HintPilot: LLM-based Compiler Hint Synthesis for Code Optimization

Anonymous ACL submission

Abstract

Code optimization remains a core objective in software development, yet modern compilers struggle to navigate the enormous optimization spaces. While recent research has looked into employing large language models (LLMs) to optimize source code directly, these techniques can introduce semantic errors and miss fine-grained compiler-level optimization opportunities. We present HintPilot, which bridges LLM-based reasoning with traditional compiler infrastructures via synthesizing *compiler hints*—annotations that steer compiler behavior. HintPilot employs retrieval-augmented synthesis over compiler documentation and applies profiling-guided iterative refinement to synthesize semantics-preserving and effective hints. Upon PolyBench and HumanEval-CPP benchmarks, HintPilot achieves up to 6.88× geometric mean speedup over `-Ofast` while preserving program correctness.

1 Introduction

Code optimization is fundamental to software performance, directly affecting execution speed, energy efficiency, and resource utilization across domains ranging from embedded systems to large-scale cloud computing (Wang and O'Boyle, 2018a; Garg et al., 2022a). Traditionally, compilers have served as the primary vehicle for optimization, relying on expert-crafted heuristics applied during the translation from high-level source code to machine instructions. As modern software systems grow increasingly complex and heterogeneous, however, selecting effective optimization strategies has become both labor-intensive and insufficiently adaptive, motivating a shift from static designs toward dynamic, data-driven solutions (Scott, 2025).

Recent advances in large language models (LLMs) have shown promise in generating performant code (Gong et al., 2025; Dong et al., 2025a). Nevertheless, ensuring correctness alongside performance remains a fundamental challenge (Waghjale et al., 2024; Yang et al., 2024a). Existing LLM-based optimization approaches often operate by directly modifying source code to improve efficiency (Shypula et al., 2024; Gao et al., 2024; Zhao et al., 2025) (See Fig. 1(a)). While such methods can yield performance gains, they typically rely on invasive transformations that risk violating program semantics. Alternative work explores LLM-guided selection of compiler passes or flags without modifying source code (Cummins et al., 2024) (See Fig. 1(b)). Still, they usually apply a single global optimization configuration to the entire program, missing fine-grained opportunities and overlooking the performance impact of functions, variables, and non-loop statements.

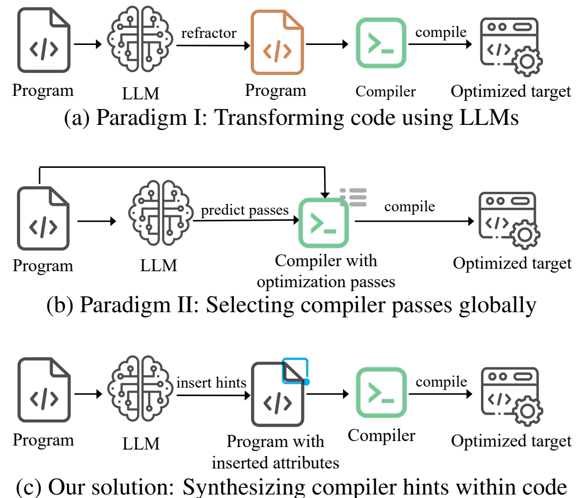


Figure 1: The comparison between different paradigms of LLM-based code optimization

To fill this gap, we introduce a new paradigm for code optimization by synthesizing compiler hints, which reconciles the flexibility of LLM-based code reasoning with the reliability guarantees of traditional compilers (see Fig. 1(c)). Specifically, compiler hints are annotations attached to functions, statements, or classes of a program, with common categories summarized in Table 1. Notably, apply-

Table 1: Examples of compiler hints

Category	Representatives
Storage control	`section`
Optimization control	`used`, `unused`
Memory layout	`packed`
Alignment control	`aligned(n)`
Warning handling	`warn_unused_result`
Visibility control	`visibility("hidden")`
Entry control	`naked`, `interrupt`
Calling convention	`cdecl`, `stdcall`
Constructor/Destructor	`constructor`, `destructor`

```
bool has_close_elems(vector<float> nums, float thd){
   int i,j;
   for (i=0;i<nums.size();i++)
      for (j=i+1;j<nums.size();j++)
            if (abs(nums[i]-nums[j])<thd)
                  return true;
   return false;
}
```

(a) The original code before optimization

```
__attribute__((optimize("unroll-loops"), always_inline))
bool has_close_elems(vector<float> nums, float thd){
   int i,j;
   for (i=0;i<nums.size();i++)
      for (j=i+1;j<num.size();j++)
            if (abs(nums[i]-nums[j])<thd)
                  return true;
   return false;
}
```

(b) The optimized code with the compiler hints

Figure 2: An example of code optimization by synthesizing compiler hints. `optimize("unroll-loops")` unrolls the loops in the function. `always_inline` inlines the function when it is called

ing a small set of such hints to a program can yield an optimized version. For example, the program in Fig. 2(b) with the compiler hint achieves a 1.47× speedup over the one in Fig. 2(a), with correctness preserved by construction. Leveraging such compiler hints, we can achieve code optimization with two key benefits. First, we optimize via declarative hints rather than direct code rewriting. Restricting the model's output to compiler-validated annotations ensures functional correctness. Second, hint-based optimization enables fine-grained, location-specific control. Unlike global compiler flags, hints can be selectively applied to individual program elements, allowing fine-grained, targeted, and context-aware performance tuning.

Despite these advantages, determining where and how to apply compiler hints remains challenging, as their effects depend on complex, nonlocal interactions among program characteristics and compiler behavior. First, synthesized hints should preserve program semantics. Incorrect usage can lead to undefined behavior or wrong results. Second, hint synthesis should be precise to avoid compiler rejections and to produce hints that yield measurable performance improvements.

To address this, we formulate the compiler hint synthesis for the first time as a structured prediction problem, where LLMs generate structured outputs based on external compiler documentation that contains the semantics and usage constraints of available hints. We instantiate this formulation in HINTPILOT, a framework for LLM-based hint synthesis. To ensure semantic preservation, we curate a knowledge base of side-effect-free hints and apply preprocessing to filter out unsafe annotations. Building on this knowledge base, we integrate retrieval-augmented generation with execution-guided feedback to synthesize and iteratively refine contextually appropriate hints, thereby improving performance while preserving correctness.

We evaluate HINTPILOT upon a diverse benchmark suite, including PolyBench (Pouchet and Yuki) and HumanEval_CPP (Zheng et al., 2024), which cover both structured numerical kernels and general-purpose C++ programs. Our experiments show that HINTPILOT consistently outperforms standard compiler optimization baselines under different prompting strategies. Particularly, it achieves geometric mean speedups of up to 3.53× over `-O3` and 6.88× over `-Ofast`. We further compare against the LLM-Compiler-based optimization-pass selection baseline (Cummins et al., 2024) and show that our method achieves higher performance. In addition, ablation studies demonstrate that the observed performance gains are attributable to HINTPILOT 's ability to identify and exploit nonlocal optimization opportunities, such as interactions across functions or program regions, which are difficult for traditional heuristic-driven optimization strategies to capture.

2 Preliminaries

Compiler Hints. Modern compilers, such as GCC, expose hints as lightweight annotations to convey developer intent to the compiler. By attaching hints to functions, loops, or variables, programmers can communicate semantic and performance-related properties that are difficult or costly for the compiler to infer reliably. Typical examples include performance-oriented hints (e.g., `hot`), branch-related cues (e.g., `likely`), and memory-behavior directives (e.g., `prefetch`).

Compiler hints occupy a distinct position in the optimization stack. Unlike source-level code transformations, hints do not alter control flow

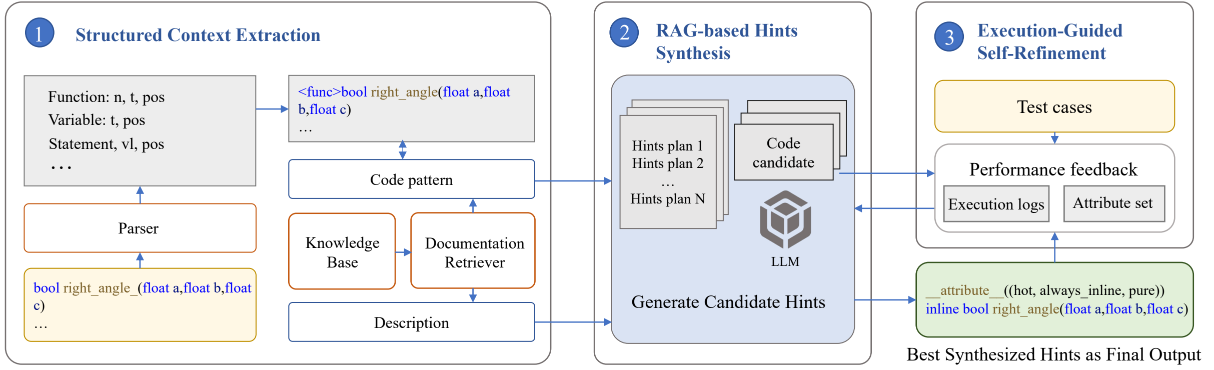


Figure 3: The workflow of HINTPILOT

or data flow; instead, they declaratively constrain or guide the compiler's internal optimization decisions. Compared to global optimization flags, hints can be selectively applied to individual program elements, enabling fine-grained, localized, and context-sensitive tuning. In this sense, hints define a "safe knob surface" for performance tuning.

Compiler Hint Synthesis. Based on the observation of compiler hints, we formulate the problem of compiler hint synthesis for code optimization. Let P be a program to be optimized and $\mathcal{I}$ be a set of inputs of P. For an input $a \in \mathcal{I}$, let $t(P, a)$ denote the execution time of running program P on a under a fixed evaluation setting. Let $\mathcal{L}(P)$ denote the set of valid insertion locations in P, and let $\mathcal{H}$ denote the set of available compiler hints that are intended to preserve program semantics. A *hint assignment* S is modeled as a (partial) mapping $S : \mathcal{L}(P) \to \mathcal{H}$, which assigns compiler hints to a selected subset of program locations. We denote by $P \oplus S$ the program obtained by augmenting P with the compiler hints specified by S at their corresponding locations.

Based on these concepts, we formulate the problem as follows: Given P and $\mathcal{I}$, the goal is to find an assignment S^* that minimizes the overall execution time over the input set $\mathcal{I}$:

$$S^* = \arg\min_S \sum_{a \in \mathcal{I}} t(P \oplus S, a).$$

However, effective compiler hint synthesis requires more than syntactic correctness; it demands semantic understanding of both program behavior and compiler internals. Fortunately, LLMs offer a promising foundation for this task, given their demonstrated strengths in code summarization (Fang et al., 2024), intent inference (Wang et al., 2025; Ruan et al., 2024), and edit generation (Dong et al., 2025b). Nevertheless, direct prompting cannot solve the problem end-to-end. First, inserted hints must preserve program semantics. Incorrect annotations, for example, marking a side-effecting function as `const`, can induce undefined behavior or miscompilation. Second, compiler hints constitute long-tail knowledge in LLM training corpora. As a result, models may hallucinate hints, apply them incorrectly, or generate annotations that are syntactically valid but semantically vacuous. Such outputs may fail to influence optimization decisions, yielding no performance benefit. To resolve the above challenges, we introduce a framework for compiler hint synthesis, named HINTPILOT, detailed in Sec. 3.

3 Our Solution: HINTPILOT

Figure 3 depicts the workflow of HINTPILOT. Following existing studies (Shypula et al., 2024), it takes a target program and a set of test cases as input. Technically, HINTPILOT synthesizes compiler hints for code optimization through three phases. First, it parses the program to identify valid insertion sites via *structured context extraction*. Second, during the stage of *RAG-based hint synthesis*, HINTPILOT retrieves relevant hint descriptions and usage examples from the knowledge base, which are incorporated into a prompt that guides the LLM to generate a sequence of candidate hints. Third, HINTPILOT applies the generated hints, compiles the program, and executes the target code. If compilation or testing fails or performance degrades, HINTPILOT performs *execution-guided self-refinement*, leveraging the diagnostics to guide iterative regeneration.

Notably, HINTPILOT should exclude compiler hints that may potentially alter program semantics. Hence, to ensure program correctness, we preprocess compiler documentation and construct a knowledge base of semantic-preserving compiler

hints. In what follows, we first describe the construction of this knowledge base (Sec. 3.1) and then present the technical details of each stage of the framework (Sec. 3.2∼Sec. 3.4).

3.1 Knowledge Base Construction

Before code optimization, we construct a structured knowledge base that maps compiler hints to their semantic intent and usage patterns. This knowledge base provides the semantic grounding required by later components to generate valid, context-sensitive hint insertions.

Selecting Semantics-Preserving Hints. We extract hint descriptions from the official documentation of compilers, such as GCC, which we use in the evaluation, specifying each hint's intended use, applicability conditions, and semantic implications. From this corpus, we conservatively select a curated subset of 46 hints that do not affect a program's observable behavior. Hints that introduce side effects or alter semantics are excluded. The retained hints serve solely as declarative guidance to the compiler, enabling optimizations without compromising functional correctness.

Notably, our knowledge base is structured around a general hint schema and is documentation-driven. This design enables straightforward adaptation to other compilers—such as Clang, ICC, and MSVC—and across architectures including x86, ARM, and RISC-V.

Extracting Optimization Patterns. We construct an external knowledge base from official compiler documentation that maps abstract optimization patterns to concrete hint usages. For each hint, we extract its full description. When descriptions are overly long, we use Gemini-2.5 to produce concise summaries of the key information. We also carefully identify and extract the applicable use cases and collect official code examples when available. For hints without official usage programs, we prompt Gemini-2.5 to generate examples and manually verify their correctness. At inference time, we retrieve contextually relevant entries and provide them to the model as explicit semantic grounding, thereby reducing the risk of generating invalid or hallucinated hints.

3.2 Structured Context Extraction

To localize the valid program location for compiler hints, HINTPILOT performs structure-aware analysis of the input program to identify valid and promising locations for hint insertion. Rather than treating code as unstructured text, HINTPILOT parses the source into a structured abstraction that exposes functions, variables, and statements along with their locations and types.

```
bool <func>has_close_elements(<var>vector<float>
numbers, <var>float threshold){
    <var>int i,<var>j;
    <stmt>for (i=0;i<numbers.size();i++)
        <stmt>for (j=i+1;j<numbers.size();j++)
            if (abs(numbers[i]-numbers[j])<threshold)
                return true;
    return false;
}
```

Figure 4: Input program format

The abstraction is obtained using the GCC parser, following the representation in (Wu et al.):

$$C = (F_{n,t,pos}, V_{t,pos}, S_{vl,pos}),$$

where $F_{n,t,pos}$ denotes function metadata including function name n, return type t, and definition location pos; $V_{t,pos}$ denotes variable types and declaration locations; and $S_{vl,pos}$ denotes statements, the variables they reference, and their locations.

This abstraction enables HINTPILOT to (i) identify syntactically valid insertion points such as function definitions and loop headers, and (ii) isolate the minimal structural context required for subsequent retrieval and generation. By constraining the search space to structurally valid regions, this component reduces noise and improves the reliability of downstream LLM guidance.

3.3 RAG-based Hint Synthesis

Building on the extracted structural context, the second component uses retrieval-augmented generation (RAG) to guide hint synthesis. HINTPILOT combines code structure analysis with semantic grounding from the knowledge base to produce context-aware optimization hints. We present an example of the retrieved content in Appendix C.

Prompt Construction. The LLM is prompted with the structural abstraction C and code to suggest possible hints. The prompt includes:

Structural Features: Observations on code structure based on C highlighting their potential impact on program behavior with corresponding markers like `<var>`, `<stmt>`, and `<func>` at the positions given by $F_{n,t,pos}$, $V_{t,pos}$, and $S_{vl,pos}$. Figure 4 demonstrates an example with the marker applied to source code, where possible insertion positions are marked for LLM.

Optimization Recommendations. A set of relevant hint information (`RAG_CONTEXT` in Figure 5) retrieved from the knowledge base, paired with con-

Context: Here are possible attributes to use with usage description and examples:
RAG_CONTEXT [1]
…
Task: You are given input with possible insertion positions marked in the source code.
{ "code": {<func>bool right_angle {…<stmt>for(i=0; i<N; i++){…}…}

Figure 5: The prompt template for hint synthesis

Given the input with possible insertion positions marked in the source code: {<code with markers>}

The following attributes/combinations previously caused performance degradation.
Avoid generating them again, you may use similar but **non-equivalent safe alternatives**:
- Bad attribute sets: {bad_attr_sets}
- And the error/compilation message is: {bad_logs}

Figure 6: The prompt sketch for refinement

cise applicability conditions to encourage correct usage in the given structural context.

Retrieval-Augmented Guidance. To enhance LLM's contextual understanding, a RAG structure retrieves examples from a dataset, stored in a vector database. Each entry in the database contains:

- Descriptions: Descriptions of the hints extracted from the official document, including their impact, usage, and conditions.
- Code pairs (P_p, P_n): Correct use examples of the hints and an example without the hints.

This retrieval mechanism addresses a common limitation of pretrained LLMs, which may lack precise compiler-specific knowledge and can misapply hints even when the intended optimization is reasonable. To further improve reliability, HINTPILOT adopts a parsing-and-planning workflow that first determines candidate insertion sites from C and then synthesizes hints conditioned on retrieved evidence, instead of directly generating end-to-end hint-annotated code.

3.4 Execution-Guided Self-Refinement

We incorporate additional test cases generated by LLMs following (Shypula et al., 2024). During the profiling phase, we utilize these cases to gather runtime information and identify potential performance bottlenecks. The model takes the program source code, profiling data, and compilation feedback as input. The iterative refinement process follows a three-step feedback loop:

- **Suggestion:** The model proposes five candidate sets of compiler hints for the target code, resulting in multiple feedback signals.
- **Execution:** The code is compiled with the proposed hints and benchmarked using test cases to obtain performance metrics.
- **Feedback:** The measured results, together with the applied hints, are fed back to the model to guide the next iteration. They are `bad hint sets` and `bad logs` in Figure 6.

If the refined program passes all test cases and achieves a measurable speedup, it is evaluated on the official test suite. This feedback-guided loop enables HINTPILOT to explore the hint space adaptively, correcting invalid insertions and converging toward performance-improving configurations.

4 Evaluation

We evaluate HINTPILOT upon benchmarks to quantify its effectiveness. This section details the experimental setup, evaluation results, and case studies.

4.1 Experimental Setup

Datasets. We select two benchmarks to cover both numerical and general-purpose algorithms. Polybench (Pouchet and Yuki) comprises 34 numerical kernels essential to high-performance computing. It includes linear algebra operations (e.g., Cholesky decomposition), stencil computations (e.g., Jacobi), and dynamic programming, which is widely used in compiler optimization research. HumanEval_CPP is the C++ version of HumanEval-X (Zheng et al., 2024) that extends HumanEval (Chen et al., 2021) to multiple languages and includes 164 tasks. This dataset evaluates optimization across diverse algorithmic patterns, including sorting, searching, string manipulation, and graph algorithms.

Baselines. We first compare HINTPILOT against two widely adopted compiler optimization flags. The first is -O3, the industry-standard optimization level that applies a comprehensive suite of optimizations to maximize performance. The second is -Ofast, a more aggressive optimization mode that includes all -O3 optimizations along with additional transformations that may relax strict language standard compliance. In addition, we compare HINTPILOT with LLM-Compiler (Cummins et al., 2024), a state-of-the-art Meta-developed LLM for compiler optimization built on Code Llama. LLM-Compiler is trained on a range of compiler-centric tasks, such as compiler pass prediction and compiler emulation. We use the LLM-Compiler-13B variant to predict optimization flags for code optimization.

Models. We evaluate a diverse set of models, including both proprietary APIs and open-weight

architectures. The evaluated models comprise Qwen3-Coder-Plus, GPT-5.2, Codestral-22B-v0.1, Qwen2.5-Coder-14B-Instruct, GPT-4o-mini, and Claude-Sonnet-4.5. To ensure a fair comparison with the baseline llm-compiler, we configure HINTPILOT to use the identical backbone model, CodeLlama-13B-Instruct, isolating the impact of our method from the model capacity.

Prompting Strategies. We investigate three prompting strategies for synthesizing compiler hints. In the zero-shot setting, the model is provided with the source code and profiling information, and is instructed to generate compiler hints directly. To enforce the multi-step reasoning, we also adopt a chain-of-thought (CoT) prompting strategy following prior work (Garg et al., 2025). Finally, in the few-shot prompting, we augment the prompt with a small number of in-context demonstrations (Brown et al., 2020) that exemplify compiler hint insertion. Concretely, we include five representative examples from the knowledge base, curated using Gemini-3 as few-shot examples.

Metrics. We use geometric mean speedup to measure code performance, defined as:

$$\text{Speedup}_{\text{geo}} = \sqrt[N]{\prod_{i=1}^{N} \frac{T_{\text{baseline},i}}{T_{\text{method},i}}}.$$

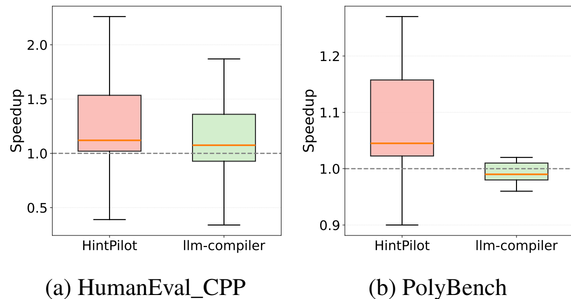
where N means the total number of test cases used for a program.

All experiments are conducted on a 32-core AMD EPYC 7543 server with 512 GB of RAM, running Ubuntu 22.04 and using GCC 13.3.0. Each experiment is repeated ten times to reduce measurement variance, and runtime results are averaged across independent runs.

4.2 Main Results

We evaluate the effectiveness of HINTPILOT by comparing it against two representative baselines. First, we compare with conventional compiler optimization levels, including -O3 and -Ofast. Second, we compare with llm-compiler, a state-of-the-art LLM for compiler optimization.

Overall Effectiveness. Figure 7 compares HINTPILOT with llm-compiler. As shown in the figure, HINTPILOT delivers higher speedups on both datasets, with its distribution consistently shifted upward relative to llm-compiler-13b. And the distribution is largely above 1, relative to the llm-compiler-13b, which may slow down the program. These results suggest that generating compiler hints is a more effective optimization pathway than directly predicting compiler passes for a whole program with LLMs.



(a) HumanEval_CPP (b) PolyBench

Figure 7: Boxplot of geometric mean speedup relative to O3 of HINTPILOT(Codallama-13b-instruct) compared to the llm-compiler-13b baseline.

Comparison with Compiler Optimization Levels. As shown in Table 4.3, HINTPILOT consistently outperforms the standard compiler optimization levels across different backend LLMs. When equipped with the model Qwen3-Coder-Plus, our framework achieves remarkable geometric mean speedups of 3.53× on HumanEval_CPP and 2.10× on Polybench relative to -O3. Even compared to the aggressive -Ofast optimization level, which relaxes strict compliance with the standard for speed, HINTPILOT still delivers substantial gains, achieving 6.88× and 1.63× speedups, respectively. This demonstrates that our method effectively identifies fine-grained optimization opportunities that traditional compiler heuristics miss.

Comparison with llm-compiler-13b. We further present the comparison results against compare HINTPILOT (using the model CodeLlama-13B-Instruct) against llm-compiler-13b (for predicting compiler flags). Figure 7 reports the distribution of speedup rates and the proportion of programs that exhibit actual performance gains. HINTPILOT consistently outperforms llm-compiler-13b, achieving both higher average speedups and greater consistency across benchmarks. These results indicate that our method yields not only larger but also more reliable performance improvements.

4.3 Ablation Study

We conducted an ablation study comparing Zero-shot, Chain-of-Thought (CoT), and CoT + Few-shot strategies, as summarized in Table 3. The results reveal a clear progressive improvement.

Impact of Prompting Strategies. Zero-shot yields only modest gains (1.34× on Humaneval_CPP and 1.12× on PolyBench), likely due to limited compiler-specific knowledge. Adding CoT im-

Table 2: Geometric mean speedup compared with O3 and Ofast. T refers to the maximum number of iterations. Here, T denotes the maximum number of iterations and N denotes the number of candidates. We first report results for different models with $T = 2$ and $N = 5$, and then analyze the impact of varying T and N

Optimization Option	**O3**		**Ofast**	
	HumanEval_CPP	**PolyBench**	**HumanEval_CPP**	**PolyBench**
Qwen3-Coder-Plus	3.53×	2.10×	6.88×	1.63×
GPT-5.2	1.41×	1.51×	1.55×	1.49×
Codestral-22B-v0.1	1.08×	1.21×	1.23×	2.17×
Qwen2.5-Coder-14B-Instruct	2.04×	1.10×	1.84×	1.25×
GPT-4o-mini	1.27×	1.26×	1.86×	1.23×
Claude-Sonnet-4.5	2.88×	1.34×	3.87×	1.35×
Qwen3-Coder-Plus($T = 2$, $N = 1$)	1.20×	1.06×	1.16×	1.20×
Qwen3-Coder-Plus($T = 3$, $N = 1$)	1.18×	1.04×	1.17×	1.06×
Qwen3-Coder-Plus($T = 2$, $N = 3$)	1.30×	1.30×	1.41×	1.16×

proves Humaneval_CPP by $1.62\times$, suggesting that reasoning helps identify optimization-friendly structures, such as dependency-free loops. CoT + Few-shot further boosts Humaneval_CPP to $2.10\times$ but does not help PolyBench ($1.15\times$ vs. $1.17\times$ with CoT), implying that fixed examples can add bias or noise for diverse numerical kernels.

In contrast, HINTPILOT achieves a clear leap to $3.53\times$ on Humaneval_CPP and $2.10\times$ on PolyBench. Moreover, CoT alone reduces the PolyBench compilation rate to 73.52%, while HINTPILOT recovers it to 80.00% and also attains the highest speedup rate with few regressions. Overall, these results support that our solution retrieves precise, context-aware patterns beyond a static few-shot prompting strategy.

Impact of Selected Models. As shown in Table , we also observe that the choice of backbone LLM significantly affects optimization quality. The Qwen3-Coder-Plus and Claude-Sonnet-4.5 models generally outperform smaller models such as Codestral-22B, validating that stronger reasoning capabilities in the base model translate into more effective compiler hints.

Selection of Iteration and Candidate Numbers. The ablation results in Table 4.3 underscore the effectiveness of our execution-guided self-refinement. Increasing the candidate pool size N from 1 to 3 (with $T = 2$) yields a clear performance gain, suggesting that broader exploration of hint combinations for richer execution feedback is key to discovering stronger configurations. However, increasing the number of refinement iterations leads to a slight performance drop, likely due to noise accumulation in the feedback signals.

4.4 Failure Case Analysis

Though the hints in our knowledge base are designed to be semantics-preserving when applied correctly, LLMs may still apply them incorrectly, such as inserting a hint at an inappropriate location or using an invalid syntax, which can lead to compilation errors. We therefore manually inspected cases where HINTPILOT failed to improve performance or caused compilation failures, and summarized the primary failure modes as follows:

- **Syntax Hallucinations**: Despite retrieval augmentation, the model occasionally generates directives with invalid formats or hints unsupported by the specific compiler version.
- **Contextual Mismatch**: Instances where hints are applied to incompatible scopes. For example, loop pragmas on non-loop statements or require compiler flags that were not active.
- **Profiling Instability**: In rare cases, the feedback loop overfits to measurement noise, selecting candidates that offer negligible or unstable gains, as shown in Appendix C.3.

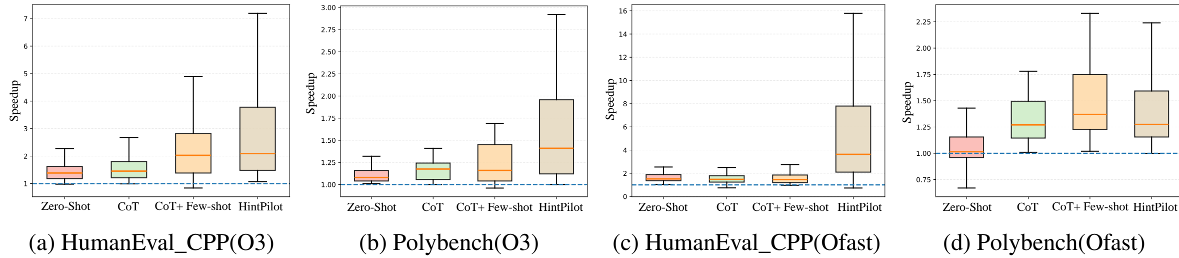
These observations highlight the need for future improvements in static hint verification and more robust profiling protocols to mitigate system noise.

5 Related Work

LLM-based Code Optimization. Recent work on LLM-based code optimization falls into two main categories. The first category comprises generative code refactoring approaches (Zhao et al., 2025; Acharya et al., 2025; Gao et al., 2024; Wang et al., 2022; Garg et al., 2022b, 2025; Gee et al.), which utilize LLMs to rewrite program structures. While benchmarks like PIE (Shypula et al., 2024), ECCO (Waghjale et al., 2024), Mercury (Du et al., 2024), EffiBench (Huang et al.), and Hu-

Table 3: Analysis of compilation rate and efficiency on PolyBench and HumanEval_CPP benchmarks. We use Qwen3-Coder-Plus. We report compilation rate (Comp.), speedup rate (Spd.), and geometric mean speedup relative to -O3. Best results among LLM-based methods are highlighted in bold.

Method	HumanEval_CPP			PolyBench		
	Comp. (%)	Spd. (%)	Speedup	Comp. (%)	Spd. (%)	Speedup
Zero-shot	94.51%	87.19%	1.34×	78.04%	78.04%	1.12×
CoT	97.56%	96.34%	1.62×	73.52%	61.76%	1.17×
CoT + Few-shot	97.56%	96.95%	2.10×	70.58%	61.76%	1.15×
HINTPILOT	97.56%	97.56%	**3.53**×	80.00%	80.00%	**2.10**×



(a) HumanEval_CPP(O3) (b) Polybench(O3) (c) HumanEval_CPP(Ofast) (d) Polybench(Ofast)

Figure 8: Speedup boxplot of different methods across datasets and baselines

manEval (Chen et al., 2021) have demonstrated the potential of these models, they are primarily in Python, and such invasive changes often risk semantic drift. More recently, learning-based alignment strategies, such as EffiCoder (Huang et al., 2025), PerfCodeGen (Peng et al., 2024), and ACE-Code (Yang et al., 2024b), employ fine-tuning or reinforcement learning to align models with efficiency metrics, but incur high training costs.

The second category consists of compiler-centric techniques (Cummins et al., 2024; Merouani et al., 2025; Lamouri et al., 2025; Baghdadi et al.), which integrate LLMs with internal compiler representations or cost models to guide transformations such as pass selection. While these methods offer a principled interface to the compiler, they typically lack fine-grained control at the source level. In contrast, we introduce a lightweight, holistic paradigm spanning multiple granularities. Unlike heavy-weight, training-based, or invasive rewriting methods, we combine RAG with execution feedback to achieve significant performance gains while maintaining semantic correctness.

Machine Learning for Compilers. Compiler optimization involves navigating a high-dimensional transformation space to improve program performance. Traditionally, this process has relied on hand-crafted heuristics of compiler engineers. Recent work has explored machine learning (ML) to automate heuristic design. ML-based approaches have been applied to a range of compiler tasks, including vectorization (Mendis et al., 2019), loop transformations such as unrolling and distribution (Stephenson and Amarasinghe, 2005; Jain et al., 2022), function inlining (Trofin et al., 2021), and register allocation (Das et al., 2020). These methods typically train predictive models offline to replace fixed heuristics or to guide search-based optimization. Reinforcement learning has also been used to dynamically explore optimization sequences. Surveys such as (Allamanis et al., 2018; Wang and O'Boyle, 2018b) provide a broad overview of these techniques. Prior work has primarily focused on selecting global compiler flags or tuning specific phases, such as register allocation. In contrast, our approach enables fine-grained synthesis of optimization hints, tailored to individual program components and capable of influencing multiple compiler phases.

6 Conclusion

Software systems are increasingly complex and performance-critical, yet achieving effective optimization remains challenging. This work identifies an exciting, principled role for LLMs in addressing this challenge by synthesizing compiler hints—an interpretable and constrained interface between developers and compilers. We introduce HINTPILOT, a system that leverages LLMs to generate such hints, enabling compilers to uncover and exploit optimization opportunities without compromising correctness. Our results suggest that LLM-guided hint synthesis is a promising direction for improving code performance and making advanced compiler optimizations more accessible.

7 Limitations

While HINTPILOT demonstrates strong empirical performance across diverse benchmarks, several limitations remain.

Scope of Optimization. HINTPILOT focuses on source-code–level compiler hints that can be attached to localized program elements, such as functions, variables, and statements. As a result, it primarily targets fine-grained, local optimization opportunities exposed through compiler hints. More global optimization decisions, such as whole-program memory layout, interprocedural register allocation, or cross-module code placement, are outside the scope of the current framework because they are not directly controllable via localized hints. Extending HINTPILOT to reason about such global optimizations would require richer interfaces to the compiler and new forms of feedback beyond per-input runtime profiling.

Reliance on Underlying LLMs. The effectiveness of HINTPILOT depends on the reasoning and generalization capabilities of the underlying language model. Stronger models consistently produce higher-quality hint plans, particularly for non-local or uncommon optimization patterns, while smaller models are more prone to invalid or ineffective suggestions. Although retrieval-augmented grounding mitigates hallucinations and syntactic errors, it cannot fully compensate for limited model capacity. As LLMs continue to evolve, we expect HINTPILOT to benefit directly from improvements in model reasoning, code understanding, and long-context handling.

Benchmark Coverage and Generalization. Our evaluation spans numerical kernels, algorithmic programming tasks, and competitive programming benchmarks, providing a broad view of HINTPILOT 's effectiveness. Nevertheless, these datasets do not fully capture all real-world optimization scenarios, such as large-scale industrial codebases, highly concurrent systems, or performance-critical I/O-intensive applications. Moreover, the input sets used for profiling and evaluation may not reflect the full diversity of production workloads. Expanding evaluation to additional datasets, architectures, and workload distributions is necessary to further assess robustness and generalization.

References

Manish Acharya, Yifan Zhang, Kevin Leach, and Yu Huang. 2025. Optimizing code runtime performance through context-aware retrieval-augmented generation. (arXiv:2501.16692). ArXiv:2501.16692 [cs].

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81.

Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. A deep learning based cost model for automatic code optimization.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165.

Harrison Chase. 2022. Langchain. If you use this software, please cite it as below.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. (arXiv:2107.03374). ArXiv:2107.03374 [cs].

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

D Das, S A Ahmad, and V Kumar. 2020. Deep learning-based approximate graph-coloring algorithm for register allocation. In *Workshop on the LLVM Compiler Infrastructure in HPC*, pages 23–32.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025a. A survey on code generation with llm-based agents. (arXiv:2508.00083). ArXiv:2508.00083 [cs].

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025b. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. (arXiv:2402.07844). ArXiv:2402.07844 [cs].

Chunrong Fang, Weisong Sun, Yuchen Chen, Xiao Chen, Zhao Wei, Quanjun Zhang, Yudu You, Bin Luo, Yang Liu, and Zhenyu Chen. 2024. Esale: Enhancing code-summary alignment learning for source code summarization. *IEEE Transactions on Software Engineering*, 50(8):2077–2095.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. (arXiv:2408.12159). ArXiv:2408.12159 [cs].

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022a. Deepperf: A deep learning-based approach for improving software performance. (arXiv:2206.13619). ArXiv:2206.13619 [cs].

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022b. Deepperf: A deep learning-based approach for improving software performance. (arXiv:2206.13619). ArXiv:2206.13619 [cs].

Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2025. Rapgen: An approach for fixing code inefficiencies in zero-shot. (arXiv:2306.17077). ArXiv:2306.17077 [cs].

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. Code-optimise: Self-generated preference data for correctness and efficiency.

Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. 2025. Language models for code optimization: Survey, challenges and future directions. (arXiv:2501.01277). ArXiv:2501.01277 [cs].

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code.

Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. 2025. Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. (arXiv:2410.10209). ArXiv:2410.10209 [cs].

Shalini Jain, S. VenkataKeerthy, Rohit Aggarwal, Tharun Kumar Dangeti, Dibyendu Das, and Ramakrishna Upadrasta. 2022. Reinforcement learning assisted loop distribution for locality and vectorization. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–12.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Djamel Rassem Lamouri, Iheb Nassim Aouadj, Smail Kourta, and Riyadh Baghdadi. 2025. Pearl: Automatic code optimization using deep reinforcement learning. (arXiv:2506.01880). ArXiv:2506.01880 [cs].

C Mendis, C Yang, Y Pu, S Amarasinghe, and M Carbin. 2019. Compiler auto-vectorization with imitation learning. In *NeurIPS'19*, volume 32.

Massinissa Merouani, Islem Kara Bernou, and Riyadh Baghdadi. 2025. Agentic auto-scheduling: An experimental study of llm-guided loop optimization. (arXiv:2511.00592). ArXiv:2511.00592 [cs].

Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Perfcodegen: Improving performance of llm generated code with execution feedback. (arXiv:2412.03578). ArXiv:2412.03578 [cs].

Louis-Noël Pouchet and Tomofumi Yuki. polybench download. SourceForge. Last updated: 2025-05-13. Accessed: 2026-01-04.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*.

Christopher Scott. 2025. Ai-driven code optimization: Improving program efficiency through reinforcement learning approaches.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning performance-improving code edits. (arXiv:2302.07867). ArXiv:2302.07867 [cs].

M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *CGO'05*, pages 123–134.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a machine learning guided compiler optimizations framework. *CoRR*, abs/2101.04808.

Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. 2024. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? (arXiv:2407.14044). ArXiv:2407.14044 [cs].

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2025. Boosting static resource leak detection via llm-based resource-oriented intention inference. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 668–668. IEEE Computer Society.

Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng

Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 129–143, New York, NY, USA. Association for Computing Machinery.

Zheng Wang and Michael O'Boyle. 2018a. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901.

Zheng Wang and Michael O'Boyle. 2018b. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901.

Jiangchang Wu, Yibiao Yang, Maolin Sun, and Yuming Zhou. Unveiling compiler faults via attribute-guided compilation space exploration.

Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. 2024a. Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models. (arXiv:2412.17264). ArXiv:2412.17264 [cs].

Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. 2024b. Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models. *Preprint*, arXiv:2412.17264.

Yuwei Zhao, Yuan-An Xiao, Qianyu Xiao, Zhao Zhang, and Yingfei Xiong. 2025. Semopt: Llm-driven code optimization via rule-based analysis. (arXiv:2510.16384). ArXiv:2510.16384 [cs].

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *Preprint*, arXiv:2303.17568.

A Algorithm of HintPilot

The algorithm of our approach is illustrated in Algorithm 1, which is an iterative, retrieval-augmented code optimization framework driven by large language models (LLMs). Given an input program P_{src}, a domain-specific knowledge base $\mathcal{K}$, and a test suite $\mathcal{T}_{LLM}$, HintPilot searches for a semantically equivalent but higher-performing variant by automatically inserting optimization attributes (or hints) into the source code.

Initialization. The algorithm begins by profiling the original program P_{src} on $\mathcal{T}_{LLM}$ to establish a baseline performance metric $Mbest$. The best-known program P_{best} is initialized to P_{src}. In addition, HintPilot constructs a structural representation S_{struct} of the source code using a compiler-based parser, extracting salient program elements such as functions, loops, and variables. An initially empty feedback history $H_{feedback}$ is maintained to accumulate information from prior optimization attempts.

Algorithm 1 HintPilot

Require: Source Code P_{src}, Knowledge Base $\mathcal{K}$, Test Suite $\mathcal{T}_{LLM}$, Max Iterations T, Candidate Size N
Ensure: Optimized Code P_{best}
  Initialize: $P_{best} \leftarrow P_{src}$, $M_{best} \leftarrow$ Profile$(P_{src}, \mathcal{T}_{LLM})$
  $H_{feedback} \leftarrow \emptyset$ ▷ Initialize interaction history
  $S_{struct} \leftarrow$ GCCParser(P_{src}) ▷ Extract functions, loops, variables
  for $t = 1$ to T **do**
    `// Phase 1: Retrieval-Augmented Prompting`
    $D_{rag} \leftarrow$ Retrieve$(S_{struct}, \mathcal{K})$ ▷ Fetch relevant attribute docs & examples
    $Prompt_t \leftarrow$ ConstructPrompt$(P_{src}, S_{struct}, D_{rag}, H_{feedback})$
    `// Phase 2: Batch Generation of Plans`
    $\mathcal{S}_{plans} \leftarrow \text{LLM}_\pi(Prompt_t, \text{samples} = N)$ ▷ Generate N independent insertion plans
    $\mathcal{R}_{batch} \leftarrow \emptyset$
    for each plan $s_k \in \mathcal{S}_{plans}$ **do**
      $P'_k \leftarrow$ InsertHints(P_{src}, s_k) ▷ Deterministically apply attributes
      $status_k, metric_k \leftarrow$ Profile$(P'_k, \mathcal{T}_{LLM})$ ▷ Compile and benchmark
      $\mathcal{R}_{batch}$.add$(\{s_k, status_k, metric_k\})$
      if $status_k ==$ PASS $\wedge$ $metric_k > M_{best}$ **then**
        $P_{best} \leftarrow P'_k$
        $M_{best} \leftarrow metric_k$
      end if
    end for
    `// Phase 3: Feedback Aggregation`
    $H_{feedback} \leftarrow$ UpdateFeedback$(\mathcal{R}_{batch})$ ▷ Summarize errors and perf gains
  end for
  return P_{best}

Retrieval-Augmented Prompting. In each iteration, HintPilot first performs retrieval-augmented prompting. Using the structural summary S_{struct} as a query, the retriever selects a set of relevant documents D_{rag} from the knowledge base $\mathcal{K}$, including attribute specifications and usage examples. These retrieved artifacts, together with the source code and aggregated feedback from previous iterations, are used to construct a prompt $Prompt_t$ that contextualizes the optimization task for the LLM.

Batch Plan Generation and Evaluation. Given $Prompt_t$, the LLM generates a batch of N candidate insertion plans, each specifying a structured set of attribute insertions. Rather than directly emitting modified code, HintPilot deterministically applies each plan to the original program via InsertHints, yielding a candidate program $P'k$. Each candidate is then compiled and executed against $\mathcal{T}_{LLM}$ to assess both correctness and perfor-

Attribute name: __attribute__((pure))
Description: The function has no observable side effects except through its return value. It may read non-volatile objects, and may modify data only if this does not affect the return value or program-visible state.
Optimization impact: Enables optimizations such as common-subexpression elimination.

Figure 9: **Case I.** HINTPILOT retrieves the usage pattern for the `pure` attribute and inserts it, enabling the compiler to optimize the call.

898 mance. Candidates that pass all tests and improve
899 upon the current best metric are used to update
900 P_{best} and M_{best}.
901 **Feedback Aggregation.** After evaluating all can-
902 didates in the batch, HINTPILOT summarizes
903 the observed outcomes—including compilation
904 failures, runtime errors, and performance im-
905 provements—into an updated feedback history
906 $H_{feedback}$. This feedback is fed into subsequent
907 iterations, enabling the LLM to avoid previously
908 unsuccessful patterns and refine future insertion
909 plans.
910 **Termination.** The algorithm repeats this three-
911 phase process for a fixed number of iterations T
912 and finally returns the best-performing program
913 variant P_{best} discovered during the search.

914 B Implementation Details

915 We implement HINTPILOT using
916 LangChain (Chase, 2022) and vLLM (Kwon
917 et al., 2023). To reduce hallucinations and
918 guarantee syntactic correctness, we adopt a
919 two-stage generation pipeline. Instead of emitting
920 annotated code directly, the model first generates
921 a structured insertion plan, which we then apply
922 deterministically to the source code.

923 To encourage diversity among candidates, we
924 use sampling with temperature $= 1.0$ and top-
925 $p = 1.0$. To ensure these plans are reliably
926 parsable, we enforce a strict JSON schema using
927 the structured output mechanisms provided by the
928 underlying frameworks. For the RAG component,
929 we configure the retriever to return the top $k = 4$
930 relevant documents as context.

931 C Case Study

932 As discussed in Section 3, HINTPILOT significantly
933 mitigates hallucinations in compiler hint generation
934 by grounding decisions in retrieved documentation.
935 We illustrate this process with several representa-
936 tive examples.

```
#include <stdio.h>
#include <math.h>
#include <vector>
#include <string>
using namespace std;
#include <algorithm>
#include <stdlib.h>
__attribute__((pure))
int skjkasdkd(vector<int> lst){
    int largest=0;
    for (int i=0;i<lst.size();i++)
    if (lst[i]>largest)
    {
        bool prime=true;
        for (int j=2;j*j<=lst[i];j++)
        if (lst[i]%j==0) prime=false;
        if (prime) largest=lst[i];
    }
    int sum=0;
    string s;
    s=to_string(largest);
    for (int i=0;i<s.length();i++)
        sum+=s[i]-48;
    return sum;
}
```

Figure 10: **Case I.** The original code is dominated by a deeply-nested loop nest in the dynamic programming kernel.

C.1 Case I 937

Consider the source program in Figure 10, which 938
contains a computationally intensive function 939
called within a loop. HINTPILOT first parses the 940
code structure and queries the knowledge base. Fig- 941
ure 9 displays one retrieved result: a canonical us- 942
age example of the `pure` attribute, explicitly stating 943
that it applies to functions with no side effects. 944

Grounded by this context, the LLM correctly in- 945
fers that the target function depends solely on its 946
arguments and modifies no global state. It then gen- 947
erates a plan to insert `__attribute__((pure))`. 948
This annotation explicitly informs the compiler 949
that the function is side-effect-free, enabling ag- 950
gressive optimizations such as loop-invariant code 951
motion (hoisting the function call out of the loop) 952
and common subexpression elimination, thereby 953
significantly reducing runtime overhead. Remark- 954
ably, this single attribute injection results in a 325× 955
speedup, primarily by eliminating redundant com- 956
putations inside the hot loop. 957

C.2 Case II 958

Consider the source program in Figure 11, whose 959
computation is dominated by two consecutive ma- 960
trix multiplications in the `2mm` kernel. HINTPI- 961
LOT first parses the program and then queries the 962
knowledge base for relevant compiler attributes. 963
Figure 12 displays retrieved knowledge including 964

```
__attribute__((optimize("O3"), hot)) static void init_array(...)) {
    for (i = 0; i < ni; i++)
        for (j = 0; j < nk; j++)
            A[i][j] = (DATA_TYPE) ((i*j+1) % ni) / ni;
    /*Other initialization loops omitted*/
}
__attribute__((optimize("O3"))) static void print_array(...) {
    for (i = 0; i < ni; i++)
        for (j = 0; j < nl; j++) {
            if ((i * ni + j) % 20 == 0) fprintf (POLYBENCH_DUMP_TARGET, "\n");
        fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, D[i][j]);
        }
}
__attribute__((optimize("O3"), hot, pure)) static void kernel_2mm(...) {
    int i, j, k;
    #pragma scop
    for (i = 0; i < _PB_NI; i++)
        for (j = 0; j < _PB_NJ; j++) {
            tmp[i][j] = SCALAR_VAL(0.0);
            for (k = 0; k < _PB_NK; ++k)
                tmp[i][j] += alpha * A[i][k] * B[k][j];
        }
    /*Second matrix multiplication omitted*/
    #pragma endscop
}
__attribute__((optimize("O3"), hot)) int main(int argc, char** argv) {
    /*Calls omitted*/
}
```

Figure 11: **Case II.** The original code contains a redundant function call inside a loop.

Attribute name: __attribute__((optimize))
Description: Specifies that a function is compiled with optimization options different from those provided on the command line. The specified options are treated as if appended to the compiler's command-line flags for that function.
Optimization impact: Enables per-function control of optimization settings, overriding the global compilation level.

Attribute name: __attribute__((hot))
Description: Indicates that the execution path following the annotated label is more likely than alternative paths. The compiler assumes that this labeled path is frequently executed when making optimization decisions.
Optimization impact: Encourages allocating more optimization effort along the hot path.

Figure 12: **Case II.** HINTPILOT retrieves compiler optimization attributes for compute-intensive kernels and applies them to prioritize optimization of the main execution path.

`optimize("O3")` and `hot`, which convey optimization priority in compute-intensive regions.

Grounded by this context, the LLM synthesizes a plan that assigns `__attribute__((optimize("O3"), hot))` to the kernel as well as other routines on the main execution path. These annotations inform the compiler that the functions lie on the performance-critical execution path, enabling aggressive optimizations and prioritizing code generation. The kernel is additionally marked as `pure`, which reduces side-effect-related constraints and allows more effective instruction scheduling and register allocation. The attributed version achieves an 89× speedup by reducing computation and memory-access overhead in the core kernel.

C.3 Case III

Consider the program in Figure 13, which consists of a compute-intensive initialization routine followed by the factorization kernel. HINTPILOT profiles the execution to identify performance-critical regions, determines that the LU kernel is the main bottleneck, and queries the knowledge base. Figure 14 displays the retrieved `cold` attribute, indicating expected execution frequency and allowing the compiler to adjust optimization effort accordingly.

Grounded in this context, the LLM marks `kernel_lu` as `__attribute__((hot))` and assigns `__attribute__((cold))` to the initialization routine. However, in this case, the initialization phase performs substantial computation and accounts for a significant portion of the total runtime, even though it is executed only once. The `cold` attribute, therefore, biases the compiler toward more conservative code generation in a compute-intensive region, leading to increased execution time, while the `hot` attribute on the LU kernel provides little benefit due to inherent loop dependences.

Moreover, this case reveals instability in the feedback loop. Since performance is evaluated only at

```c
__attribute__((cold))
static void init_array (int n, DATA_TYPE POLYBENCH_2D(A,N,N,n,n)) {
 /* Matrix initialization omitted */
   int r,s,t;
   POLYBENCH_2D_ARRAY_DECL(B, DATA_TYPE, N, N, n, n);
   for (r = 0; r < n; ++r)
      for (s = 0; s < n; ++s)
         (POLYBENCH_ARRAY(B))[r][s] = 0;
   for (t = 0; t < n; ++t)
      for (r = 0; r < n; ++r)
         for (s = 0; s < n; ++s)
            (POLYBENCH_ARRAY(B))[r][s] += A[r][t] * A[s][t];
   for (r = 0; r < n; ++r)
      for (s = 0; s < n; ++s)
         A[r][s] = (POLYBENCH_ARRAY(B))[r][s];
   POLYBENCH_FREE_ARRAY(B);
}
__attribute__((hot))
static void kernel_lu (int n, DATA_TYPE POLYBENCH_2D(A,N,N,n,n)) {
   int i, j, k;
   #pragma scop
   for (i = 0; i < _PB_N; i++) {
      for (j = 0; j <i; j++) {
         for (k = 0; k < j; k++) {
            A[i][j] -= A[i][k] * A[k][j];
         }
         A[i][j] /= A[j][j];
      }
      for (j = i; j < _PB_N; j++) {
         for (k = 0; k < i; k++) {
            A[i][j] -= A[i][k] * A[k][j];
         }
      }
   }
   #pragma endscop
}
```

Figure 13: **Case III.**The original program includes a compute-intensive initialization phase followed by an LU factorization kernel

Attribute name: __attribute__((cold))
Description: Indicates that the execution path following the annotated label is unlikely to be executed. The compiler treats the labeled path as rarely executed when generating code.
Optimization impact: Allows deprioritizing optimization effort for cold paths.

Figure 14: **Case III.** HintPilot retrieves the `cold` attribute and applies it to the initialization routine, deprioritizing optimization in a compute-intensive region.

1006 the program level, the feedback is dominated by
1007 coarse-grained runtime noise, making it impossi-
1008 ble to attribute the slowdown to the initialization
1009 phase and preventing effective refinement in sub-
1010 sequent optimization attempts. As a result, it fails
1011 to complete within the time limit because it incor-
1012 rectly marks the performance-critical routine as
1013 non-critical.