quality reasoning dataset and a novel two-stage RL training pipeline. The claims are supported by experimental results in Section 4.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: The paper discusses limitations in Section 5, such as the need for further research on interpretability and the exploration of synergies between compiler passes.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper focuses on empirical methods and does not include theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 4 provides detailed experimental setups, including datasets, baselines, and evaluation metrics. The code and datasets are publicly available (link provided in the abstract).

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The abstract mentions that the code and datasets are publicly available at https://anonymous.4open.science/r/Compiler-R1-C59C.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4.1 describes the datasets, baselines, optimization space, and evaluation metrics in detail.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Table 1 reports average performance metrics (OverOz%) across multiple benchmarks, and Section 4.2 discusses task success rates with statistical significance.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 4.1 mentions the use of the computer resources.

   Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms to the NeurIPS Code of Ethics, as it involves compiler optimization without ethical risks.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 5 discusses potential positive impacts (e.g., advancing compiler optimization) and acknowledges the need for further research on interpretability, which could mitigate unintended negative effects.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: The research does not involve high-risk models or datasets requiring safeguards.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The paper cites datasets like CompilerGym and AutoPhase, and the public release of code/data implies adherence to licensing terms.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The paper introduces the Compiler-R1 dataset and framework, documented in Sections 3 and 4, with public release details in the abstract.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.

- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The research does not involve human subjects or crowdsourcing.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The research does not involve human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: The LLM was used only for writing or editing purposes and does not affect the core methodology or scientific rigor of the research.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.