A Hybrid, Knowledge-Guided Evolutionary Framework for Personalized Compiler Auto-Tuning

Haolin Pan[1,2,3], Hongbin Zhang[2], Mingjie Xing[2,*], Yanjun Wu[2,3]

[1] Hangzhou Institute for Advanced Study at UCAS, Hangzhou, China
[2] Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China, Beijing, China
panhaolin21@mails.ucas.ac.cn,{hongbin2019,mingjie,yanjun}@iscas.ac.cn

Abstract

Compiler pass auto-tuning is critical for enhancing software performance, yet finding the optimal pass sequence for a specific program is an NP-hard problem. Traditional, general-purpose optimization flags like `-O3` and `-Oz` adopt a one-size-fits-all approach, often failing to unlock a program's full performance potential. To address this challenge, we propose a novel **Hybrid, Knowledge-Guided Evolutionary Framework**. This framework intelligently guides online, personalized optimization using knowledge extracted from a large-scale offline analysis phase. During the offline stage, we construct a comprehensive compilation knowledge base composed of four key components: (1) Pass Behavioral Vectors to quantitatively capture the effectiveness of each optimization; (2) Pass Groups derived from clustering these vectors based on behavior similarity; (3) a Synergy Pass Graph to model beneficial sequential interactions; and (4) a library of Prototype Pass Sequences evolved for distinct program types. In the online stage, a bespoke genetic algorithm leverages this rich knowledge base through specially designed, knowledge-infused genetic operators. These operators transform the search by performing semantically-aware recombination and targeted, restorative mutations. On a suite of seven public datasets, our framework achieves an average of **11.0%** additional LLVM IR instruction reduction over the highly-optimized `opt -Oz` baseline, demonstrating its state-of-the-art capability in discovering personalized, high-performance optimization sequences.

CCS Concepts

• **Software and its engineering** → **Compilers**; *Search-based software engineering*; • **Computing methodologies** → *Machine learning*.

Keywords

Compiler auto-tuning, Code optimization, Knowledge base, Pass sequence tuning

1 Introduction

The relentless pursuit of software performance is a cornerstone of modern computing, fundamentally driven by the compiler's ability to translate high-level source code into efficient machine instructions. Contemporary compilers, such as LLVM, provide a vast arsenal of hundreds of optimization passes. The specific combination and ordering of these passes—commonly known as the "phase-ordering problem"—define a search space that grows exponentially with the number of available passes (see Figure 1). Finding the optimal optimization sequence for any given program is a well-known NP-hard problem, posing a significant challenge to consistently achieving peak application performance.

Mainstream compiler optimization strategies face two primary limitations. First, default optimization levels like `-O3` employ a static, "one-size-fits-all" sequence of passes, failing to adapt to diverse program characteristics and thus leaving significant performance potential untapped [22]. Second, while auto-tuning methods such as iterative compilation can find superior, program-specific sequences [4, 8, 15, 28], they are often impractical. Their need to explore a vast combinatorial search space through costly trial-and-error compilations results in prohibitive time overheads, making them unsuitable for typical development cycles.

In recent years, machine learning (ML) has emerged as a promising paradigm for addressing the long-standing compiler optimization challenge [6, 7, 13, 17, 33]. ML-based models aim to predict effective optimization strategies directly from program features. While powerful, these approaches often treat the compiler as a black box, thereby overlooking the rich, domain-specific knowledge embedded within its internal design. Critical relationships between optimization passes—such as functional similarity, synergistic collaborations, and antagonistic interferences—are typically ignored. Moreover, these models frequently require extensive, meticulously labeled training data and continue to face persistent challenges in generalizing across new programs, diverse hardware architectures, and evolving compiler versions.

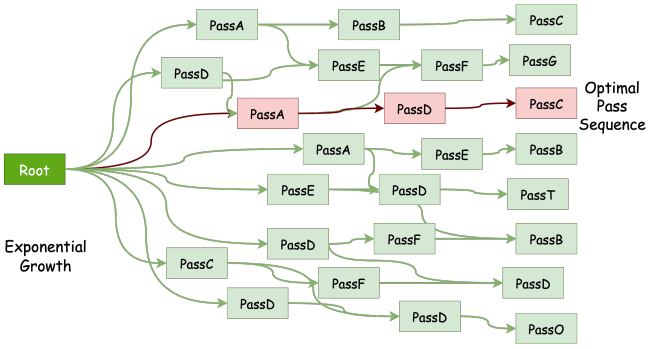

Figure 1: An illustration of the compiler phase-ordering problem. Starting from the initial program state (Root), the application of different optimization passes (e.g., PassA, PassD) creates a vast, branching search space of possible optimization sequences. The red path highlights optimal pass sequence among countless alternatives.

* Corresponding author.

To address these shortcomings, we propose a novel **Hybrid, Knowledge-Guided Evolutionary Framework**. The core tenet of our methodology is the strategic decoupling of a time-intensive, global knowledge discovery phase from a highly efficient, targeted online optimization phase. In the offline stage, we perform a large-scale, automated analysis of a diverse program corpus. Using a combination of unsupervised learning and graph-based knowledge modeling, we construct a comprehensive and structured compilation knowledge base. This knowledge base encapsulates not only deep insights into program archetypes and pass behaviors but also explicitly models the intricate relationships between them. Subsequently, in the online stage, when a new, unseen program is presented for compilation, our framework leverages this pre-computed knowledge to intelligently steer a bespoke evolutionary algorithm, enabling the rapid and effective discovery of a personalized, high-performance optimization sequence.

We rigorously evaluated our framework on seven widely-used public datasets, using the number of LLVM IR instructions as the metric for code size. The results are compelling: compared to the highly-tuned official LLVM `opt -Oz` (which aims for minimal code size) optimization level, the personalized sequences generated by our framework achieve an average additional instruction reduction of **11.0%**. This significant improvement validates our central hypothesis that intelligently guiding an online search with deeply-mined offline knowledge is an effective path toward solving the compiler optimization problem. The principal contributions of this work are as follows:

- We propose a novel, hybrid auto-tuning framework that seamlessly integrates an extensive offline knowledge extraction phase with a rapid online search. This framework automatically constructs a comprehensive compilation knowledge base, which is composed of four key, data-driven components: Pass Behavioral Vectors, behavior Pass Groups, a Synergy Pass Graph, and Prototype Pass Sequences.
- We design a set of novel, knowledge-guided genetic operators, including a crossover operator that leverages functional pass clusters for semantic recombination, and a targeted, restorative mutation operator that utilizes the knowledge base for intelligent candidate generation and parallelized online validation.
- We conduct extensive experiments, demonstrating that our framework significantly outperforms highly optimized, general-purpose compiler baselines, and consistently achieves state-of-the-art performance in personalized optimization sequence discovery tasks.

The remainder of this paper is structured as follows. We begin by surveying related work in Section 2. Section 3 provides a detailed exposition of our proposed hybrid framework, detailing both the offline knowledge-base construction and the online personalized evolution phases. In Section 4, we present our comprehensive experimental evaluation, including a comparative performance analysis against state-of-the-art methods and an in-depth ablation study. Finally, we conclude the paper in Section 5.

2 Related Work

The automatic tuning of compiler optimizations, a domain encompassing the phase-ordering and pass selection problems, has been the subject of extensive research for several decades. This section surveys pivotal developments in the field, charting the progression from foundational iterative compilation techniques to machine learning-based predictive models and the nascent application of Large Language Models (LLMs). We conclude by positioning our hybrid framework, highlighting how it synthesizes insights from these preceding paradigms while introducing unique contributions to address their inherent limitations.

2.1 Iterative Compilation

The earliest and most direct methodologies for compiler auto-tuning fall under the umbrella of iterative compilation (IC) [5]. The core principle of IC is to treat the compiler as a configurable black box, empirically exploring a vast space of optimization pass sequences for a single program. By repeatedly compiling and evaluating the program with different sequences, an optimal or near-optimal configuration for that specific program-input pair can be discovered. The exploration of this search space has been approached with a diverse array of heuristic search strategies [4, 8, 25, 28], including genetic algorithms [15], particle swarm optimization, and simulated annealing. Frameworks such as OpenTuner [2] have generalized this approach, providing extensible platforms for autotuning not only compilers but a wide range of configurable software.

While IC is capable of uncovering highly specialized and effective optimization sequences, its primary drawback is the prohibitive time cost associated with its search process. Faced with a combinatorial search space of astronomical size, IC must perform numerous, costly trial-and-error compilations and evaluations for each new program. This high compilation latency renders it impractical for integration into standard software development workflows, where rapid build times are essential. Our work acknowledges the undeniable power of empirical search but fundamentally re-architects its application. Instead of performing this exhaustive exploration at compile-time for every program, we strategically shift the heavy computational burden to a one-time, offline knowledge discovery phase. This phase builds a durable knowledge base from a large program corpus, which then serves to intelligently guide a fast and targeted online search.

2.2 ML-based Approaches

To circumvent the significant overhead of iterative compilation, the research community has increasingly turned to ML to create predictive models for compiler optimization [7, 23, 29, 31–33]. The central premise of these approaches is to learn a mapping from a representation of a program's characteristics—its features—to an effective optimization strategy. Once trained on a large dataset of programs and their corresponding effective pass sequences (often derived from an initial IC phase), these models can infer beneficial optimizations for new, unseen programs at a fraction of the cost of a full iterative search.

This paradigm has been explored through various ML techniques. Supervised learning models, for instance, have been trained to predict optimal loop unroll factors or to select from a pre-defined

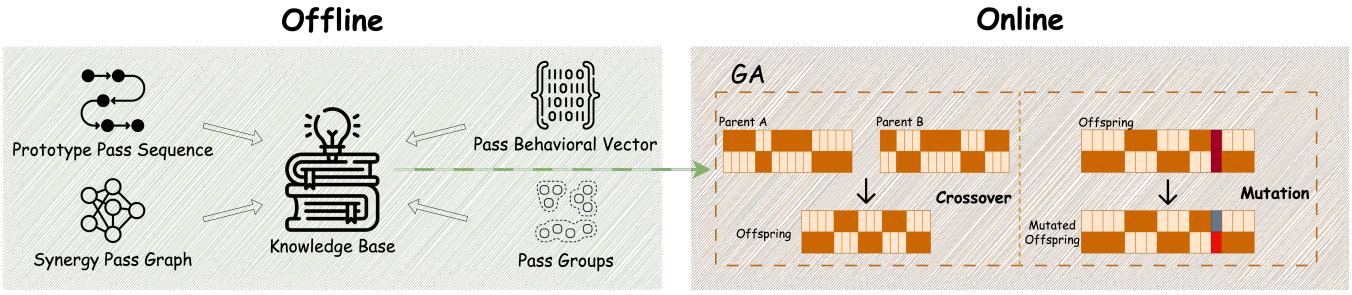

Figure 2: An overview of the proposed Hybrid, Knowledge-Guided Evolutionary Framework, which is divided into Offline and Online phases.

set of promising optimization sequences. A particularly prominent technique in this domain is reinforcement learning (RL)[30], which frames the phase-ordering challenge as a sequential decision-making problem. In this formulation, an RL agent learns a policy to navigate the optimization space by selecting one pass at a time to apply to the program's intermediate representation. Seminal works like Autophase[17] and the standardized research environment CompilerGym [11] have demonstrated the potential of RL to dynamically construct effective optimization sequences. However, ML-based approaches—especially RL—are often data-hungry and computationally expensive to train. They may also oversimplify the problem by treating optimization passes as independent actions, thus failing to capture the complex and synergistic relationships that exist among them.

2.3 LLMs in Compilation

The most recent frontier in automated compiler optimization involves the application of large language models (LLMs)[12, 20, 21]. Capitalizing on their unprecedented ability to process and generate human-like text, researchers are now exploring the potential of LLMs to "understand" source code and to suggest or generate effective optimization sequences[10, 19, 24, 27, 29]. This emerging area typically involves fine-tuning pre-trained models on vast corpora of code along with associated optimization data. The primary allure of LLMs lies in their potential to bypass traditional, handcrafted feature engineering by learning complex program representations and optimization patterns directly from raw or semi-structured code. Although this field is still in its infancy, it holds strong promise for enabling a more holistic and unified approach to program understanding. Nevertheless, LLM-based methods face several significant challenges, including immense data and computational requirements for training, as well as a general lack of interpretability in their decision-making processes.

2.4 Advantages and Novelty of Our Framework

Our proposed framework is engineered to build upon the foundational insights of these prior works while directly addressing their core limitations through a novel hybrid design. It distinguishes itself by synergistically integrating offline knowledge mining with an efficient, knowledge-guided online evolutionary search.

First, unlike pure iterative compilation, our framework decouples the expensive empirical analysis from the time-critical compilation process of a new program. The offline phase represents a one-time investment that produces a durable and structured **compilation knowledge base**. This knowledge base does not merely store effective sequences; rather, it encapsulates a deep and generalizable understanding of the optimization space, including quantitative models of pass behavior and their complex inter-relationships. Such a structured representation makes the knowledge more interpretable and broadly applicable than the single-point solutions produced by per-program iterative search approaches.

Second, in contrast to many traditional ML models that learn a direct and often opaque mapping from features to optimizations, our framework constructs an explicit and multi-faceted representation of compiler domain knowledge. By modeling not only what a pass does but also how it relates to other passes, we enable a more nuanced and "semantically-aware" search process. This explicit knowledge representation allows our online algorithm to make more informed decisions, reasoning about the functional roles and collaborative potential of different optimizations.

Finally, our framework's core innovation lies in its **knowledge-infused genetic operators**. The online evolutionary search is not a blind exploration. It is intelligently initialized with high-quality candidate sequences derived from our offline analysis. Its crossover operator reasons about functional pass clusters, and its targeted mutation operator acts as a "diagnose-and-repair" mechanism. It uses the lightweight behavioral vectors to identify underperforming segments of a sequence and then leverages the knowledge base to intelligently search for superior replacements. This hybrid approach strikes a critical balance, combining the empirical grounding of iterative methods with the efficiency of predictive models, ultimately delivering a solution that is both highly effective and practical for real-world compilation scenarios.

3 Methodology

The core of our work is a novel hybrid framework designed to automate the discovery of high-performance, personalized compiler optimization sequences. Our methodology is strategically bifurcated into two distinct yet interconnected phases: an extensive, one-time **Offline Knowledge-Base Construction** phase and a rapid, targeted **Online Personalized Evolution** phase. This architectural separation is paramount: it allows us to amortize the significant computational cost of deep program analysis and knowledge extraction over a vast corpus of programs, creating a durable

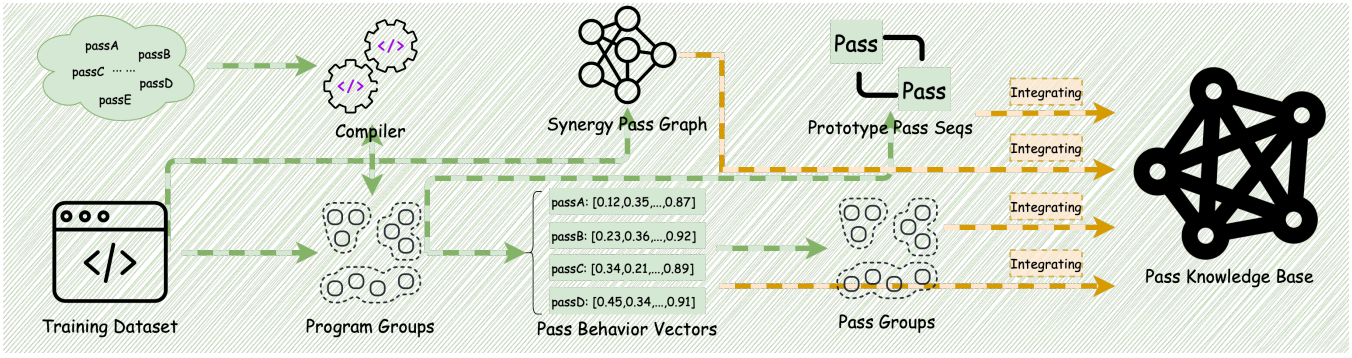

Figure 3: The offline knowledge extraction pipeline. Starting from a training dataset, we cluster programs to form prototypes. These prototypes are then used to generate a behavioral vector for each compiler pass. The behavioral vectors, in turn, are used to discover functional Pass Groups. In parallel, a Synergy Pass Graph is constructed from empirical data, and Prototype Pass Sequences are evolved for each program prototype. All four of these knowledge components are then integrated into a final, comprehensive Pass Knowledge Base.

and reusable knowledge base. This pre-computed knowledge then serves as a powerful prior to guide the lightweight, efficient online search for an optimal sequence when a new, unseen program is to be compiled.

Figure 2 depicts the overall architecture of our framework. The offline phase (Section 3.1) processes a large and diverse training set of programs to build a comprehensive compilation knowledge base. This knowledge integrates four key, independently derived components: (1) a quantitative **Pass behavioral vector** for each optimization; (2) **Pass Groups** clustering optimizations by functional similarity; (3) a **Synergy Pass Graph** modeling beneficial sequential interactions; and (4) a library of high-quality **Prototype Pass Sequences** evolved for distinct program types.

The online phase leverages this rich repository of knowledge. When a new program arrives, a bespoke evolutionary algorithm, equipped with specially designed, knowledge-guided genetic operators, is initiated. Our approach directly uses the offline-generated prototype sequences **as the initial population** for the evolutionary search, without requiring an initial selection phase. These pre-vetted, high-quality sequences serve as strong starting points. Throughout the search, the algorithm continues querying the knowledge base to perform semantically aware crossover and execute targeted, restorative mutations, enabling rapid convergence on a high-quality, personalized optimization sequence.

3.1 Offline Knowledge-Base Construction

The offline phase of our framework is designed to systematically distill a large, unstructured corpus of programs, $\mathcal{P}_{train}$, into a structured and actionable compilation knowledge base, $\mathcal{K}$. The core design principle is to pre-compute and store generalizable knowledge, thereby transforming the online optimization task from a blind search into an informed, guided exploration. This knowledge base integrates four key, independently-derived knowledge components. Figure 3 provides a high-level overview of this construction pipeline, which consists of four primary stages, each producing a distinct knowledge asset that is subsequently integrated into the final pass knowledge base.

3.1.1 Pass Behavioral Vector. To enable effective, data-driven reasoning about compiler passes, we introduce a method to capture their true, context-dependent behavior. The process begins by partitioning the program corpus $\mathcal{P}_{train}$ into N distinct **program prototypes**. This is achieved by applying K-Means clustering to their Autophase feature vectors [17], which are normalized prior to clustering to ensure sensitivity to their proportional composition. The resulting clustering model, which we denote as $\mathcal{M}_{prog}$, is preserved for the online phase.

With these prototypes, we define the **Pass Behavioral Vector** for a pass π, denoted $\mathbf{f}_\pi$, as an N-dimensional vector where each component $(\mathbf{f}_\pi)_i$ quantifies the average effectiveness of π on programs of prototype C_i. Effectiveness is measured as the percentage reduction in LLVM IR instructions:

$$(\mathbf{f}_\pi)_i = \mathbb{E}_{p \in C_i}\left[\frac{\text{IR}_{\text{before}}(p) - \text{IR}_{\text{after}}(p,\pi)}{\text{IR}_{\text{before}}(p)} \times 100\right] \tag{1}$$

where $\text{IR}_{\text{before}}$ and IR_{after} are the instruction counts before and after applying pass π.

The resulting library of behavioral vectors, $\mathcal{F}_{pass}$, provides a rich semantic space for pass analysis. Figure 4 visualizes these vectors for a selection of passes, revealing several critical insights. First, the wide performance distribution of most passes confirms their high **context-dependency**, validating the need for personalized optimization. Second, passes exhibit distinct **behavioral profiles**, ranging from the consistently conservative to the high-risk, high-reward. Third, the existence of both positive and negative performance regions for nearly every pass underscores the complexity of the optimization landscape and justifies the necessity of a knowledge-guided search strategy.

Beyond their immediate use in our framework, these vectors serve as a general tool for compiler introspection. They can be used, for instance, to automatically identify risky passes with high performance variance, detect functionally redundant passes via correlation analysis, and provide a quantitative basis for developing more sophisticated, context-aware compiler heuristics.

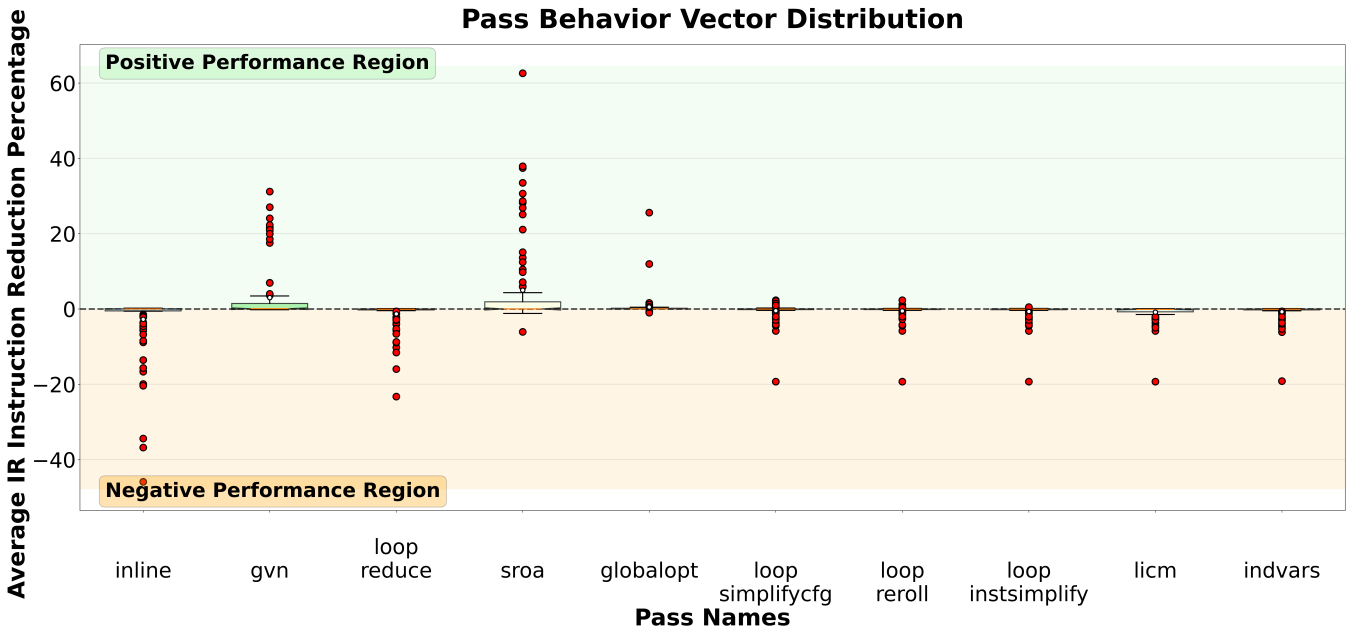

Figure 4: Distribution of behavioral vector values for a selection of compiler passes. Each boxplot illustrates the performance variation of a single pass across all N program prototypes, highlighting the high context-dependency of pass effectiveness and their diverse behavioral profiles.

3.1.2 Functional Pass Groups. While behavioral vectors provide a detailed view, a higher-level abstraction is needed for efficient reasoning. We aim to identify groups of passes that are functionally similar, as this knowledge can guide substitution and recombination in the online search. We posit that passes exhibiting similar behavioral vectors are likely to perform analogous transformations. Consequently, we discover these **Pass Groups**, denoted $C_{pass} = \{G_1, \ldots, G_k\}$, by applying K-Means clustering directly to the set of all pass behavioral vectors, $\mathcal{F}_{pass}$. The algorithm seeks to minimize the total intra-cluster variance:

$$\arg\min_{C_{pass}} \sum_{j=1}^{k} \sum_{\mathbf{f}_\pi \in G_j} ||\mathbf{f}_\pi - \boldsymbol{\mu}_j||^2 \tag{2}$$

where $\boldsymbol{\mu}_j$ is the centroid of group G_j. The optimal number of clusters, k, is determined automatically using the Elbow Method. By partitioning the pass space in this manner, we create a set of semantic categories, allowing our online genetic operators to operate on functionally coherent blocks of passes rather than on individual, unrelated ones, leading to a more structured exploration.

3.1.3 Synergy Pass Graph. Effective optimization sequences are not merely collections of good passes; their ordering is paramount. To capture the critical, directional relationships between passes, we construct a **Synergy Pass Graph** [28]. This graph is designed to explicitly model enabling relationships, where one pass creates opportunities for another.

We formally define an ordered pair of passes (π_A, π_B) as synergistic for a given program p if applying the sequence $\langle \pi_A, \pi_B \rangle$ yields a greater instruction reduction than applying π_B alone, under the precondition that π_B is itself beneficial. This relationship is captured by the following condition:

$$\text{IR}(p, \langle \pi_A, \pi_B \rangle) < \text{IR}(p, \langle \pi_B \rangle) < \text{IR}(p, \langle\rangle) \tag{3}$$

where $\text{IR}(p, S)$ denotes the instruction count of program p after applying the optimization sequence S, and $\langle\rangle$ represents the empty sequence (the original program).

To build a robust, global model of these interactions, we systematically identify all such synergistic pairs for *every* program p in the entire training corpus $\mathcal{P}_{train}$. These individually discovered pairs are then **aggregated into a single, comprehensive graph**. The nodes of the Synergy Pass Graph are the passes Π, and a directed edge exists from π_A to π_B if the pair was identified as synergistic in at least one training program. The weight of each edge (π_A, π_B) is proportional to its frequency of occurrence across the corpus, reflecting the strength and generality of the synergistic relationship. The resulting graph, shown as the "Synergy Pass Graph" component in Figure 3, encapsulates a wealth of empirically-validated sequential heuristics, providing a strong, domain-specific prior to guide the construction of logically sound and potent optimization sequences during the online phase.

3.1.4 Prototype Pass Sequences. To mitigate the risk of a cold start and accelerate convergence during the online phase, our framework prepares a library of high-quality initial solutions. This is achieved by evolving **Prototype Pass Sequences**. The rationale is that a sequence optimized for a class of similar programs serves as a much stronger starting point than a random one. For each of the N program prototypes, we execute a dedicated Genetic Algorithm to find the optimal prototype sequence, $\boldsymbol{\pi}_i^*$. The objective is to maximize the average instruction reduction across all programs in

prototype C_i:

$$\boldsymbol{\pi}_i^* = \arg\max_{\boldsymbol{\pi} \in \Pi^L} \left(\frac{1}{|C_i|} \sum_{p \in C_i} \text{Reduction}(p, \boldsymbol{\pi}) \right) \tag{4}$$

where Π^L is the space of all pass sequences of a fixed length L, and $\text{Reduction}(p, \boldsymbol{\pi})$ is the performance gain from applying sequence $\boldsymbol{\pi}$ to program p. This intensive, prototype-specific offline search yields a library of pre-vetted "seed" sequences, $\mathcal{L}_{proto}$, enabling the online phase to begin its search from regions of the solution space already known to be highly effective.

3.2 Online Personalized Evolution

With the comprehensive, offline-constructed knowledge base $\mathcal{K}$ in place, the framework is ready to perform its primary function: rapidly discovering a high-performance, personalized optimization sequence for a new, unseen program. This is the role of the **Online Personalized Evolution** phase. The design philosophy of this phase is to be lightweight, fast, and maximally guided by the pre-computed offline knowledge, transforming the search from a blind exploration into an informed, goal-oriented process.

The online process commences with two preparatory steps. First, upon receiving a new program p_{new}, we perform **rapid profiling** using the pre-trained model $\mathcal{M}_{prog}$ to instantly classify it into its corresponding program prototype, $C_{i_{new}}$. This classification, $i_{new} = \text{Predict}(\mathcal{M}_{prog}, \mathbf{v}_{new})$, is the critical link that unlocks the relevant segment of our knowledge base. Second, we perform a **smart initialization** of the evolutionary algorithm's population. Instead of generating random sequences, we leverage the library of empirical prototype sequences, $\mathcal{L}_{proto}$. We quickly evaluate all sequences in $\mathcal{L}_{proto}$ on p_{new} once and select the top-k best-performing ones to form the initial population, P_0. This strategy ensures our search begins from a set of high-quality solutions known to be effective for programs of this type.

The core of the online phase is an iterative search performed by our custom-designed Genetic Algorithm (GA). The fitness of any given sequence $\boldsymbol{\pi}$ is formally defined as its percentage reduction in instruction count on p_{new} relative to the `opt -Oz` baseline:

$$\text{Fitness}(\boldsymbol{\pi}, p_{new}) = \frac{\text{IR}_{\text{Oz}}(p_{new}) - \text{IR}_{\boldsymbol{\pi}}(p_{new})}{\text{IR}_{\text{Oz}}(p_{new})} \times 100 \tag{5}$$

The defining characteristic of our GA lies in its specialized genetic operators, which we detail below.

Knowledge-Guided Crossover. Traditional crossover operators are semantically blind, often disrupting beneficial, co-evolved building blocks (schemata). To overcome this, we introduce a **Knowledge-Guided Crossover** operator that leverages the functional structure of optimization sequences.

As illustrated in Figure 5, the operator first partitions a parent sequence $\boldsymbol{\pi}$ into a sequence of "functional blocks" $\langle B_1, B_2, \ldots, B_m \rangle$, where each block B_j is a contiguous subsequence of passes belonging to the same functional cluster. The *a priori* effectiveness of a block B_j for the current program's prototype i_{new} is estimated by aggregating the behavioral vector scores of its constituent passes:

$$\text{Score}(B_j, i_{new}) = \sum_{\pi_l \in B_j} (\mathbf{f}_{\pi_l})_{i_{new}} \tag{6}$$

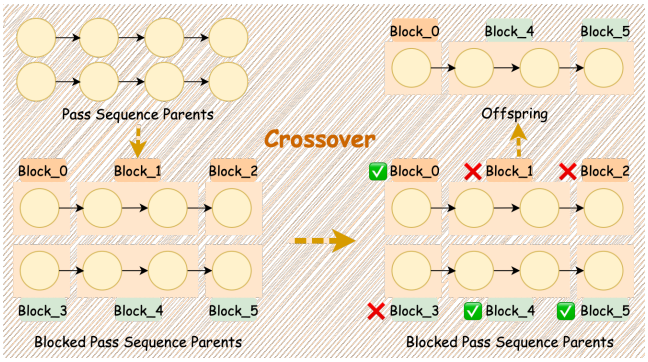

Figure 5: Conceptual diagram of the Knowledge-Guided Crossover operator. Parent sequences are first segmented into functional blocks. The operator then probabilistically selects and recombines entire blocks, with the selection probability biased by the block's pre-computed effectiveness score (Eq. 6) for the target program's prototype.

where $(\mathbf{f}_{\pi_l})_{i_{new}}$ is the i_{new}-th component of the vector for pass π_l. When creating an offspring from two parents, $\boldsymbol{\pi}_a$ and $\boldsymbol{\pi}_b$, for each corresponding block position j, the block from parent $\boldsymbol{\pi}_a$ is chosen with a probability proportional to its estimated score. Specifically, the probability of selecting block $B_{j,a}$ from parent $\boldsymbol{\pi}_a$ is given by:

$$P(\text{select } B_{j,a}) = \frac{\text{Score}(B_{j,a}, i_{new})}{\text{Score}(B_{j,a}, i_{new}) + \text{Score}(B_{j,b}, i_{new})} \tag{7}$$

(Scores are normalized to ensure positivity). This mechanism promotes the preservation and propagation of functionally coherent and historically effective groups of passes, leading to a more meaningful exploration of the search space.

Targeted, Restorative Mutation. Standard mutation operators introduce random perturbations, a process that is often destructive and inefficient. To address this, we designed a **Targeted, Restorative Mutation** operator, a sophisticated, multi-stage "diagnose-and-repair" procedure depicted in Figure 6.

(1) **Rapid Diagnosis:** First, the operator diagnoses the sequence to identify its "weakest link." It performs a lightweight scan, calculating the estimated effectiveness score (using Equation 6) for each functional block in the sequence. The block B_{worst} with the minimum score is identified as the target for mutation:

$$B_{worst} = \arg\min_{B_j} \text{Score}(B_j, i_{new}) \tag{8}$$

(2) **Intelligent Candidate Generation:** Second, rather than selecting a random replacement, the operator intelligently constructs a high-quality candidate pool, denoted as C_{pool}. This pool is populated by querying the Pass Knowledge Base and prioritizes passes that exhibit a strong, directed `synergy` relationship with the pass immediately preceding the target block. To ensure sufficient diversity, the pool is further augmented with passes that share a high `similarity` to those within the block being replaced (i.e., from the same functional cluster).

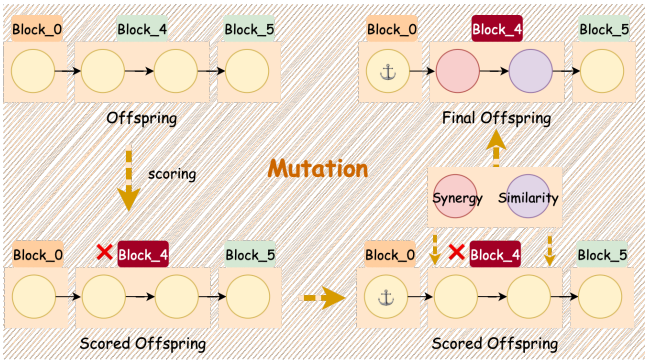

Figure 6: The Targeted, Restorative Mutation workflow. (1) A sequence is diagnosed using lightweight behavioral vector scores (Eq. 6) to identify the "weakest link" block. (2) The Pass Knowledge Base is queried to build a high-quality candidate pool based on synergy and similarity. (3) A new, empirically superior block is formed and used to replace the weakest link.

(3) **Empirical Validation and Repair:** Finally, a small set of new candidate blocks, $\{\hat{B}_1, \ldots, \hat{B}_q\}$, are randomly sampled from C_{pool}. These few, promising candidates are then evaluated in parallel via actual compilation on p_{new}. The best-performing candidate block, $\hat{B}^*$, is identified:

$$\hat{B}^* = \arg\max_{\hat{B}_r} \text{Fitness}(\boldsymbol{\pi}_{\text{new},r}, p_{new}) \tag{9}$$

where $\boldsymbol{\pi}_{\text{new},r}$ is the sequence with B_{worst} replaced by $\hat{B}_r$. Only if the fitness of the sequence containing $\hat{B}^*$ is strictly greater than the fitness of the original sequence is the mutation accepted.

This procedure transforms mutation from a blind, random perturbation into a targeted, data-driven, and restorative enhancement, improving search efficiency. After the final generation, the algorithm returns the best sequence found, representing a highly-optimized, personalized solution discovered in a fraction of the time required by traditional iterative methods.

4 Experiments

4.1 Main Experiment

This section presents a comprehensive empirical evaluation of our proposed framework. We begin by detailing the experimental setup, including the environment, datasets, and baseline methods used for comparison. We then present and thoroughly analyze the main experimental results, comparing the performance of our framework against a suite of state-of-the-art, widely recognized compiler auto-tuning techniques.

4.1.1 *Experimental Setup.*

Compiler Environment. All experiments were conducted using the LLVM compiler infrastructure, version 10.0.0. Our framework interacts with LLVM's `opt` tool to apply sequences of optimization passes and to extract the final instruction counts. The full set of optimization passes available in this LLVM version constitutes the action space for our evolutionary search. The experiments were executed on a high-performance computing system equipped with an AMD EPYC 7763 64-Core Processor, ensuring sufficient computational resources for both the offline knowledge extraction and the parallelized online evaluations.

Datasets. To ensure a robust and generalizable evaluation, we utilized a comprehensive collection of C and C++ benchmarks, primarily sourced from the CompilerGym environment [11]. The dataset is strategically divided into a large training set for the offline phase and a distinct test set for the final evaluation, as detailed in Table 1. The training set, composed of **19,603** program files from sources such as `github-v0`, `linux-v0`, and `poj104-v0`, is exclusively used for building our knowledge base. The test set consists of **335** unseen program files across seven diverse and curated benchmark suites (`blas`, `cbench`, `chstone`, `mibench`, `npb`, `opencv`, and `tensorflow`), strictly reserved for evaluating the performance and generalization capabilities of our framework.

Table 1: Dataset Composition detailing the number of programs for training and testing from each source dataset.

Type	Dataset	Train	Test
Uncurated	blas	133	29
	github-v0	7,000	0
	linux-v0	4,906	0
	opencv-v0 [9]	149	32
	poj104-v1 [26]	7,000	0
	tensorflow-v0 [1]	415	90
Curated	cbench-v1 [14]	0	11
	mibench-v1 [16]	0	40
	chstone-v0 [18]	0	12
	npb-v0 [3]	0	121
Total	–	**19,603**	**335**

Baselines. To rigorously assess the effectiveness of our framework, we compare it against a wide array of established baseline methods. These baselines represent the state-of-the-art in both search-based iterative compilation and machine learning-based approaches. The selected baselines include:

- **Iterative and Search-Based Methods:** OpenTuner [2], a standard Genetic Algorithm (GA) [15], TPE [4], RIO [8], CompTuner [33], and BOCA [7]. These methods represent various heuristic search strategies applied to the phase-ordering problem.
- **Machine Learning-Based Methods:** CFSAT [28] and Coreset-NVP [24], which are recent, strong performers in this domain, as well as Autophase [17] with its reinforcement learning PPO-LSTM and PPO-noLSTM variants.
- **Large Language Model-Based Method:** To position our work in relation to the latest advancements, we include **Compiler-R1** [27] as a representative LLM-based baseline. This method leverages a large language model to automatically generate optimization sequences. In our experiments,

Table 2: Average OverOz (%) Improvement and Time (s) per Program Comparison. Higher OverOz is better. Bold indicates the best performance in each column.

Method	blas-v0	cbench-v1	chstone-v0	mibench-v1	npb-v0	opencv-v0	tensorflow-v0	Avg.	Time (s)
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	593
TPE	−2.24	0.97	7.60	0.20	24.62	1.46	1.23	4.83	905
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	10800
BOCA	−2.36	−0.16	3.18	−0.69	22.87	1.13	1.22	3.60	3310
CFSAT	4.60	5.80	11.90	3.70	25.90	**5.90**	6.10	9.13	6
Coreset-NVP	2.60	3.50	9.30	1.70	9.80	5.20	6.10	5.46	11
Compiler-R1	5.27	5.52	9.08	6.67	22.44	4.52	5.72	8.46	29
Autophase (PPO-LSTM)	−1.12	5.60	4.49	4.41	−4.67	−0.09	0.05	1.24	3
Autophase (PPO-noLSTM)	−4.77	−79.69	−80.90	−107.33	−76.69	−2.32	−0.76	−50.35	**2**
Ours	**6.15**	**6.80**	**12.57**	**9.02**	**29.74**	5.82	**6.93**	**11.00**	10

the timing results for Compiler-R1 were measured on a system equipped with four NVIDIA H100 GPUs, capturing the total time required for model inference using Qwen-7B, along with the subsequent compiler tool invocation.

The primary baseline for performance comparison is LLVM's official size-optimization level, `opt -Oz`.

Evaluation Metric. The primary performance metric used in our evaluation is the **Average OverOz (%) Improvement**. This metric quantifies the additional percentage reduction in the number of LLVM IR instructions achieved by a given method compared to the `opt -Oz` baseline. It is formally defined as:

$$\text{OverOz (\%)} = \frac{1}{|\mathcal{P}_{test}|} \sum_{p \in \mathcal{P}_{test}} \frac{I_{\text{Oz}}(p) - I_{\pi^*}(p)}{I_{\text{Oz}}(p)} \times 100$$

where $\mathcal{P}_{test}$ is the set of programs in a test dataset, $I_{\text{Oz}}(p)$ is the instruction count of program p after applying the `opt -Oz` sequence, and $I_{\pi^*}(p)$ is the instruction count after applying the optimization sequence π^* found by the method being evaluated. A higher OverOz value indicates a better optimization performance.

4.2 Results

Table 2 presents the main experimental results, comparing the performance of our framework against all baseline methods across the seven test datasets. The table reports the Average OverOz (%) improvement for each method on each dataset, the overall average improvement, and the average time taken per program to find the optimization sequence.

Overall Performance. The results clearly demonstrate the superior performance of our proposed framework. As shown in the "Avg." column, our method achieves an average OverOz improvement of **11.00%**, significantly outperforming all other baselines. This represents a substantial advancement over the next-best method, CFSAT, which achieved a 9.13% average improvement. A crucial factor contributing to this performance gap is the comprehensive nature of the search space explored. Many search-based baselines, such as TPE, BOCA, CompTuner, and Opentuner, primarily focus on selecting an effective *combination* (or subset) of passes, often applying them in a fixed or heuristically determined order. In contrast, our evolutionary framework is explicitly designed to simultaneously optimize both the selection of passes and their precise sequential ordering. This holistic approach allows our method to uncover complex, synergistic interactions that are inaccessible to methods that decouple these two tightly intertwined aspects of the phase-ordering problem. This strong overall performance underscores the effectiveness of our hybrid, knowledge-guided approach in consistently finding high-quality optimization sequences.

Performance on Individual Datasets. Our framework's strength is not limited to its average performance; it consistently delivers top-tier results on individual benchmark suites. It achieves the best performance on five out of the seven datasets: `blas` (6.15%), `cbench` (6.80%), `mibench` (9.02%), `npb` (29.74%), and `tensorflow` (6.93%). The particularly strong results on computationally intensive benchmarks like `npb` further underscore the advantage of our holistic search, as such programs are often highly sensitive to the precise ordering of optimizations. In contrast, many traditional search-based methods and even some ML-based approaches show negative results on certain datasets, indicating that their found sequences were worse than the default `opt -Oz`, a pitfall our robust framework successfully avoids.

Efficiency Analysis. In addition to its superior optimization quality, our framework demonstrates a remarkable balance of efficiency and effectiveness. The reported timings in Table reftab:main_results represent the average time required for each method to converge on its final reported solution. Our framework requires an average of only **10 seconds** per program to find its high-quality sequences.

This efficiency is particularly notable when contextualized within different methodological categories. When compared to other **iterative search-based methods**, our framework is not only the most effective but also among the fastest. Traditional iterative compilers like CompTuner (10800s) and BOCA (3310s) require orders of magnitude more time to converge, often because they intertwine their search process with on-the-fly model training or complex heuristic evaluations. Our framework, by contrast, front-loads all expensive learning into the one-time offline phase, making the online search

Table 3: Ablation study results showing the average percentage of IR instruction reduction over `opt -Oz`. Each row represents a variant of our method, and columns correspond to different benchmark suites. The final column shows the average performance across all datasets. Higher values are better.

Method Variant	cbench-v1	chstone-v0	mibench-v1	npb-v0	blas-v0	opencv-v0	tensorflow-v0	Average
Full (Our Method)	**6.80**	**12.57**	**9.02**	**29.74**	**6.15**	**5.82**	**6.93**	**11.0**
Random Init	4.93	8.99	6.36	27.94	2.21	4.92	6.45	8.8
No-KC (No Knowledge Crossover)	6.60	11.69	7.90	29.51	5.62	5.68	6.75	10.5
No-KM (No Knowledge Mutation)	6.42	10.83	7.66	29.19	5.18	5.68	6.75	10.2
No Knowledge	-4.13	0.19	-0.95	16.81	-1.32	1.97	2.22	2.1

a pure, guided evaluation process. When compared to **ML-based predictive methods**, such as CFSAT (6s), our framework's slightly longer search time is an expected trade-off for its higher optimization quality. Methods like CFSAT primarily rely on a single, fast forward pass through a pre-trained model to predict a sequence. Our approach, while also guided by a pre-trained knowledge base, is fundamentally an **iterative compilation process** that performs multiple, empirical evaluations on the target program. This iterative refinement allows our framework to fine-tune a sequence specifically for the given program, uncovering optimizations that a pure predictive model might miss. Consequently, our framework achieves the best performance among all iterative methods while maintaining a convergence time that is competitive and practical, positioning it as a highly viable solution for real-world compiler optimization challenges.

4.3 Ablation Study

To rigorously evaluate the efficacy of the individual components within our knowledge-guided evolutionary framework, we conducted a comprehensive ablation study. The primary objective of this study is to dissect the framework and quantify the performance contribution of each of our proposed knowledge-infused mechanisms: (1) the smart, prototype-based initialization, (2) the knowledge-guided crossover operator, and (3) the targeted, restorative mutation operator. By systematically disabling these components, we can isolate their impact and validate their necessity.

4.3.1 Experimental Setup. To systematically evaluate the contributions of our framework's components, we conduct a comprehensive ablation study. Our complete proposed method, denoted as **Full**, is compared against four specialized, ablated variants. First, to measure the direct benefit of a strong starting point, the **Random-Init** variant disables our smart initialization strategy, replacing the high-quality empirical prototype sequences with a randomly generated population while all other knowledge-guided operators remain active. Second, to isolate the contribution of our semantically-aware block recombination, the **No-KC** (No Knowledge Crossover) variant substitutes our knowledge-guided crossover with a standard, semantically-blind single-point crossover, while retaining smart initialization and knowledge-guided mutation. Similarly, the **No-KM** (No Knowledge Mutation) variant measures the impact of our "diagnose-and-repair" mechanism by replacing our targeted, restorative mutation with a conventional random mutation operator. Finally, the **No-Knowledge** variant serves as our primary baseline, disabling all knowledge-guided components, thereby allowing us to quantify the total performance gain attributable to our entire knowledge-guidance paradigm. For all experiments, core GA hyperparameters were held constant for a fair comparison, and performance was measured as the average percentage reduction in LLVM IR instructions relative to the highly-optimized `-Oz` baseline.

4.3.2 Results and Analysis. The results of our ablation study are summarized in Table 3, which reports the final converged performance of each variant. To provide deeper insight into the search process itself, Figure 7 illustrates the convergence behavior of each method on a selection of representative benchmarks.

The convergence curves clearly visualize the impact of our knowledge-guided components. The **Full** method consistently starts from a high initial performance and converges rapidly to the best solution, a direct result of its smart initialization and efficient search operators. The ablated variants exhibit progressively degraded convergence behavior, with the `No Knowledge` often starting from a significantly negative performance and struggling to improve. This visual evidence reinforces the quantitative results in the table, demonstrating not only that our full method achieves superior final performance, but also that its search process is fundamentally more efficient and better-directed. The subsequent analysis will focus on the final performance metrics from the table.

The Impact of Smart Initialization. Comparing the **Full** method with the **Random Init** variant reveals the substantial benefit of our smart initialization strategy. Across all benchmarks, replacing prototype-based initialization with random initialization leads to a significant performance drop, with the average reduction declining from approximately 11.0% to 8.8% (a relative decrease of 20%). For instance, on 'chstone-v0', performance falls from 12.41% to 8.99%. This confirms that starting the search from a set of high-quality, pre-vetted sequences provides a critical advantage, allowing the GA to focus its efforts on refining already strong solutions rather than discovering them from scratch.

The Contribution of Knowledge-Guided Operators. The contributions of our specialized crossover and mutation operators become clear when comparing the **Full** method to the **No-KC** and **No-KM** variants. Disabling the knowledge-guided crossover (**No-KC**) results in a consistent, though moderate, performance degradation. For example, on `mibench-v1`, performance drops from 9.02% to 7.90%. This indicates that our semantically aware block recombination effectively preserves and propagates beneficial building blocks, contributing to a more efficient search. The removal of knowledge-guided mutation (**No-KM**) causes an even more pronounced decline

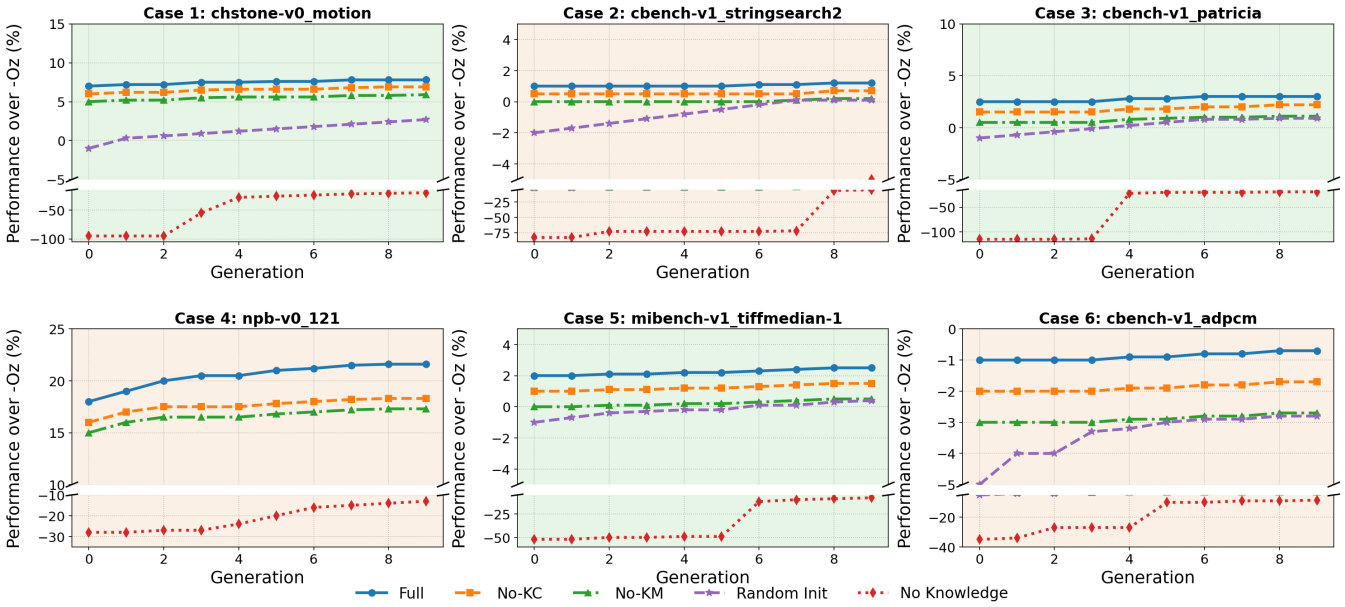

Figure 7: Convergence curves on representative benchmarks, demonstrating the critical role of our knowledge-guided components. Our Full method shows rapid convergence to superior solutions, while disabling knowledge—particularly smart initialization—results in a significantly less efficient search.

than removing the intelligent crossover. On `chstone-v0`, performance decreases from 12.57% to 10.83%. This suggests that our "diagnose-and-repair" mutation mechanism, which intelligently targets and fixes weak points in a sequence, is a more critical contributor to finding high-quality solutions than the crossover operator. It actively enhances solutions, whereas crossover primarily recombines existing genetic material.

The Overall Value of Knowledge Guidance. The most dramatic performance drop occurs in the **Standard GA** variant, where all knowledge-guided components are disabled. In this scenario, the performance plummets, even becoming significantly worse than the `opt -Oz` baseline on several benchmarks (e.g., -4.13% on 'cbench-v1'). This stark contrast highlights the immense value of our complete knowledge-guidance paradigm. Without the priors from the offline knowledge base, a standard GA struggles to navigate the vast and complex optimization space effectively within a limited time budget. The consistently positive and superior results of our **Full** method underscore the conclusion that a holistic, knowledge-driven approach is essential for achieving state-of-the-art performance in personalized compiler optimization.

5 Conclusion

In this paper, we presented a **Hybrid, Knowledge-Guided Evolutionary Framework** for the compiler phase-ordering problem, designed to balance optimization quality with practical compilation overhead. Our approach is founded on the strategic decoupling of an extensive offline knowledge extraction phase from a knowledge-driven online search. The offline phase automatically constructs a multi-faceted Pass Knowledge Base from a large program corpus. This knowledge base provides a quantitative, data-driven understanding of the optimization space by modeling: (1) the context-dependent performance of passes via Behavioral Vectors, (2) their functional relationships through Pass Groups, (3) their sequential dependencies in a Synergy Graph, and (4) effective initial solutions via Prototype Sequences. This pre-computed knowledge is then utilized by a bespoke online evolutionary algorithm, whose knowledge-infused genetic operators enable an efficient and structured exploration of the solution space.

Empirically validated on seven public datasets, our framework achieved an average additional instruction count reduction of 11.0% over the highly-optimized `-Oz` baseline, confirming its practicality and effectiveness. Future work will extend this framework to multi-objective optimization (e.g., execution time, energy) and explore integrating learned program representations from large language models. In conclusion, our work demonstrates that a deep, offline analysis of compiler behavior, when systematically leveraged by a lightweight online search, provides a promising path towards more intelligent and adaptive self-tuning compilers.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

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

[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. 1991. The NAS parallel benchmarks. *The International Journal*

of Supercomputing Applications 5, 3 (1991), 63–73.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. 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. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on profile and feedback-directed compilation*.

[6] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. 2020. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 15–25.

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

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

[9] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. 2012. A brief introduction to OpenCV. In *2012 proceedings of the 35th international convention MIPRO*. IEEE, 1725–1730.

[10] 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. arXiv:2407.02524 [cs.PL] https://arxiv.org/abs/2407.02524

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

[12] DeepSeek-AI DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z.F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J.L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R.J. Chen, R.L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S.S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W.L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X.Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y.K. Li, Y.Q. Wang, Y.X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y.X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z.Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. (Jan 2025).

[13] Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. 2024. CompilerDream: Learning a Compiler World Model for General Code Optimization. *arXiv preprint arXiv:2404.16077* (2024).

[14] Grigori Fursin. 2009. Collective Tuning Initiative: automating and accelerating development and optimization of computing systems. In *GCC Developers' Summit*.

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

[16] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. 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)*. IEEE, 3–14.

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

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

[19] Davide Italiano and Chris Cummins. 2024. Finding Missed Code Size Optimizations in Compilers using LLMs. *arXiv preprint arXiv:2501.00655* (2024).

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

[21] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. arXiv:2503.09516 [cs.CL] https://arxiv.org/abs/2503.09516

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

[23] Hugh Leather and Chris Cummins. 2020. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 1–8.

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

[25] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative compilation optimization based on metric learning and collaborative filtering. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–25.

[26] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[27] Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. 2025. Compiler-R1: Towards Agentic Compiler Auto-tuning with Reinforcement Learning. *arXiv preprint arXiv:2506.15701* (2025).

[28] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. 2025. Towards Efficient Compiler Auto-tuning: Leveraging Synergistic Search Spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 614–627.

[29] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. 2022. SRTuner: Effective compiler optimization customization by exposing synergistic relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 118–130.

[30] Nadav Rotem and Chris Cummins. 2021. Profile guided optimization without profiles: A machine learning approach. *arXiv preprint arXiv:2112.14679* (2021).

[31] Hafsah Shahzad, Ahmed Sanaullah, Sanjay Arora, Ulrich Drepper, and Martin Herbordt. 2024. A Neural Network Based GCC Cost Model for Faster Compiler Tuning. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[32] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[33] Mingxuan Zhu, Dan Hao, and Junjie Chen. 2024. Compiler Autotuning through Multiple-phase Learning. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–38.