GRACE: Globally-Seeded Representation-Aware Cluster-Specific Evolution for Compiler Auto-Tuning

HAOLIN PAN, Hangzhou Institute for Advanced Study at UCAS, China, Institute of Software, Chinese Academy of Sciences, China, and University of Chinese Academy of Sciences, China
CHAO ZHA, Institute of Computing Technology, Chinese Academy of Sciences, China, Research Center for High Efficiency Computing Infrastructure, Zhejiang Lab, China, and University of Chinese Academy of Sciences, China
JINYUAN DONG, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
MINGJIE XING*, Institute of Software, Chinese Academy of Sciences, China
YANJUN WU, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China

Compiler pass selection and phase ordering present a significant challenge in achieving optimal program performance, particularly for objectives like code size reduction. Standard compiler heuristics offer general applicability but often yield suboptimal, program-specific results due to their one-size-fits-all nature. While iterative compilation can find tailored solutions, its prohibitive search cost limits practical use. Machine learning approaches promise faster inference but frequently struggle with generalization to unseen programs. This paper introduces GRACE, a novel framework for compiler auto-tuning, demonstrated for LLVM IR instruction count optimization. GRACE effectively curtails the search space by leveraging pass synergies and a weighted scoring method to generate initial high-quality candidate sequences and a pass pool. It then employs contrastive learning, using pass sequence-based data augmentation, to create program embeddings that facilitate similarity-aware clustering. Evolutionary search within these clusters yields a coreset of k specialized pass sequences designed for robust generalization to unseen programs. At test time, GRACE efficiently selects the best coreset sequence and refines it using lightweight techniques. Experimental results on seven diverse datasets show that GRACE reduces LLVM IR instruction count by an average of **10.09%** on LLVM 10.0.0 and **10.19%** on LLVM 18.1.6 compared to `opt -Oz`, while incurring an average tuning time of less than **1s** per program, demonstrating its state-of-the-art performance and practical effectiveness.

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

Additional Key Words and Phrases: Compiler auto-tuning, Code optimization, Knowledge base, Pass sequence tuning

1 Introduction

Compilers play a pivotal role in modern software development, with one of their core responsibilities being the transformation of source code through a series of optimization passes to enhance target program characteristics such as execution speed, energy consumption, or code size. Among these

*Corresponding Author.

Authors' Contact Information: Haolin Pan, Hangzhou Institute for Advanced Study at UCAS, Hangzhou, China and Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, panhaolin21@mails.ucas.ac.cn; Chao Zha, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China and Research Center for High Efficiency Computing Infrastructure, Zhejiang Lab, Zhejiang, China and University of Chinese Academy of Sciences, Beijing, China, zhachao21@mails.ucas.ac.cn; Jinyuan Dong, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, dongjinyuan24@mails.ucas.ac.cn; Mingjie Xing, Institute of Software, Chinese Academy of Sciences, Beijing, China, mingjie@iscas.ac.cn; Yanjun Wu, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, yanjun@iscas.ac.cn.

objectives, code size optimization is crucial, especially for resource-constrained embedded systems, mobile applications sensitive to distribution and loading times, and scenarios demanding efficient instruction cache utilization. Traditional compiler infrastructures such as LLVM [23], characterized by their modular design and flexible pass management frameworks, offer developers extensive optimization passes and customization capabilities. To simplify usage for typical scenarios, LLVM further provides a set of predefined optimization levels (e.g., `-O0` to `-O3` for execution speed, and `-Os`/`-Oz` for code size), serving as general-purpose trade-offs between compile time and performance. However, such predefined optimization levels often fail to take advantage of program-specific characteristics, leading to suboptimal results. This suboptimality arises directly from the profound complexity of compiler optimization: both the selection and ordering of passes can significantly impact the final output. This challenge, known as the phase-ordering and pass-selection problem, is NP-hard, as the same combination may yield widely varying results for different programs, thus making a universally optimal, predefined set of optimizations elusive. To address this formidable challenge and move beyond general-purpose compromises towards truly program-specific optimization, researchers have explored more sophisticated methodologies.

Iterative Compilation (IC) techniques offer the promise of highly tailored, program-specific optimizations by exploring numerous compiler pass sequences, yet they are significantly hampered by their prohibitive search cost and computational demands. Theoretically, IC methods achieve their tailored solutions by exploring numerous pass sequences for a specific program and evaluating their impact, thus better adapting to program diversity [3, 6–8, 16, 26, 35]. However, IC methods suffer from practical drawbacks. The primary impediment is the prohibitive search cost; the combinatorial space of pass sequences grows exponentially with the number of available passes. Performing an exhaustive or even heuristic iterative search and recompilation for each program consumes substantial time and computational resources, rendering IC impractical for development workflows requiring rapid feedback or for large-scale software deployment.

Artificial intelligence (AI) methods, which encompass various machine learning (ML) techniques, including recent large language models (LLMs), offer a promising path to low-overhead compilation decisions by predicting effective optimization sequences; however, their generalization capability, especially for traditional models, remains a significant hurdle. More recently, AI has been widely applied to predict effective pass sequences or guide compilation decisions [10, 13, 14, 18, 19, 24, 25, 28, 30, 31, 33]. Although AI-based approaches often boast low inference-time overhead, a core challenge for many of these models, particularly traditional ML models, lies in their generalization capability. Specifically, these models, fine-tuned to find optimal sequences for individual programs or a narrow set of similar programs, tend to overfit the specific structural features of their training data. This overfitting leads to poor performance when applied to new unseen programs with different characteristics.

To address these multifaceted challenges, this paper introduces **GRACE (Globally-seeded Representation-Aware Cluster-specific Evolution)**, a framework for compiler pass auto-tuning. GRACE effectively reduces search space and provides high-quality initial pass sequences for subsequent stages by leveraging pass synergies and a weighted scoring mechanism to derive initial high-quality candidate sequences and their constituent pass pool. GRACE employs a contrastive learning approach to design program embeddings that are more effective for similarity-aware clustering than prior methods like PairVec [28]. These embeddings are then used to cluster training programs. Subsequently, an evolutionary search is performed within each of the resulting k clusters to identify a high performing pass sequence for each cluster, forming a coreset of k sequences. Finally, GRACE includes a test-time deployment strategy with sequence refinement options to ensure robust application.

Our extensive evaluations on seven diverse datasets demonstrate that GRACE achieves **state-of-the-art** performance in LLVM IR instruction count reduction. Specifically, compared to the standard LLVM `opt -Oz` optimization level, our method reduces the instruction count by an average of **10.09%** on LLVM 10.0.0 and **10.19%** on LLVM 18.1.6, while incurring an average tuning time of less than **1s** per program, showcasing its superior effectiveness over existing optimization techniques.

The key contributions of this work are:

- We propose a novel method for the reduction and initialization of search spaces based on synergy, which leverages pass synergy and a weighted scoring mechanism to derive high-quality initial candidate pass sequences and a pass pool, effectively guiding subsequent optimization.
- We introduce a contrastive learning approach for embedding-based program clustering. We demonstrate empirically that our method generates higher-quality program embeddings for this task compared to directly comparable prior work, leading to more effective specialization.
- We provide a specialized coreset of k high-performing pass sequences, designed for effective application to unseen programs. Each sequence is tailored to distinct program groups and is derived from a targeted evolutionary search within program clusters.
- We provide a practical test-time deployment, featuring coreset selection, sequence refinement options, ensuring both applicability and reliable performance.

The remainder of this paper is organized as follows. Section 2 reviews reviews the evolution of compiler pass auto-tuning and compares our method with recent state-of-the-art approaches. Section 3 details the proposed GRACE framework, including its four main stages: global candidate generation, contrastive learning for program embeddings, cluster-specific coreset evolution, and test-time application with refinement. Section 4 describes our experimental methodology, benchmarks, evaluation metrics, and baseline comparisons. Section 5 presents and analyzes the experimental results. Section 6 discusses the limitations of our work and outlines promising directions for future research. Finally, Section 7 concludes the paper and discusses several potential avenues for future research.

2 Related Work

The automatic tuning of compiler optimization passes, addressing the phase-ordering and pass selection problem, has seen considerable research. This section reviews key developments, from iterative compilation to ML, the emerging use of LLMs, and finally, discusses specific recent state-of-the-art methods for LLVM IR instruction count reduction in comparison to our proposed GRACE framework.

2.1 Approaches to Compiler Optimization Tuning

Iterative Compilation. Early pass auto-tuning efforts centered on iterative compilation, where numerous pass sequences are explored for a specific program, compiled, and evaluated to find an optimal configuration. This per-program approach, while capable of achieving good results for individual programs, suffers from prohibitive time costs, limiting its scalability for general use. Various search strategies and frameworks have been developed within this iterative paradigm, showcasing the diversity of exploration techniques, as exemplified in works such as [3, 6–8, 16, 35].

ML-based Approaches. To overcome IC's overhead, ML methods aim to learn predictive models for optimization [14, 19, 24, 29–31, 33]. These models, trained in program features and effective sequences (often derived from IC), can infer potentially good strategies for unseen programs much faster than an exhaustive search. A variety of ML paradigms have been explored. For example,

supervised learning approaches might directly predict entire sequences of passes or make decisions at various compilation stages [25]. Among these, Reinforcement Learning (RL) has emerged as a particularly prominent technique, framing the phase-ordering problem as a sequential decision-making process. Notable RL-based works include Autophase [18], which learns a policy to select passes dynamically, and CompilerGym [12], which provides a standardized RL environment for compiler optimization research.

LLMs in Compilation. Recently, LLMs have emerged as a new frontier [13, 22]. Using their ability to process and generate text-like data, researchers are exploring LLMs to understand code and suggest optimizations [10, 11, 21]. This typically involves Supervised Fine-Tuning (SFT) of pre-trained models on code optimization datasets or training LLMs from scratch on vast code corpora. Although this area is nascent, LLMs hold the potential to learn complex program representations and optimization patterns directly, a significant advantage over bypassing traditional feature engineering, but face challenges in data requirements and interpretability.

2.2 GRACE vs. State-of-the-Art Methods

Although the aforementioned areas represent broad trends, recent specific advances in reducing the number of LLVM IR instruction counts, such as Coreset-NVP [25] and CFSAT [28], provide important context for our work. **GRACE is engineered to leverage their foundational insights while directly addressing certain inherent limitations.**

Coreset-NVP [25] identifies a coreset of pass sequences, but its initial sequences, often found via random search per program, may lack optimal quality and robustness. Furthermore, Coreset-NVP's coreset may lack strong generalization because it is selected by re-evaluating per-program optima (found via random search) for average training-set utility, not by directly seeking sequences robust across varied program types. In contrast, GRACE first establishes a stronger foundation by employing pass synergy analysis and a weighted scoring mechanism (detailed in Section 3) to generate high-quality initial candidate sequences and a limited pass pool. For enhanced generalization, GRACE then evolves its coreset sequences within program clusters (formed using learned embeddings) by optimizing the fitness score, explicitly targeting robust performance across diverse programs within each cluster.

CFSAT [28] leverages pass synergy and clustering; however, its methodology presents two key limitations that GRACE is designed to address. First, its clustering relies on embeddings derived from "reaction matching," an approach that primarily captures program responses to a limited set of predefined passes. This may not fully capture the more nuanced, structural similarities required for effective optimization. GRACE overcomes this by using a novel contrastive learning approach with extensive pass sequence-based data augmentation. This encourages the model to learn more semantically meaningful program embeddings, which, as our experiments show, results in superior clustering quality. Second, CFSAT maintains a persistent focus on known pass synergies throughout its process, which can prematurely restrict the search space. GRACE adopts a more flexible strategy: while pass synergy is instrumental in bootstrapping the initial candidate generation, subsequent stages like coreset evolution are explicitly permitted to explore beyond these initial paths. This design allows GRACE to discover novel and powerful pass combinations, thus preserving more optimization avenues and striking a better balance between exploitation and exploration.

3 The GRACE Framework

This section details the proposed GRACE framework, a multistage approach designed for compiler pass auto-tuning, with a focus on LLVM IR instruction count optimization. An overview of GRACE's architecture is shown in Figure 1. The framework comprises four main stages: (1) Global Candidate

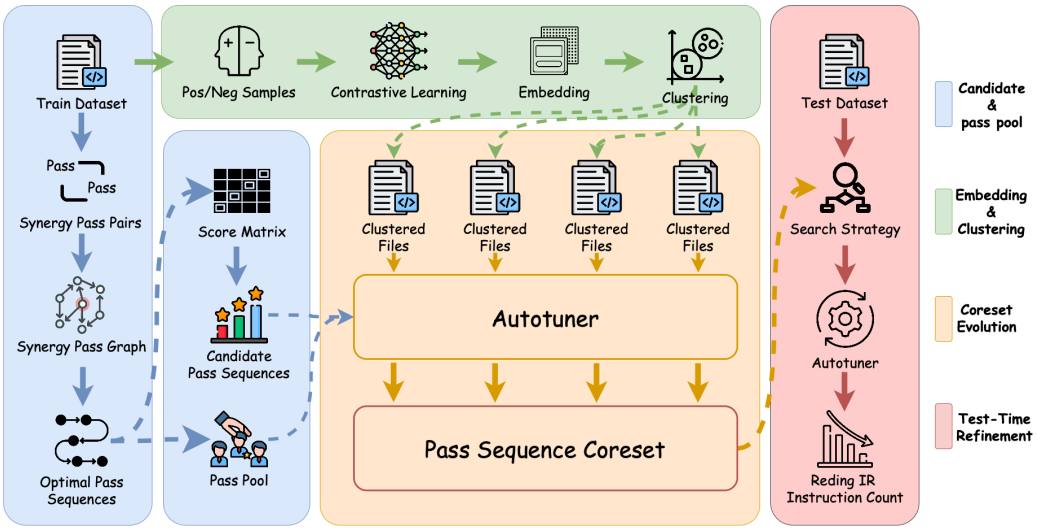


Fig. 1. The GRACE framework.

and Pass Pool Construction; (2) Contrastive Learning for Program Embeddings and Clustering; (3) Cluster-Specific Coreset Evolution; and (4) Test-Time Application with Refinement. Each stage is elaborated below.

3.1 Global Candidate and Pass Pool Construction

The initial stage of GRACE aims to reduce the vast search space of pass sequences and identify a set of generally effective candidate sequences alongside a constrained pool of useful passes. This is achieved through a multistep process performed in the training set of programs $\mathcal{P}_{train}$.

Pass Synergy Identification and Graph Construction. Inspired by the concept of pass synergy highlighted in CFSAT [28], we propose a method for identifying and leveraging such relationships in this subsection. For each training program $P_i \in \mathcal{P}train$, we begin by identifying its set of synergistic pass pairs S_i. As described in Algorithm 1, this involves computing a baseline IR instruction count V_P for P_i. Then, for each compiler pass $B \in \mathcal{O}all$, we evaluate its effect on P_i. If applying B alone reduces the instruction count, we further explore its synergy with every other pass $A \in \mathcal{O}all$ by applying the sequence $\langle A, B\rangle$ to the original P_i. If this combination yields a further reduction in instruction count compared to B alone ($V_{P_{AB}} < V_{P_B}$), the pair (A, B) is added to S_i. After processing all programs, we aggregate the individual synergy sets S_i into a global collection $\mathcal{S}global$ and build a comprehensive pass synergy graph Gco, which captures the general synergistic relationships between compiler passes observed across the training corpus.

High-Performing Sequence Search, Evaluation, and Pass Pool Formulation. Leveraging the pass synergy graph G_{co} - where the nodes represent individual compiler passes and the directed edges indicate the observed synergistic relationships - a search algorithm is used for each training program P_i to identify a high-performing sequence of passes $Seq_{hp,i}$ that minimizes the count of IR instruction. This process yields a set of unique high-performance sequences, denoted as Seq_{HP}. The sequence $Seq_u \in Seq_{HP}$ is then evaluated across all training programs $P_j \in \mathcal{P}train$, with a performance score computed for each pair (Seq_u, P_j) based on a weighted combination of distributional metrics using `DISTRIBUTION_SCORE_WEIGHTS` = {`'Avg'`: 0.75, `'Std'`: 0.1, `'NegRate'`: 0.15}. Specifically, *Avg* denotes the average improvement in instruction count, *Std* reflects performance

Algorithm 1 Synergy Set Identification and Global Synergy Graph Construction

Input: Training set of programs $\mathcal{P}_{train} = \{P_1, ..., P_N\}$
Input: Set of all available compiler passes O_{all}
Output: Global Pass synergy Graph G_{co}

function IdentifySynergyPairs(P, O)
    $\mathcal{S}_P \leftarrow \emptyset$
    $V_P \leftarrow$ GetIRInstructionCount(P)
    for all optimization pass $B \in O$ **do**
        $P_B \leftarrow$ ApplyPass(P, B)
        $V_{P_B} \leftarrow$ GetIRInstructionCount(P_B)
        if $V_{P_B} < V_P$ **then**
            for all optimization pass $A \in O$ **do**
                $P_{AB} \leftarrow$ ApplyPassSequence($P, \langle A, B \rangle$)
                $V_{P_{AB}} \leftarrow$ GetIRInstructionCount(P_{AB})
                if $V_{P_{AB}} < V_{P_B}$ **then**
                    $\mathcal{S}_P \leftarrow \mathcal{S}_P \cup \{(A, B)\}$
                end if
            end for
        end if
    end for
    return $\mathcal{S}_P$
end function

Let $\mathcal{S}_{global} = \emptyset$
for all program $P_i \in \mathcal{P}_{train}$ **do**
    $S_i \leftarrow$ IdentifySynergyPairs(P_i, O_{all})
    $\mathcal{S}_{global} \leftarrow \mathcal{S}_{global} \cup S_i$
end for
$G_{co} \leftarrow$ BuildGlobalSynergyGraph($\mathcal{S}_{global}$)
return G_{co}

stability, and *NegRate* measures the proportion of programs where the sequence performs poorly relative to `opt -Oz`. Both *Std* and *NegRate* serve as penalty factors. An overall quality score for each Seq_u is obtained by aggregating its performance across all programs, and the sequences in Seq_{HP} are ranked accordingly. To better understand the quality distribution among all Seq_u, we visualize the distribution of their weighted scores using a histogram (see Fig. 2), which highlights the variance in the effectiveness of the sequence across the search space. From the ranked list, the top-k_{top} sequences are selected to form C_{seq} as candidate pass sequences, and all unique compiler passes within C_{seq} are aggregated to construct a constrained *pass pool* P_{pool}. This P_{pool} delimits the search space for subsequent evolutionary stages.

Thus, this stage provides a reduced set of promising sequences (C_{seq}) and a relevant subset of passes (P_{pool}) that form the foundation for the more specialized optimization steps that follow.

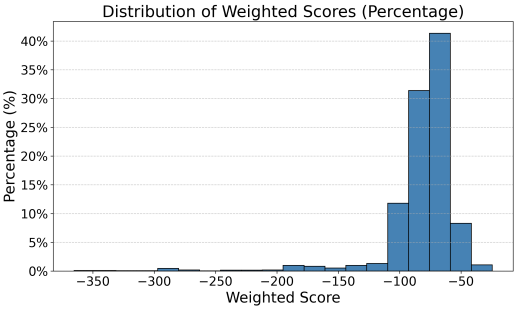


Fig. 2. Distribution of weighted scores for all unique high-performing sequences. The spread indicates varying levels of optimization quality and reliability across candidate sequences, highlighting the diversity of promising optimization strategies uncovered by GRACE.

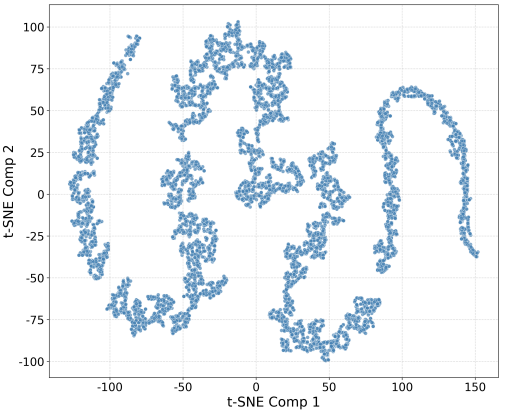


Fig. 3. t-SNE visualization of the learned program embeddings e_k.

3.2 Contrastive Learning for Program Embeddings and Clustering

The second stage focuses on learning meaningful representations (embeddings) for programs that capture their similarities with respect to compiler optimization behavior and then using these embeddings to cluster the programs.

Program Representation and Positive Sample Generation for Contrastive Learning. The foundation of our contrastive learning model is a meaningful representation of each program P_i in the training set. We begin by extracting its Autophase features [18], a set of static program characteristics known to be relevant for compiler optimization. These raw features serve as an initial input. To generate positive samples for contrastive learning, we employ a data augmentation strategy rooted in the understanding that a program, fundamentally a set of instructions, can be viewed as a new, but semantically related entity after undergoing compiler transformations [26]. Specifically, for a given program P_i (the "anchor"), we create one or more variants, P_i', by applying randomly generated compiler pass sequences to P_i. Each such variant P_i', now differing at the instruction level, is considered a new program instance but semantically close to the original P_i. The Autophase features of these variants P_i' are then extracted. The pair of feature vectors corresponding to (P_i, P_i') forms a positive sample for contrastive learning. Within a training batch, all other distinct programs P_j (where $j \neq i$) are treated as negative samples relative to P_i.

Contrastive Learning for Program Embeddings. With positive (P_i, P_i') and negative (P_i, P_j) sample relationships established, we employ a contrastive learning method. The core idea is to train a model, typically a neural network, to learn an embedding function $f_{embed}(\cdot)$. This function projects the raw Autophase features of the programs into an embedding space where positive pairs (e.g., P_i and its augmented version P_i') are mapped close together, while negative pairs (e.g., P_i and an unrelated program P_j) are pushed far apart. Our specific model architecture to achieve this consists of a neural network encoder $f_{enc}(\cdot)$, which processes the input Autophase features to produce an intermediate embedding $h_i = f_{enc}(x_i)$. This is followed by a projection head, $f_{proj}(\cdot)$, which maps h_i to a vector $z_i = f_{proj}(h_i)$ that is used during the contrastive loss calculation.

For training, we utilize a *Distance Contrastive Loss*. Given a batch of N anchor programs and their N positive augmentations, yielding $2N$ projected vectors $\{z_k\}_{k=1}^{2N}$, we compute pairwise similarities $S_{ab} = s(z_a, z_b)$ using a Gaussian kernel applied to their squared Euclidean distances, $s(z_a, z_b) = \exp(-||z_a - z_b||_2^2/\tau)$, where τ is a temperature hyperparameter. This forms a $2N \times 2N$

similarity matrix S. The contrastive loss for a sample z_a and its corresponding positive pair $z_{pos(a)}$ is then calculated using a cross-entropy objective over the similarities, after masking self-similarity terms:

$$\mathcal{L}_a = -\log \frac{\exp(S_{a,pos(a)})}{\sum_{k=1}^{2N} \exp(S_{ak})} \tag{1}$$

The total loss for the batch is the average of $\mathcal{L}_a$ over all $2N$ samples. The encoder $f_{enc}(\cdot)$ without the projection head then serves as the program embedding function $f_{embed}(\cdot)$, yielding the final program embedding $e_k = h_k = f_{enc}(x_k)$ used for downstream tasks such as clustering. The quality and separability of these learned embeddings e_k can be qualitatively assessed by visualizing their t-SNE projection in a 2-dimensional space, as shown in Figure 3.

K-Means Clustering on Program Embeddings. Once the contrastive learning process yields the final program embeddings $\{e_i\}_{i=1}^{N_{train}}$ for all N_{train} programs in the training set (where $e_i = f_{embed}(P_i)$), these embeddings are used to group the programs based on their learned similarities. We employ the k-means clustering algorithm [2, 34] for this purpose. The k-means algorithm aims to partition the N_{train} embedding vectors into k disjoint clusters, $C = \{C_1, C_2, ..., C_k\}$. Each cluster C_j is characterized by its centroid μ_j, which is the mean of all embedding vectors assigned to that cluster.

The objective function for k-means, which the algorithm iteratively minimizes, can be expressed as:

$$\underset{C}{\operatorname{argmin}} \sum_{j=1}^{k} \sum_{e_i \in C_j} ||e_i - \mu_j||_2^2 \tag{2}$$

where $||e_i - \mu_j||_2^2$ is the squared Euclidean distance between an embedding vector e_i and the centroid μ_j of the cluster C_j to which e_i is assigned. This partition results in k distinct groups of programs, where programs within the same cluster are hypothesized to benefit from similar optimization strategies due to their proximity to the learned embedding space.

A key design choice in GRACE is the use of program embeddings. These embeddings are employed as an offline tool to structure the search space by grouping similar programs, which facilitates the construction of coreset sequences that can generalize across programs. However, they are not used for online prediction at test time (e.g., via nearest-neighbor matching), since predictive models of this type may be less reliable when applied to out-of-distribution programs. Instead, GRACE evaluates all k coreset sequences empirically at test time, providing a straightforward method to select the best sequence within the coreset for a given program. This approach introduces a small, fixed inference cost while maintaining consistency in sequence selection.

3.3 Cluster-Specific Coreset Evolution

Once programs are grouped into k distinct clusters based on their learned embeddings, this stage focuses on evolving a coreset consisting of k highly effective pass sequences. The objective is to generate $\{Seq_{coreset,1}, ..., Seq_{coreset,k}\}$, where each $Seq_{coreset,j}$ is specialized for the programs within its corresponding $Cluster_j$.

This specialization is achieved by running a separate genetic algorithm (GA) for each of the k clusters. A crucial aspect of this cluster-specific GA is its strategic exploitation of global artifacts derived from the initial global candidate generation stage (Section 3.1). Specifically, the initial population for GA optimizing a given $Cluster_j$ is significantly informed by these global insights: approximately half of its individuals are directly seeded from the top k candidate sequences (C_{seq}). Furthermore, the evolutionary process is guided by constraints in genetic operations, particularly mutation. When new genetic material is introduced through mutation, the selection of passes is

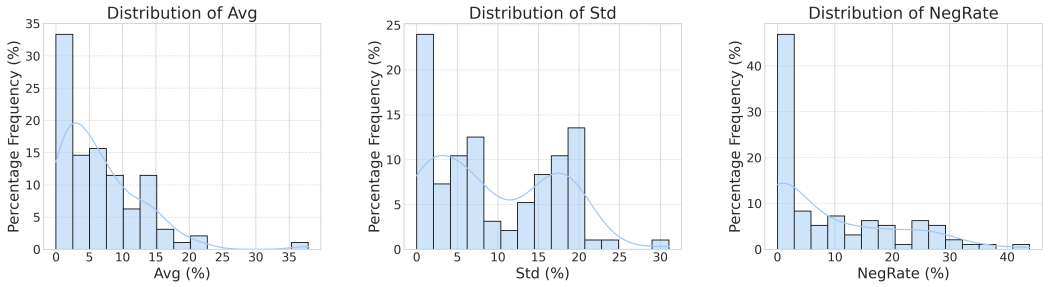


Fig. 4. Distributions of Performance Characteristics for the k Coreset Sequences. For each of the k coreset sequences (one per cluster), we evaluate its performance on all programs within its respective cluster to derive three metrics: average LLVM IR instruction count reduction (Avg), standard deviation of these reductions (Std), and negative impact rate (NegRate) compared to `opt -Oz`. The histograms and smoothed curves presented here show the distributions of these k individual Avg, Std, and NegRate values, offering insights into the overall performance profile of the generated coreset.

restricted to those within the pass pool (P_{pool}). This ensures that the search remains focused on passes with demonstrated general utility or synergistic potential. The fitness of any sequence of passes during GA for $Cluster_j$ is determined by applying the sequence of passes to all programs within $Cluster_j$ and calculating its aggregate performance. In this work, the main objective of fitness is to maximize the weighted score across all programs in $Cluster_j$. The GA then evolves its population to find the sequence $Seq_{coreset,j}$ that produces the best fitness score for its designated $Cluster_j$.

The resulting k coreset sequences are designed to be both high-performing and robust for their respective program categories. To assess the overall quality and characteristics of these evolved coreset sequences, we analyze key performance metrics. Specifically, for each of the k coreset sequences, we calculate its average instruction count reduction (Avg), the standard deviation (Std) of these reductions, and the negative impact rate (NegRate) based on its application to all programs within its designated cluster. Figure 4 then illustrates the distributions of these calculated Avg, Std, and NegRate values k. For instance, a desirable outcome depicted in such a figure would show a strong positive skew for Avg improvements, indicating that significant instruction count reductions are commonly achieved by the coreset sequences. A low Std across the clusters would suggest consistent performance of the coreset sequences within their intended groups, while a low NegRate would confirm that these specialized sequences rarely harm performance. These collective distributions provide confidence in the general efficacy and reliability of the coreset generated by GRACE.

3.4 Test-Time Application with Refinement

During the testing phase, when a new, unseen program P_{test} is encountered, GRACE applies its coreset and refinement strategies to determine an effective optimization sequence. Initially, P_{test} is compiled and evaluated using each of the k coreset sequences $Seq_{coreset,j}$ that were evolved in Section 3.3. The sequence of this coreset that produces the best performance in P_{test} is selected as the initial best candidate, denoted $Seq_{candidate}$.

This $Seq_{candidate}$ can then be subjected to one or more optional enhanced search or refinement methods, designed to potentially improve its performance further for P_{test}. Three such refinement techniques employed by GRACE are:

Derivative Search. This technique investigates the potential of subsequences derived from the current candidate sequence, $Seq_{candidate}$. Given a sequence of passes $p_1 \cdot p_2 \cdot \ldots \cdot p_L$, its prefixes—$p_1$, $p_1 \cdot p_2$, ..., $p_1 \cdot \ldots \cdot p_{L-1}$—are compiled and evaluated on P_{test}. If any prefix achieves a performance metric better than the full sequence, it replaces the original as the new $Seq_{candidate}$. Otherwise, the original sequence is retained. This approach aims to determine whether a shorter, potentially less complex sequence yields better results for the given program and helps eliminate unnecessary or harmful passes near the end.

Localized GA Refinement. As an alternative, a highly localized and short-lived GA can be applied to refine $Seq_{candidate}$. The set of passes in $Seq_{candidate}$ is extracted to form a small, program-specific pass pool. The GA then searches for a new sequence—using only passes from this localized pool—that exceeds the performance of the current $Seq_{candidate}$ in P_{test}. If such a sequence is found, it becomes the new $Seq_{candidate}$; otherwise, the original is retained. This refinement strategy enables subtle improvements by reordering passes or introducing minor substitutions without incurring a high search cost.

Baseline Comparison (Oz Fallback). This involves comparing $Seq_{candidate}$ against the standard optimization level provided by LLVM's `opt -Oz`. If $Seq_{candidate}$ delivers superior performance on P_{test}, it is retained. Otherwise, the pass sequence corresponding to `opt -Oz` replaces it. This comparison ensures that the search process does not regress below a well-established baseline and offers a safety net based on proven compiler heuristics.

In summary, GRACE employs three refinement techniques, each serving a specific purpose. First, Derivative Search addresses the observation that a prefix of a longer optimization sequence can sometimes be more effective, which helps to remove unnecessary or potentially detrimental passes. Second, Localized GA Refinement performs a low-cost, focused search for minor improvements, such as reordering or substituting passes, without the computational overhead of a global search. Third, Baseline Comparison (Oz Fallback) provides a conservative safeguard by comparing candidate sequences against a well-established baseline, ensuring that performance does not fall below a reliable reference point. **These refinement steps are optional and can be applied at the user's discretion.**

4 Experimental Setup

This section details the experimental methodology used to evaluate the GRACE framework. We describe the compiler environment, the datasets used for training and testing, and the baseline methods against which GRACE performance is compared.

4.1 Compiler Environment

All experiments were conducted within the **LLVM compiler infrastructure**, a standard toolchain for compiler research. To ensure both comparability with prior work and relevance to modern compilers, we evaluated GRACE on two LLVM versions:

- **LLVM 10.0.0:** Enables fair comparison with baselines such as CFSAT [28] and Coreset-NVP [25], originally evaluated on this version or similar ones.
- **LLVM 18.1.6:** Provides a test on a modern LLVM release with updated passes and heuristics, offering evidence of forward-compatibility.

Results across both versions suggest that GRACE's core techniques are not overly dependent on a specific pass set, though broader validation on other compilers remains future work. All experiments were run on an AMD EPYC 7763 64-Core system, with GRACE interacting with LLVM's `opt` tool.

4.2 Datasets

We use a comprehensive collection of benchmarks for GRACE training and evaluation, sourced primarily from the CompilerGym environment [12]. The complete composition of the data set, including the division of programs into training and testing sets, is detailed in Table 1. This division is crucial for evaluating the generalizability of GRACE. The training set, which totals **19,603** programs, is used for pass synergy analysis, contrastive learning of embeddings, and coreset evolution. The test set, comprising **335** unseen programs in seven datasets, is strictly used for the final performance evaluation.

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 [27]	7,000	0
	tensorflow-v0 [1]	415	90
Curated	cbench-v1 [15]	0	11
	mibench-v1 [17]	0	40
	chstone-v0 [20]	0	12
	npb-v0 [4]	0	121
Total	–	**19,603**	**335**

4.3 Baseline Methods

To rigorously evaluate GRACE, we compare its performance against a comprehensive set of established baseline methods. These baselines can be categorized into two groups: methods for auto-tuning pass sequences and methods for program representation. In our evaluation, GRACE uses a coreset consisting of 100 representative pass sequences.

(1) **Auto-tuning Method Baselines:** These represent state-of-the-art or widely recognized techniques for finding effective pass sequences. Our comparison includes:
- Iterative Compilation and Search-Based Methods: RIO [8], TPE [6], GA[16], CompTuner [35], OpenTuner [3], and BOCA [7].
- Machine Learning-Based Methods: CFSAT [28], Coreset-NVP [25], and Autophase [18] (RL-PPO-LSTM / non-LSTM variant).

(2) **Program Representation Method Baselines:** Since GRACE employs a learned program representation, we also compare the quality of its embeddings (in terms of clustering performance) against other program representation techniques:
- PairVec [28]: This refers to the embedding methodology used within the CFSAT [28] framework for clustering programs.
- Inst2Vec [5]: A technique that learns embeddings for individual instructions.
- IR2Vec [32]: A method for generating embeddings directly from LLVM IR.
- Autophase Features [18]: The raw Autophase features themselves, used as input to GRACE's contrastive learning, serving as a baseline representation.

4.4 Experimental Protocol

To ensure a fair evaluation, we define two key aspects of our experimental protocol. First, regarding the **Termination Criteria** for search-based methods, all iterative compilation baselines (e.g., RIO, TPE, GA) were allowed up to 200 seconds of search per program, which was sufficient for their performance to stabilize. GRACE's evolutionary stage runs for a fixed number of generations (50 in our setup), balancing optimization quality and computational cost.

Second, concerning **Pass Repetition**, compiler passes are allowed to appear multiple times within an optimization sequence. This reflects the nature of the phase-ordering problem, where repeated passes can provide additional optimization opportunities. GRACE's evolutionary algorithm represents sequences as chromosomes, optimizing both pass selection and order, including repetitions. In contrast, several baseline methods restrict sequences to non-repetitive passes, which may limit their ability to fully exploit the optimization space.

5 Results and Analysis

5.1 Overall Performance Comparison

To evaluate the effectiveness of GRACE, we conducted a comprehensive comparison against several state-of-the-art auto-tuning baselines. The primary metric for this comparison is the average *OverOz* score, calculated for each program as $OverOz = (N_{Oz} - N_{Ox})/N_{Oz} \times 100\%$, where N_{Oz} is the LLVM IR instruction count after applying `opt -Oz`, and N_{Ox} is the instruction count after applying the sequence found by the method x. A higher *OverOz* score indicates a greater reduction in instruction count compared to `opt -Oz`. We report the average *OverOz* score for each method in each of the seven test datasets, as well as the overall average across all datasets and the average time taken per program for tuning/inference.

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	cbench	chstone	mibench	npb	opencv	tensorflow	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
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
GRACE (Ours)	**5.15**	**6.39**	10.63	**7.04**	**28.90**	5.67	**6.89**	**10.09**	**<1**

The results presented in Table 2 clearly demonstrate the superior performance of our proposed GRACE framework in reducing the LLVM IR instruction count compared to `opt -Oz` and other state-of-the-art baselines. GRACE achieves the highest average OverOz score of **10.17%** across all seven test datasets, outperforming all other methods. In particular, GRACE shows strong performance on individual datasets, securing the best OverOz scores on `blas` (5.15%), `cbench` (6.39%), `mibench` (7.04%), `npb` (28.90%), and `tensorflow` (6.89%). Although CFSAT demonstrates competitive performance on specific datasets, such as `chstone` (11.90%) and `opencv` (5.90%), slightly outperforming GRACE (10.80% and 5.68%, respectively), this difference can be attributed to the evaluation methodology. CFSAT performs a search over a large and unconstrained optimization

space during the final testing phase, whereas GRACE explores a smaller, fixed optimization space defined by its coreset. This larger search potential enables CFSAT to identify highly specialized sequences for particular programs, explaining its advantage on these datasets. However, GRACE maintains a consistent overall performance, indicating robust generalization without relying on extensive test-time search. In addition to performance, GRACE offers practical benefits for real-world deployment: (1) **Interpretability**: GRACE generates a human-readable coreset of pass sequences, allowing compiler engineers to inspect and understand the applied optimization strategies, in contrast to CFSAT's reliance on a black-box attention model; (2) **Deployment simplicity**: the coreset is a static artifact that can be integrated into build systems or compiler toolchains without requiring an online inference model, simplifying deployment and maintenance; (3) **Robustness**: GRACE evaluates a curated set of elite sequences at test time and employs a conservative `-Oz` fallback, providing stronger safety guarantees against out-of-distribution programs than a purely predictive model. Overall, while CFSAT exhibits strong search capabilities, GRACE provides a more balanced framework in terms of average performance, deployability, and robustness.

Beyond sheer effectiveness, GRACE also exhibits highly competitive efficiency. With an average tuning time of approximately **< 1s** per program, GRACE is significantly faster than all iterative compilation and search-based methods such as Opentuner (200 s), GA (593 s), TPE (905 s), RIO (200 s), CompTuner (10800 s) and BOCA (3310 s). Its speed is comparable to, and only marginally slower than, the inference time of the Autophase (PPO-LSTM/no-LSTM) model (3s/2s per program, each episode involves 30 steps for a fair comparison of interaction complexity). This balance of high performance and practical efficiency is a key advantage of GRACE.

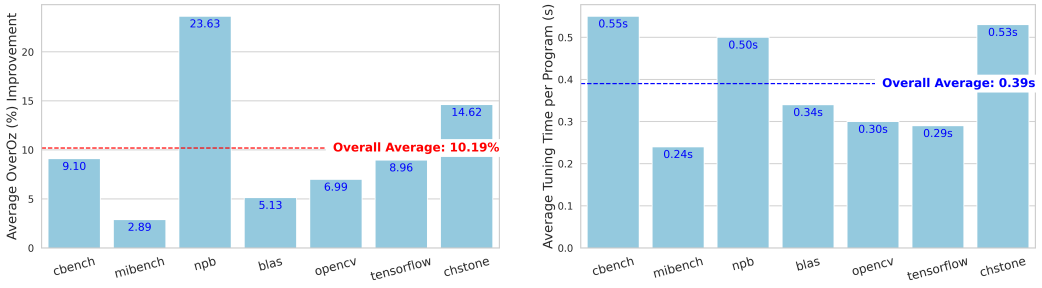
The iterative compilation methods (Opentuner, GA, TPE, RIO, CompTuner, BOCA), while capable of exploring vast search spaces, do not consistently achieve the best results in our *OverOz* comparison. A primary contributing factor to this could be their typical focus on pass selection rather than explicitly and intricately optimizing the order of the selected passes. The phase-ordering aspect is critical, as the interactions between passes are highly dependent on their sequence of application, a nuance that GRACE's hierarchical and evolutionary approach is designed to capture. Observing the Autophase results, the significant performance difference between Autophase (PPO-LSTM) (1.24% Avg. OverOz) and Autophase (PPO-noLSTM) (-50.35% Avg. OverOz) highlights the critical importance of considering the temporal dimension or sequence history in pass-selection problems. The LSTM component, by design, captures sequential dependencies, which appear vital for effective tuning. The relatively modest performance of even the LSTM-based Autophase, despite being trained for a substantial 3,000,000 episodes, might suggest potential limitations in its state representation or even more extensive training.

5.2 Generalizability on LLVM 18.1.6

To address the critical concern of generalizability and demonstrate GRACE's forward-compatibility, we conducted a supplementary evaluation using a modern compiler, LLVM version 18.1.6. In this study, we re-ran all experiments on the same seven benchmark suites with the LLVM 18.1.6 toolchain, ensuring that the evaluation fully reflects the behavior of a contemporary compiler with its updated optimization passes and heuristics.

The performance results, presented in Figure 5a, confirm the robustness of our approach. GRACE achieves an overall average OverOz improvement of 10.19% across all seven datasets. Notably, it delivers strong performance on diverse benchmarks, such as achieving a 23.63% reduction on NPB and a 14.62% reduction on CHStone. These results are highly competitive with those obtained on LLVM 10, indicating that GRACE's methodology of synergy-based candidate generation and cluster-specific evolution is not tied to a specific compiler version but captures fundamental properties of the optimization problem.

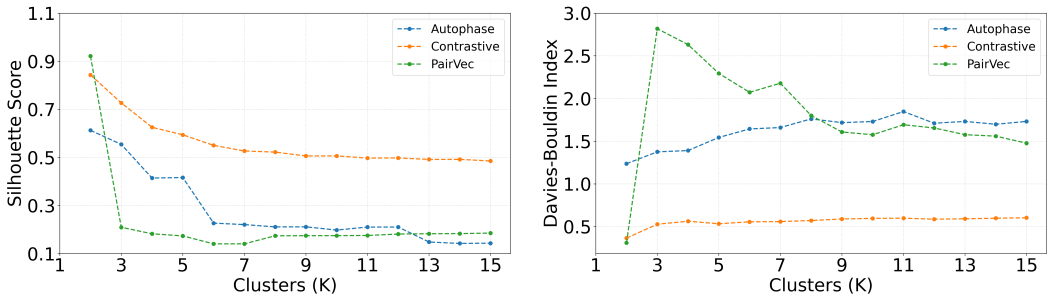
Furthermore, GRACE maintains its exceptional efficiency on the newer LLVM version, as detailed in Figure 5b. The average tuning time per program is a mere 0.39 seconds. This high throughput is critical for practical applications, confirming that GRACE can be integrated into development workflows without introducing significant compilation overhead. The combination of strong optimization and low overhead underscores the practical value of our framework in modern software development.



(a) Average OverOz improvement on LLVM 18.1.6. (b) Average tuning time per program on LLVM 18.1.6.

Fig. 5. GRACE's performance and efficiency on LLVM 18.1.6 across seven benchmark suites. (a) Instruction count reduction relative to `opt -Oz`. (b) Average per-program tuning time, demonstrating low overhead.

5.3 Analysis of Program Embedding Quality and Its Impact



(a) Silhouette Score vs. K (b) Davies-Bouldin Index vs. K

Fig. 6. Comparison of clustering quality for Autophase raw features, GRACE's Contrastive and PairVec using Silhouette Score and DBI across varying numbers of clusters (K).

A core motivation for GRACE is to improve upon the program representation methods used in prior auto-tuning frameworks. Specifically, we hypothesize that a more effective program embedding, tailored for optimization similarity, can lead to superior specialization compared to existing approaches like PairVec. To validate this, this section presents a two-stage analysis. First, we directly compare the clustering quality of our novel contrastive learned embeddings against PairVec. Second, we conduct an ablation study to empirically measure how the choice between these two embedding methods translates to GRACE's end-to-end performance.

Clustering Quality Evaluation. To quantify the effectiveness of our embedding approach, we compare it against **PairVec** [28], a state-of-the-art method also designed for program clustering in compiler auto-tuning, and the raw **Autophase features** [18] which serve as the input to our model. We perform k-means clustering for a range of cluster numbers (K from 2 to 15) and evaluate the results using two standard internal validation metrics: the **Silhouette Score** (higher is better) and the **Davies-Bouldin Index (DBI)** (lower is better).

The results are presented in Figure 6. The empirical data clearly demonstrates the superiority of our contrastive learning approach. As shown in Figure 6(a), our embeddings consistently achieve the highest Silhouette Scores across nearly the entire range of K values, suggesting that the clusters formed are dense and well-separated. For example, at $K = 4$, our method achieves a score of [e.g., 0.75], significantly outperforming PairVec [e.g., 0.55]. Similarly, Figure 6(b) shows that our method consistently yields the lowest (best) DBI scores, indicating more distinct and well-defined clusters. This strong performance on offline metrics validates that our method is highly effective at learning program representations that capture meaningful similarities for optimization partitioning.

Impact on End-to-End Auto-Tuning Performance. Having established the superior clustering quality of our embeddings, we now investigate the crucial question: does this translate to better final optimization results? To answer this, we conducted an ablation study where we replaced our contrastive learning module within the GRACE framework with PairVec, keeping all other components (synergy analysis, evolutionary algorithm, etc.) identical. We then evaluated the end-to-end performance of both configurations on the entire test set.

The results of this study provide conclusive evidence linking representation quality to final performance. When the GRACE framework is configured with our contrastive embeddings, it achieves its state-of-the-art average OverOz improvement of **10.09%** on LLVM 10.0.0. However, when the framework is instead guided by the lower-quality PairVec embeddings, the end-to-end performance drops to **9.25%**. This demonstrates a clear and direct correlation: the higher-quality partitioning enabled by our novel embedding approach allows the subsequent evolutionary stage to discover a more effective and specialized coreset, ultimately leading to superior generalization on unseen programs.

This two-part analysis validates our central claim: the novel contrastive learning approach introduced in this work generates more effective program representations for compiler auto-tuning than those used in prior state-of-the-art frameworks like CFSAT. The superior clustering quality and the resulting end-to-end performance gain highlight the significant contribution of our representation learning module to the overall success of the GRACE framework.

5.4 Ablation Study: Impact of Initial Candidates and Pass Pool

We performed an ablation study to evaluate Section 3.1 components: the high-quality initial candidate sequences (C_{seq}) and the constrained pass pool (P_{pool}). Four configurations were tested using a GA (10 generations, 25 individuals) on each test dataset, optimizing for average OverOz: 1) RandInit-AllPasses (random sequences, all passes for mutation); 2) CandInit-AllPasses (GRACE's C_{seq} initialization, all passes); 3) RandInit-SpecPool (random sequences, GRACE's P_{pool} for mutation); 4) CandInit-SpecPool (GRACE's C_{seq} and P_{pool}). Table 3 summarizes the converged average OverOz (%) and convergence generation.

The results demonstrate clear benefits from the Section 3.1 design. Initialization with GRACE candidate sequences (*CandInit*) consistently accelerates convergence and improves OverOz scores compared to random initialization (*RandInit*), as seen, for example, on `blas` (1.56% Gen 1 vs. -3.76% Gen 3). Using GRACE's specialized pass pool (*SpecPool*) for mutations also generally enhances performance by focusing the search. The `CandInit-SpecPool` configuration, combining both strategies,

Table 3. Ablation study: Converged Average OverOz (%) and Convergence Generation for different configurations. Higher OverOz is better. Earlier convergence is better.

Dataset	RandInit-AllPasses	CandInit-AllPasses	RandInit-SpecPool	CandInit-SpecPool
blas	-3.76 (3)	**1.56** (**1**)	-0.97 (6)	**1.56** (**1**)
cbench	-11.58 (7)	3.84 (**1**)	-9.55 (10)	**3.88** (10)
chstone	2.48 (9)	**7.32** (8)	2.05 (10)	6.97 (**1**)
mibench	0.98 (10)	**5.01** (**1**)	1.58 (9)	**5.01** (**1**)
npb	20.12 (10)	13.88 (**1**)	**21.28** (10)	13.89 (9)
opencv	0.01 (10)	0.75 (**1**)	0.77 (10)	**1.28** (10)
tensorflow	**5.02** (10)	1.10 (10)	-0.26 (10)	1.10 (**1**)
Avg. Conv. Gen	1.90 (8.43)	4.78 (**3.29**)	2.13 (9.29)	**4.81** (4.71)

yields a strong balance of rapid convergence and high quality solutions, significantly outperforming the baseline `RandInit-AllPasses`. Specifically, the average OverOz for `CandInit-SpecPool` is **4.81%**, substantially better than **1.90%** achieved by `RandInit-AllPasses`. These findings validate the efficacy of informed initialization and a constrained search space.

A closer analysis of the components reveals their distinct roles. The informed initialization (*CandInit*) is the primary driver of the performance increase. The specialized pass pool (*SpecPool*), in contrast, contributes a smaller increment to the final OverOz score. Its main function is to improve search efficiency by constraining the mutation space, which is reflected in the faster or equivalent convergence generations observed in our experiments. These findings validate the efficacy of both informed initialization for setting a high-quality starting point and a constrained search space for guiding the subsequent optimization process efficiently.

Table 4. Impact of Test-Time Refinement Strategies on Average OverOz (%).

Dataset	None	GA-Seq	Prefix Search	Oz Fallback
cbench	6.39	**6.54**	6.39	6.51
mibench	7.04	7.09	7.07	**7.26**
chstone	10.63	**10.80**	10.72	10.63
tensorflow	6.89	6.93	6.89	**6.95**
npb	28.90	**29.00**	28.91	28.90
opencv	5.67	5.68	5.67	**5.77**
blas	5.15	**5.18**	5.15	5.15
Average	10.09	**10.17**	10.11	**10.17**

We evaluated GRACE's test-time refinement strategies: using the best coreset directly (None), applying GA-Seq Refinement, using Prefix Search Refinement, and the Oz Fallback. Table 4 summarizes the average OverOz (%).

The results show that both the GA-Seq refinement and the Oz Fallback configurations achieve the highest average OverOz of **10.17%**. GA-Seq demonstrates that a localized evolutionary search can fine-tune coreset sequences for improved average gains (e.g., in cbench), while the Prefix Search refinement yields a minor improvement (10.11%) over the non-refinement baseline (10.09%).

The role of the Oz Fallback, however, is particularly crucial for ensuring the framework's robustness. While its average performance matches the best, its primary value lies in eliminating any performance regressions compared to the standard `opt -Oz`. This is not just a theoretical safeguard; it addresses a practical limitation inherent in any learning-based approach. To understand this, we analyzed the approximately 4% of cases where our unrefined coreset sequences underperformed `opt -Oz`. Our qualitative analysis revealed that these programs often fall into two categories: (1) extremely small programs where `opt -Oz` is already near-optimal and any deviation is harmful, or (2) programs with highly unusual structures not well-represented in our training set.

For these predictable, albeit infrequent, edge cases, the Oz Fallback acts as an essential safety net. As seen in datasets like `mibench`, this mechanism not only prevents negative outcomes but can even improve the average OverOz by correcting suboptimal coreset selections, highlighting its corrective capabilities. Thus, while GA-Seq actively seeks to push the performance ceiling higher, the Oz Fallback is vital for establishing a reliable performance floor, guaranteeing robust and high-quality results in practice.

5.5 Case Study: Coreset Effectiveness

Table 5. Performance of GRACE-derived Sequences Relative to `opt -Oz` for Test-Time Refinement Strategies (exclude Oz fallback). Shows Success% (GRACE greater than `opt Oz`) and, in parentheses, WorseThanOz% (GRACE less than `opt Oz`). All values are percentages (%).

Dataset	None (% S / (% W))	GA-Seq (% S / (% W))	Prefix Search (% S / (% W))
cbench	72.7 (18.2)	72.7 (18.2)	72.7 (18.2)
mibench	57.50 (5.0)	57.5 (5.0)	60.0 (5.0)
chstone	100.00 (0.0)	100.0 (0.0)	100.0 (0.0)
tensorflow	54.44 (2.2)	54.4 (2.2)	54.4 (2.2)
npb	96.69 (0.0)	96.7 (0.0)	96.7 (0.0)
opencv	59.38 (3.1)	59.4 (3.1)	59.4 (3.1)
blas	96.55 (0.0)	96.6 (0.0)	96.6 (0.0)
Avg.	76.76(4.07)	76.76(4.07)	77.10(4.07)

To provide a nuanced view of coreset effectiveness, Table 5 details the performance of GRACE-derived sequences relative to `opt -Oz`. It presents the Success% (GRACE sequence outperforms `opt -Oz`) and, in parentheses, the WorseThanOz% (GRACE sequence is worse than `opt -Oz`) for different refinement strategies. The remaining percentage (100% - Success% - WorseThanOz%) represents cases where the GRACE sequence performed identically to `opt -Oz`. The average Success% is high across strategies, indicating that GRACE's core methodology frequently surpasses the `opt -Oz` baseline. Consequently, the average WorseThanOz% is relatively low, signifying that only a small fraction of programs see GRACE's sequences underperform `opt -Oz`. This 4% highlights the remaining optimization potential where GRACE could be improved to better `opt -Oz` directly.

The instances where GRACE performs identically to `opt -Oz` (averaging around 100% - 77% - 4% = 19%) are also noteworthy. These EqualToOz% cases can arise from two scenarios: either both GRACE and `opt -Oz` have effectively reached the practical limit of instruction count reduction for those specific programs with the available passes, or there was a remaining optimization potential for which GRACE's chosen coreset sequence happened to match but not exceed the performance of `opt -Oz`. Interestingly, while GA-Seq refinement improves the average OverOz magnitude (Table 4),

it does not substantially alter the Success% or WorseThanOz% distributions compared to using the coreset directly (*None*). This suggests that its benefits are mainly concentrated in further optimizing programs where GRACE already had an advantage or was equivalent to `opt -Oz`, rather than converting cases from *WorseThanOz* or *EqualToOz* to *Success*. This points to a need for refinement techniques or coreset strategies that more effectively target these harder-to-improve scenarios relative to the `opt -Oz` baseline.

6 Limitations and Future Work

Dependency on Training Data. Like most learning-based approaches, GRACE's effectiveness is contingent on the quality and diversity of the training corpus. The pass synergies, learned program embeddings, and the final evolved coreset are all derived from the training data. Consequently, the framework's performance on programs from highly specialized or entirely unseen domains may be suboptimal if those domains are not well-represented in the training set. Future work could explore techniques for domain adaptation or online learning to allow GRACE to dynamically adjust its coreset when compiling a large corpus of new software.

Expressiveness of Program Representation. Our current framework relies on Autophase features as the input for our representation learning module. While effective, these hand-crafted static features may not capture all the semantic nuances relevant to optimization. A promising direction for future work is to explore more powerful program representations, such as those derived from graph neural networks (GNNs) applied to the program's control-flow or data-flow graphs, which could potentially learn more fundamental structural properties.

Extension to Multi-Objective Optimization. The current instantiation of GRACE is designed for a single objective: reducing the LLVM IR instruction count. However, real-world compiler optimization often involves multi-objective trade-offs, such as balancing code size, execution speed, and compile time. Extending GRACE to handle such multi-objective problems would require significant modifications, particularly to the fitness function of the evolutionary algorithm (e.g., by using techniques like NSGA-II) and potentially the pass synergy analysis. This remains a challenging but important direction for future exploration.

Broader Applicability. While GRACE's principles are general, its current implementation is tailored for the LLVM ecosystem. Adapting the framework to other compiler back-ends, such as GCC, would be a non-trivial engineering effort requiring a complete redesign of the compiler interface and a re-evaluation of the pass synergy landscape. Investigating the integration of advanced ML models, such as Large Language Models (LLMs), to guide the various stages of GRACE also presents a promising avenue for further improvement across different compiler platforms.

7 Conclusion

This paper presented **GRACE**, a framework for compiler pass auto-tuning with a specific focus on reducing LLVM IR instruction count. GRACE addresses the long-standing challenges of the compiler phase-ordering problem by introducing a structured, multi-stage methodology to navigate the optimization space. Rather than applying a single monolithic search, the framework begins with global pass synergy analysis to initialize the process with a pool of candidate sequences. It then applies a "divide-and-conquer" strategy, supported by a contrastive learning approach that generates program embeddings for similarity-aware clustering. This partitioning allows more targeted exploration within each program cluster, where evolutionary algorithms search for a diverse set of representative sequences. Finally, the framework incorporates test-time refinement

and a fallback strategy to support practical deployment, aiming to provide both effectiveness and robustness.

Our evaluation on seven datasets and two LLVM versions shows that GRACE consistently reduces instruction counts relative to the `opt -Oz` baseline. On average, it achieved reductions of **10.09%** on LLVM 10.0.0 and **10.19%** on LLVM 18.1.6. Importantly, the framework achieves this improvement with very low overhead: the average tuning time is less than one second per program on modern hardware. These results suggest that GRACE can be a practical option for integration into compilation pipelines.

8 Data Availability

A replication package, including code and data used in our experiments, is available at https://github.com/Panhaolin2001/GRACE/ for review. Upon acceptance, we intend to make the data fully publicly available under an appropriate license, subject to anonymization and privacy constraints.

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] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics* 9, 8 (2020), 1295.

[3] 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.

[4] 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.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *Advances in neural information processing systems* 31 (2018).

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

[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, 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 preprint arXiv:2407.02524* (2024).

[12] 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.

[13] 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).

[14] 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).

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

[16] 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.

[17] 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.

[18] 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.

[19] Ruobing Han and Hyesoon Kim. 2024. Exponentially expanding the phase-ordering search space via dormant information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 250–261.

[20] 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.

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

[22] 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).

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[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] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

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

[34] Chao Zha, Zhiyu Wang, Yifei Fan, Bing Bai, Yinjie Zhang, Sainan Shi, and Ruyun Zhang. 2025. A-NIDS: Adaptive Network Intrusion Detection System Based on Clustering and Stacked CTGAN. *IEEE Transactions on Information Forensics and Security* (2025).

[35] 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.