Synergy-Guided Compiler Auto-Tuning of Nested LLVM Pass Pipelines

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
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 optimization relies on sequences of passes to improve program performance. Selecting and ordering these passes automatically, known as compiler auto-tuning, is challenging due to the large and complex search space. Existing approaches generally assume a linear sequence of passes, a model compatible with legacy compilers but fundamentally misaligned with the hierarchical design of the LLVM New Pass Manager. This misalignment prevents them from guaranteeing the generation of syntactically valid optimization pipelines. In this work, we present a new auto-tuning framework built from the ground up for the New Pass Manager. We introduce a formal grammar to define the space of valid nested pipelines and a forest-based data structure for their native representation. Upon this foundation, we develop a structure-aware Genetic Algorithm whose operators manipulate these forests directly, ensuring that all candidate solutions are valid by construction. The framework first mines synergistic pass relationships to guide the search. An optional refinement stage further explores subtle performance variations arising from different valid structural arrangements.

We evaluate our approach on seven benchmark datasets using LLVM 18.1.6. The discovered pipelines achieve an average of 13.62% additional instruction count reduction compared to the standard `opt -Oz` optimization level, showing that our framework is capable of navigating this complex, constrained search space to identify valid and effective pass pipelines.

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

Modern compilers use many optimization passes to improve program performance. Selecting and ordering these passes to form an effective sequence is a central task in compiler engineering. Standard optimization levels such as `-Oz` offer a general-purpose solution, but they are often not optimal for specific applications. Compiler auto-tuning aims to reduce this gap by automatically finding program-specific pass sequences. The main challenge is the very large search space of possible combinations. This difficulty has motivated research into heuristic search and machine learning approaches that can discover better customized strategies.

Most existing auto-tuning research focused on legacy compiler infrastructures, such as the LLVM legacy Pass Manager, where optimization pipelines are modeled as linear sequences. In

*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; 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.

this setting, the key challenge is the **combinatorial explosion**: for N passes and a sequence of length k, the number of possible permutations grows factorially, which makes exhaustive search infeasible. To address this, many studies explored efficient search strategies in the sequential space. Early work in **iterative compilation (IC)** [2, 6] showed the effectiveness of empirical search guided by heuristics such as Genetic Algorithms [16], though with high compilation cost. Later work introduced **machine learning-based methods** [22], which built predictive models linking program features to useful optimizations. These included supervised learning to predict pass benefits [30] and reinforcement learning agents that treat pass ordering as a sequential decision process [13, 18]. More recently, **Large Language Models (LLMs)** have been applied to learn optimization patterns directly from code [12]. Although diverse in methodology, these approaches share the simplifying assumption of a flat, sequential pipeline, which no longer reflects current compiler infrastructures.

The LLVM New Pass Manager (New PM) introduces a hierarchical design that reflects the structure of the Intermediate Representation (**Module > CGSCC > Function > Loop**). This architectural shift renders simple linear pass sequences not just insufficient, but **invalid**. Any auto-tuner for the New PM faces a primary, non-negotiable requirement: it **must** produce syntactically valid nested pipelines that conform to the New PM's grammar. Consequently, existing approaches that manipulate flat sequences are unusable in this new context, as they lack the fundamental mechanisms to ensure structural validity and navigate the constrained search space.

This fundamental requirement motivates our work: to create a complete auto-tuning framework built from the ground up for the LLVM New PM. To achieve this, we first establish a formal foundation for the problem. We introduce a grammar that precisely defines the space of all valid hierarchical pipelines and propose a logical forest data structure for their direct in-memory representation. This representation is the cornerstone of our structure-aware Genetic Algorithm. Unlike traditional approaches that manipulate linear sequences, its genetic operators (e.g., subtree crossover and mutation) operate directly on the forest structure itself. This design choice provides a guarantee: every candidate pipeline generated during the evolutionary search is, by construction, syntactically valid. To navigate this vast space of valid pipelines efficiently, the search is guided by a knowledge base of synergistic pass relationships mined offline. Furthermore, the framework leverages its deep structural awareness through an optional refinement stage, enabling it to explore small performance nuances between different valid nesting structures.

We implemented and evaluated our framework on seven benchmark datasets against the `opt -Oz` optimization level of LLVM 18.1.6. Our pipelines achieve an average of **13.62%** additional reduction in instruction count compared to `opt -Oz`, and they perform better than existing auto-tuning methods.

This paper makes the following contributions:

- We provide the first formalization of the auto-tuning problem for the LLVM New Pass Manager's hierarchical architecture. We introduce a grammar to describe the space of valid nested pipelines and propose a logical forest data structure for their representation and manipulation.
- We propose a complete methodology for discovering high-quality pass sequences that is aware of structural constraints. This includes a new **synergy mining technique** to build a knowledge graph of structured pass relationships, and a **Synergy-Guided Genetic Algorithm** that uses this graph to construct high-performance hierarchical pass sequences.
- We identify and analyze the performance impact of different valid nesting structures for a fixed pass sequence. We then introduce a targeted refinement mechanism that leverages

this insight to capture modest but consistent performance improvements, demonstrating the ability to exploit structural variations for additional optimization.
- We evaluate our framework and show that it achieves **superior instruction count reduction compared to existing auto-tuning methods** on the LLVM Intermediate Representation. Our method consistently outperforms LLVM's standard optimization `opt -Oz` levels across a wide set of benchmarks.

2 Background and Motivation

2.1 Background

The LLVM compiler infrastructure's New PM constitutes an architectural change from the earlier linear optimization pipeline. It organizes transformations and analyses in a hierarchical manner, supporting module-level, function-level, CGSCC-level and loop-level passes within a unified framework. The design provides mechanisms for finer-grained control of pass execution, improved analysis reuse, and support for parallelism in both analysis and transformation. These features reflect a response to the growing scale and complexity of modern software and hardware, and they establish a foundation for exploring compiler optimization and auto-tuning in a more flexible setting.

A key characteristic of the New PM is its replacement of the conventional flat-list pipeline with a hierarchical organization that reflects the nested structure of the IR. This hierarchy comprises four principal levels of IR units: **Module**, the top-level container for a translation unit; **CGSCC** (Call Graph Strongly Connected Component), a unit designed for inter-procedural optimizations; **Function**, a single function body; and **Loop**, a natural loop within a function's control-flow graph. Each optimization pass is typed to operate at exactly one of these granularities, and its placement within a pipeline must respect this hierarchy. For instance, a `LoopPass` can only appear within a `loop` manager, which itself must be contained inside a `function` manager.

This hierarchical model is realized through a system of pass managers and adaptors. Consequently, an optimization pipeline is no longer a simple, unstructured sequence. Instead, it assumes a specific, organized form based on how passes are grouped and nested, a configuration we refer to as a pipeline *skeleton*. The skeleton determines the precise scope and context for each pass. This design provides several significant advantages over the legacy system. First, it enables a more refined and efficient *analysis management* mechanism: analyses are computed lazily, cached, and invalidated with fine-grained precision, thereby avoiding unnecessary recomputation. For example, a pass that modifies a single function invalidates only the analyses pertaining to that function, leaving all others unaffected. Second, this structured grouping improves *cache locality* during compilation. By executing multiple `FunctionPasses` consecutively on one function before moving to the next, the compiler can make better use of CPU caches. These characteristics show that the grouping and nesting of passes—the skeleton—is not only a syntactic requirement, but also a structural feature with direct implications for compilation efficiency and, as we will argue, for the quality of the generated code.

2.2 Motivation

The hierarchical design of the LLVM New Pass Manager introduces a fundamental challenge for auto-tuning that poses fundamental challenges to traditional sequence-based approaches, which cannot guarantee syntactic validity. The primary issue is one of **syntactic validity**: any generated optimization pipeline must conform to the strict nesting rules of the New PM's grammar. An auto-tuner that generates syntactically incorrect pass strings would simply fail at compile-time, rendering the entire search process ineffective. Approaches that treat the pipeline as a flat list

cannot guarantee this conformance, making them fundamentally unsuitable. This establishes the baseline requirement: a modern auto-tuner must be built upon a formal representation capable of modeling and manipulating these hierarchical structures.

Beyond this primary challenge of validity, however, lies a more subtle question: does the specific choice of a valid hierarchical structure impact the quality of the final optimized code? To investigate this, we conducted an experiment to isolate the effect of the pipeline's structure while keeping the sequence of optimization passes fixed. We applied a fixed sequence of passes—`globalopt` (M), `inline` (C), `gvn` (F), and `loop-deletion` (L)—to several CBench programs using five distinct, but equally valid, nesting skeletons that preserve the pass order. These skeletons are defined as follows:

- **Fully Sequential:** `module(M), module(cgscc(C)), module(function(F)), module(function(loop(L)))`
- **F+L Combined:** `module(M), module(cgscc(C)), module(function(F, loop(L)))`
- **M+C, F+L Combined:** `module(M, cgscc(C)), module(function(F, loop(L)))`
- **C+F+L Combined:** `module(M), module(cgscc(C, function(F, loop(L))))`
- **Fully Nested:** `module(M, cgscc(C, function(F, loop(L))))`

Table 1. Instruction count variation due to skeleton structure for representative CBench programs where the impact is most notable. All tests use the fixed pass sequence (`globalopt, inline, gvn, loop-deletion`), with the lowest instruction count for each program highlighted in bold.

Program	Original IC	Skeleton 1 (Sequential)	Skeleton 2 (F+L Comb.)	Skeleton 3 (M+C, F+L Comb.)	Skeleton 4 (C+F+L Comb.)	Skeleton 5 (Fully Nested)
dijkstra	450	**372**	**372**	**372**	481	481
patricia	1282	**958**	**958**	**958**	1042	1042
qsort	638	627	627	627	**601**	**601**
sha	799	**687**	**687**	**687**	758	758
stringsearch	1348	1109	1109	1109	**1108**	**1108**

As summarized in Table 1, the results demonstrate that for the same sequence of passes, different nesting structures can indeed lead to different performance outcomes. This behavior arises because the nesting structure dictates the precise execution order and scope of optimizations across different IR units (such as functions). For example, a flatter, sequential structure might apply one pass to all functions before starting the next, while a deeply nested structure applies a sequence of passes to each function individually before moving to the next. This can alter which intermediate program states are visible to subsequent passes, thereby enabling or disabling different optimization opportunities.

While the primary challenge for any New PM auto-tuner is generating valid pipelines, these findings reveal an important secondary dimension to the problem: the choice among valid structures is not trivial and can influence optimization quality. This observation solidifies the need for a framework that does more than simply generate *one* valid pipeline. A truly effective approach must treat the hierarchical structure as a first-class, manipulable entity, enabling the exploration of the space of valid hierarchical configurations.

3 Formal Representation of Pass Pipelines

To address the validity challenge outlined in our motivation and to systematically explore the structured search space of the LLVM New Pass Manager, we first define a formal representation for valid pass pipelines. The conventional one-dimensional list does not fully capture the hierarchical and nested nature of the New PM, nor can it guarantee syntactic correctness. Therefore, we represent

a pass pipeline using two complementary formalisms: a Context-Free Grammar that specifies its syntactic validity, and a logical *forest* data structure that facilitates its algorithmic manipulation.

3.1 A Formal Grammar for Pipeline Skeletons

We define the set of all syntactically valid pass pipelines using a Context-Free Grammar, $G = (V, \Sigma, R, S)$. This grammar captures the hierarchical and recursive structure of the LLVM New Pass Manager, ensuring that only well-formed pipelines are considered during tuning.

- **Non-terminals** V: {⟨Pipeline⟩, ⟨ModuleManager⟩, ⟨ModuleElement⟩, ⟨CGSCCManager⟩, ⟨CGSCCElement⟩, ⟨FunctionManager⟩, ⟨FunctionElement⟩, ⟨LoopManager⟩, ⟨LoopElement⟩, ⟨ModulePass⟩, ⟨CGSCCPass⟩, ⟨FunctionPass⟩, ⟨LoopPass⟩}
- **Terminals** Σ: {"`module(`", "`cgscc(`", "`function(`", "`loop(`", "`)`", "`,`", pass_names}
- **Start Symbol** S: ⟨Pipeline⟩

The production rules R are defined as follows. The Kleene star ($*$) denotes zero or more comma-separated repetitions of an element.

R1. ⟨Pipeline⟩ → ⟨ModuleManager⟩ | ⟨ModuleManager⟩`,`⟨Pipeline⟩
R2. ⟨ModuleManager⟩ → `module(`(⟨ModuleElement⟩`,`)*⟨ModuleElement⟩`)`
R3. ⟨ModuleElement⟩ → ⟨ModulePass⟩ | ⟨ModuleManager⟩ | ⟨CGSCCManager⟩ | ⟨FunctionManager⟩
R4. ⟨CGSCCManager⟩ → `cgscc(`(⟨CGSCCElement⟩`,`)*⟨CGSCCElement⟩`)`
R5. ⟨CGSCCElement⟩ → ⟨CGSCCPass⟩ | ⟨CGSCCManager⟩ | ⟨FunctionManager⟩
R6. ⟨FunctionManager⟩ → `function(`(⟨FunctionElement⟩`,`)*⟨FunctionElement⟩`)`
R7. ⟨FunctionElement⟩ → ⟨FunctionPass⟩ | ⟨FunctionManager⟩ | ⟨LoopManager⟩
R8. ⟨LoopManager⟩ → `loop(`(⟨LoopElement⟩`,`)*⟨LoopElement⟩`)`
R9. ⟨LoopElement⟩ → ⟨LoopPass⟩ | ⟨LoopManager⟩

This grammar enforces the hierarchical constraints of the LLVM New Pass Manager. For example, a `module` manager (R3) can directly host `cgscc` and `function` managers, but `loop` managers are introduced only under `function` managers (R7). The recursive definitions (e.g., R5, R7, R9) allow managers to be nested at the same level, supporting staged optimizations over the same IR unit. Under these rules, pipelines generated from this grammar are syntactically valid.

3.2 The Skeleton as a Forest Data Structure

While the grammar defines the string representation, a logical data structure is required for algorithmic manipulation. We model a valid pipeline skeleton as a *forest*—an ordered collection of trees, as illustrated in Figure 1. Each tree represents an independent optimization stage, with manager nodes as internal nodes and terminal optimization passes as leaf nodes.

The structural components are defined as follows:

- **Forest:** Corresponds to the grammar's start symbol ⟨Pipeline⟩ and consists of a list of *trees*, representing sequential optimization phases.
- **Tree:** Rooted by a ⟨ModuleManager⟩, representing a single optimization stage.
- **Depth:** Maximum nesting level of managers within a tree, indicating the granularity of the optimization strategy.
- **Width:** Number of child nodes within a manager, representing the length and complexity of the optimization sequence at a given level.

These structural properties—forest size, tree composition, depth, and width—define the organization of a pipeline. For instance, a single, deep, narrow tree corresponds to a specialized optimization

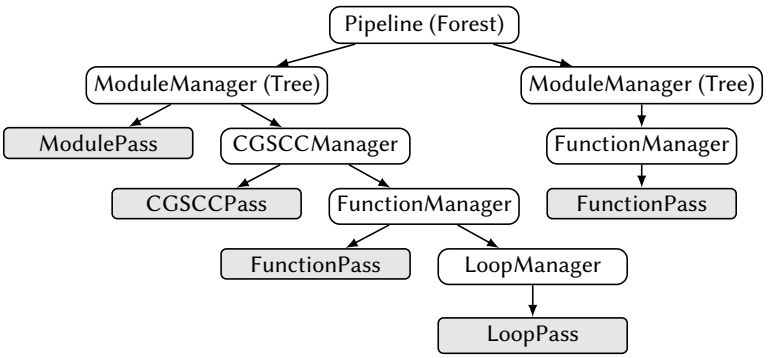


Fig. 1. Logical *Forest* representation of a Pass Pipeline. Each root ⟨ModuleManager⟩ forms a tree. Non-leaf nodes are managers, and leaf nodes are terminal passes.

strategy, while a forest of shallow, wide trees corresponds to a phased, broader strategy. This structured representation serves as the foundation for knowledge-guided and evolutionary algorithms, enabling systematic manipulation of both pass selection and hierarchical arrangement.

4 Methodology

4.1 Framework Overview

Navigating the hierarchical and syntactically constrained search space of the LLVM New Pass Manager presents challenges that motivate the need for a new auto-tuning approach. To address this, we propose a hybrid framework that combines offline knowledge mining with online, structure-aware evolutionary search. The design principle is to maintain syntactic validity at all stages while exploring the large space of pass configurations efficiently.

As illustrated in Figure 2, our framework consists of three stages that together form an integrated search strategy: (1) **Offline Synergy Mining:** In a one-time offline phase, we construct a **Synergy Knowledge Graph** by analyzing pass interactions across a large corpus of programs. This graph encodes structure-aware relationships between passes, providing heuristic knowledge that improves the tractability of the subsequent online search. (2) **Online Guided Evolutionary Search:** The core of the framework is a **structure-aware Genetic Algorithm (GA)**. Operating directly on the forest data structure defined in Section 3, its genetic operators guarantee that every generated pipeline is syntactically valid. The search is guided by the Synergy Knowledge Graph to converge towards regions of the search space associated with better performance. (3) **Optional Heuristic Refinement:** Finally, the framework includes a refinement stage. This step builds on the earlier observation that different valid nesting structures can affect performance, and makes localized heuristic adjustments to pipeline hierarchy to obtain modest additional improvements.

This multi-stage methodology decomposes the auto-tuning problem for the New PM into manageable components, aiming to balance syntactic validity with the potential for performance improvement.

4.2 Offline Synergy Mining

4.2.1 Rationale for Synergy Mining. The search space of valid pass pipelines in the LLVM New Pass Manager is combinatorially large, making a blind or purely random search for high-performance solutions infeasible. To address this, our framework employs a knowledge-guided approach that includes an offline phase to pre-mine beneficial pass interactions, or **synergistic pairs**. These serve

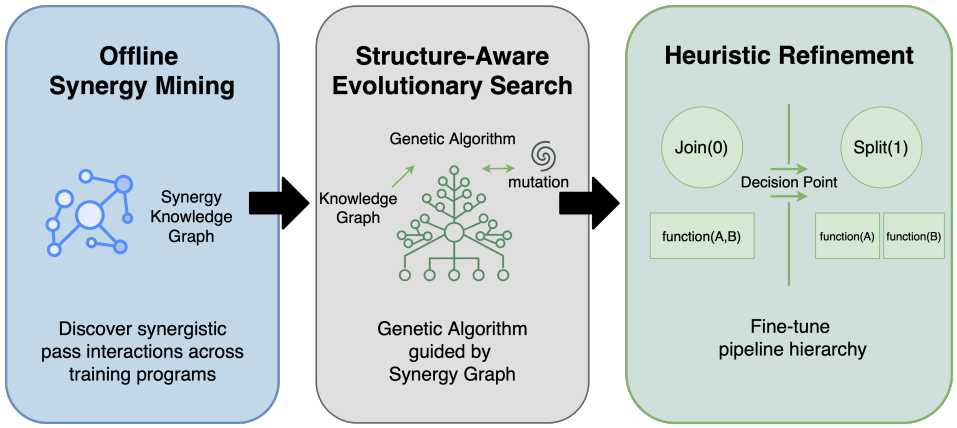
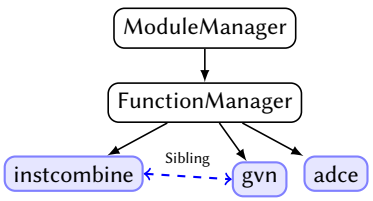


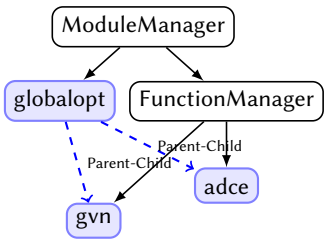
Fig. 2. Overview of the auto-tuning framework. The offline mining produces a Synergy Knowledge Graph, which guides the generation of candidate pass sequences. Heuristic refinement is then applied to improve the hierarchical arrangement.

two main purposes: (1) **Guiding the Search Process:** A pre-computed knowledge base of synergistic pairs provides a heuristic for the online search. Rather than exploring the space uniformly, the search algorithm can prioritize sequences composed of passes that have been observed to interact positively. This focuses the search on regions of the space with higher expected performance, potentially improving search efficiency and the likelihood of discovering higher-performance pipelines. (2) **Constructing Candidate Solutions:** Synergistic pairs act as building blocks informed by prior observations. This knowledge can be used to generate initial candidate solutions that incorporate known beneficial pass interactions. It can also inform modification operators (e.g., mutation), favoring changes that are more likely to lead to performance improvements, thereby enhancing the overall quality of solutions explored by the search.

As shown in Figure 3, because the New PM organizes passes hierarchically, we distinguish between two types of synergy: **Intra-Level Synergy**, between passes at the same level, and **Inter-Level Synergy**, between passes at different levels. Encoding this distinction in the knowledge graph allows the subsequent search to be both guided by prior observations and structure-aware.



(a) Intra-Level Synergy Example:
`module(function(instcombine, gvn))`



(b) Inter-Level Synergy Example:
`module(globalopt, function(licm, adce))`

Fig. 3. Illustration of the two primary structural synergy types. Blue nodes highlight the pass pair under evaluation. (a) shows two `FunctionPasses` as siblings. (b) shows a `ModulePass` providing context for a `FunctionPass`.

4.2.2 Decoupling Sequence from Micro-Structure for Tractable Mining. A key challenge in analyzing pass interactions within the New PM is the entanglement of pass sequence and nesting structure. In this initial stage, the goal is to identify first-order synergistic relationships between passes that are empirically observed to affect performance. To this end, we temporarily decouple the pass sequence from its micro-level structural arrangement.

To validate this simplification, we systematically evaluated the impact of micro-structures. Even for a minimal two-pass sequence, the New PM allows multiple valid structural arrangements depending on the pass types. For our experiment, we considered the following representative cases:

Intra-Level pairs of two Function passes (F_1, F_2), three structural variants:

(1) **Micro:** tightly coupled within a single manager. Skeleton: `module(function(`F_1, F_2`))`
(2) **Meso:** sequential within the same tree, sharing a module-level context. Skeleton: `module(function(`F_1`), function(`F_2`))`
(3) **Macro:** completely independent optimization stages. Skeleton: `module(function(`F_1`)), module(function(`F_2`))`

Inter-Level pairs of a Module pass (M_1) and a Function pass (F_1), two variants:

(1) **Nested:** both inside a single tree. Skeleton: `module(`M_1`, function(`F_1`))`
(2) **Phased:** placed in separate stages. Skeleton: `module(`M_1`), module(function(`F_1`))`

To quantify the impact of micro-structures on pairwise synergies, we conducted a large-scale experiment on a subset of the training programs (used only for discovering synergistic pairs, not for final testing). We randomly sampled a large number of two-pass pairs, spanning both intra- and inter-level combinations, and evaluated each pair across all applicable structural variants. The results were consistent: in over 99.7% of cases, different micro-structures for a given pass pair produced identical final instruction counts.

These results provide empirical evidence that for simple two-pass interactions, the dominant performance factors are the choice and order of the passes themselves, while the micro-level nesting structure appears to have minimal impact. A small fraction of cases (<0.3%) showed minor variations. The observed trend supports a pragmatic divide-and-conquer strategy for our framework. Accordingly, the offline mining phase focuses on capturing primary, sequence-oriented synergistic effects, temporarily decoupled from the less impactful structural variable. More subtle and relatively infrequent performance effects of nesting structure are addressed in a later, dedicated refinement stage (Section 4.4). This approach ensures that the framework first establishes a foundation based on pass sequence and then performs fine-grained structural adjustments where they may provide additional benefit.

4.2.3 Approach: Systematic Discovery and Quantification. Our methodology for mining synergistic relationships is encapsulated in Algorithm 1. The design of this algorithm is directly informed by the findings in the preceding section. Based on the evidence that micro-structure appears to have limited impact on pairwise interactions, we adopt a unified and efficient approach. We systematically evaluate all considered two-pass combinations using a single, representative skeleton, regardless of whether the pair is intra-level or inter-level. This allows us to isolate and quantify the primary, sequence-oriented synergistic effects across the training programs.

Synergy Knowledge Graph:

- **Node:** each node represents an individual compiler pass.
- **Edge:** a directed edge ($P_1 \rightarrow P_2$) indicates a positive synergy, i.e., P_1 followed by P_2 yields better performance than the sum of their individual effects.
- **Edge type:** labeled as *Intra-Level* (both passes in the same manager) or *Inter-Level* (passes across different managers).

Algorithm 1 Synergy Knowledge Graph Construction

Require: Dataset D, PassDatabase PDB
$KB\_counts \leftarrow$ new defaultdict(list)
for all program $p \in D$ **do**
  $IR \leftarrow$ load_ir(p); $IC_{orig} \leftarrow$ get_instruction_count(IR)
  $BaselinePerf \leftarrow$ calculate_single_pass_performance(IR, IC_{orig}, PDB)
  for all structured pair (P_1, P_2) from all passes in PDB **do**
    $S_{combined} \leftarrow$ build_representative_skeleton(P_1, P_2)
    $T \leftarrow$ get_synergy_type($S_{combined}$)
    $IC_{combined} \leftarrow$ evaluate($IR, S_{combined}$)
    $\text{Perf}_{combined} \leftarrow IC_{orig} - IC_{combined}$
    $\text{Perf}_{sum} \leftarrow BaselinePerf[P_1] + BaselinePerf[P_2]$
    if $\text{Perf}_{combined} > \text{Perf}_{sum}$ **then**
      $KB\_counts[P_1.\text{name}].\text{append}(\{P_2.\text{name}, T\})$
    end if
  end for
end for
$KnowledgeGraph, StartWeights \leftarrow$ normalize_and_build(KB_counts)
return $KnowledgeGraph, StartWeights$

- **Weight:** computed uniformly by counting occurrences of positive synergy across the dataset, then normalized across all outgoing edges from P_1.

The algorithm first establishes a performance baseline for every individual pass. The main loop then iterates through all valid structured pairs. For each pair (P_1, P_2), it constructs a single, representative, nested skeleton and evaluates its performance. The synergy is quantified by comparing the combined benefit against the sum of individual benefits. If a positive synergy is detected, the directed relationship from P_1 to P_2, including its structural type, is recorded . This recording is asymmetric to capture the order-dependent nature of pass interactions. After processing all programs, the aggregated counts are normalized to generate the final weighted Synergy Knowledge Graph and start-pass probabilities .

4.3 Structure-Aware Evolutionary Search

With the Synergy Knowledge Graph providing heuristic guidance, the next stage of our framework employs a search algorithm that can systematically explore the vast space of valid pipelines. The primary requirement for this algorithm is that it must operate natively on the hierarchical forest representation defined in Section 3 and guarantee that all operations preserve syntactic validity. To this end, we design a **structure-aware Genetic Algorithm (GA)** whose components are tailored to this constrained search space. Each individual in the GA is represented as a forest data structure corresponding to a complete and valid pipeline, and the search process is guided by the pre-computed knowledge graph $G_S = (V, E, W, T)$.

4.3.1 Initial Population Generation. The initial population is generated by performing a **Weighted Random Walk** on the knowledge graph G_S. This approach seeds the population with combinations of passes observed to be synergistic, providing an informed initialization for the subsequent search. For each individual, the process is as follows:

(1) **Start Pass Selection:** A starting pass p_0 is selected according to a probability distribution D_{start}. The probability $D_{\text{start}}(p_i)$ is proportional to the frequency with which pass p_i was observed as an initiator of positive synergy during the offline mining phase.
(2) **Sequence Extension:** Given the last pass in the current sequence, p_k, the next pass p_{k+1} is chosen from its set of successors $N(p_k)$ in G_S. The selection probability is proportional to the synergy weight:

$$P(p_{k+1}|p_k) \propto W(p_k, p_{k+1}), \quad \forall p_{k+1} \in N(p_k)$$

The placement of p_{k+1} within the forest structure is determined by the synergy's edge type $T(p_k, p_{k+1})$. For example, an *Intra-Level* synergy may place p_{k+1} as a sibling to p_k within the same pass manager, whereas an *Inter-Level* synergy may introduce a new nested manager. This process continues until a predefined maximum sequence length is reached.

4.3.2 Crossover and Mutation. The crossover operator generates offspring by exchanging structural subtrees between two parent individuals. For each parent, a random subtree (defined as a manager and its contained passes) is selected. These two subtrees are swapped to produce two offspring. After the swap, the resulting individuals are validated against the syntactic rules; if a new structure is invalid, the crossover is discarded.

The mutation operator includes both addition and replacement operations. A random pass within the individual's structure is selected as an *anchor*. The algorithm queries the synergy knowledge graph to identify passes with known synergetic relationships to the anchor. One such partner is preferentially chosen for addition or replacement. If no suitable partner is found or the operation cannot be performed, the algorithm falls back to inserting or replacing with a randomly selected valid pass, thereby promoting structural diversity in the population.

4.4 Heuristic Refinement

In our methodology, we decoupled pass sequence from micro-structure during the initial knowledge mining phase (Section 4.2.2). We now return to this second optimization variable. As shown in Table 1, the hierarchical structure of a pipeline, even for a fixed pass sequence, can alter the final optimization result. This section examines the mechanism behind this effect and introduces the final stage of our framework, which is designed to systematically explore it.

4.4.1 Impact of Nested Pipeline Structures. Consider two function passes `A` and `B` (e.g., `InstCombine` and `GVN`) and three common ways of nesting them within a module pipeline:

(1) `module(function(A,B))`: Each function undergoes `A` followed by `B` sequentially.
(2) `module(function(A)), module(function(B))`: All functions in the module first execute `A`, then all execute `B`.
(3) `module(function(A), function(B))`: Semantically equivalent to the previous case; all functions first run `A`, then all run `B`.

Assume a module with two functions, `foo` and `bar`, where `foo` calls `bar`. The execution order differs across nesting styles:

- In `module(function(A,B))`, `foo` executes `A` and `B` before `bar` has been optimized. The transformation seen by `foo` depends on the unoptimized `bar`.
- In `module(function(A)), module(function(B))` or `module(function(A), function(B))`, all functions first execute `A`, so when executing `B`, `foo` sees the IR of `bar` after `A`. This can enable or disable certain optimizations compared to the first structure.

Observation: The same set of passes, even when applied in the same order, can produce different optimization effects solely due to structural differences in the pipeline hierarchy. This indicates

that pass execution is determined not only by the sequence but also by *which IR states are visible to each pass at a given time.*

Figure 4 shows the two different execution styles in pseudocode form. The first code block corresponds to `module(function(A)), module(function(B))` or `module(function(A), function(B))`, and the second corresponds to `module(function(A,B))`.

```
for f in module.functions:
    run Pass A on f
for f in module.functions:
    run Pass B on f
```

Listing (1) All functions first run Pass A, then all functions run Pass B.

```
for f in module.functions:
    run Pass A on f
    run Pass B on f
```

Listing (2) Each function runs Pass A immediately followed by Pass B.

Fig. 4. Comparison of hierarchical pass execution strategies.

The analysis above shows how different nesting structures can create or miss optimization opportunities by altering the propagation of IR transformations. A search algorithm that only optimizes the pass sequence may settle on a solution with a sub-optimal structure. Therefore, the final stage of our framework is a dedicated **heuristic refinement process**. After the evolutionary search has identified a high-performance pass sequence, this stage performs targeted, localized adjustments to its nesting structure. The objective is to capture additional performance improvements by aligning the structure more effectively with inter-procedural dependencies.

4.4.2 Methodology: Optimizing Pipeline Partitions via a Genetic Algorithm. The analysis in Section 4.4 indicates that for a fixed pass sequence, the pipeline's hierarchical structure is a critical optimization variable. We formalize this structural optimization as a sequence partitioning problem and employ a Genetic Algorithm (GA) to search for effective solutions. The methodology follows a four-stage process: modeling the problem, analyzing the search space, pruning it based on domain-specific knowledge, and finally, applying an algorithmic search.

Problem Modeling: The Partitioning Space. Let $S = (p_1, p_2, \ldots, p_N)$ be a fixed pass sequence of length N. A hierarchical structure for this sequence can be uniquely defined by a **partitioning** $\mathcal{P}$, which divides the sequence into a series of m contiguous, non-overlapping blocks:

$$\mathcal{P} = \{B_1, B_2, \ldots, B_m\}$$

where $B_1 = (p_1, \ldots), B_2 = (\ldots), \ldots, B_m = (\ldots, p_N)$. Each block B_j corresponds to a single pass manager in the final pipeline.

This partitioning is determined by the set of $N-1$ boundaries between adjacent passes. At each boundary $i \in \{1, \ldots, N-1\}$, a binary decision is made: either to "split" the sequence (creating a new block) or to "join" it (extending the current block). The set of all possible partitions, $\mathbb{P}$, forms the search space. The size of this space is $|\mathbb{P}| = 2^{N-1}$, presenting a combinatorial optimization challenge. Our goal is to find an optimal partition $\mathcal{P}^*$ that maximizes a fitness function $f(\mathcal{P})$, which typically corresponds to the performance improvement over a baseline.

$$\mathcal{P}^* = \arg\max_{\mathcal{P} \in \mathbb{P}} f(\mathcal{P})$$

To intelligently navigate this large search space, we analyze the problem from its minimal unit: the structure governing two adjacent passes, (p_i, p_{i+1}). The performance implication of a "split" versus a "join" at boundary i is the core of our analysis. This reveals that the nature of the boundary

is the critical factor. We define $T(p)$ as the type of a pass (e.g., Function, Module). The key distinction is whether $T(p_i) = T(p_{i+1})$.

Rationale-Based Pruning of the Search Space. This minimal unit analysis provides a powerful, domain-specific insight that allows for massive pruning of the search space. We identify two distinct cases for any boundary i:

(1) **Heterogeneous Boundary ($T(p_i) \neq T(p_{i+1})$):** Here, the two passes must reside in different manager types. While both "joined" (e.g., *module(M, function(F))*) and "split" (e.g., *module(M), module(function(F))*) forms are syntactically possible, their execution order is semantically equivalent due to LLVM's execution model. As this choice does not offer an optimization trade-off, these boundaries are not true variables. We can prune them from the search, treating them as constants where a "split" is always applied.
(2) **Homogeneous Boundary ($T(p_i) = T(p_{i+1})$):** Here, the "join" (e.g., *function(A,B)*) and "split" (e.g., *function(A), function(B)*) decisions lead to fundamentally different execution orders and, consequently, can directly impact performance. These boundaries are therefore the true variables in our optimization problem.

This pruning strategy is the cornerstone of our refinement stage. It transforms the problem by identifying a small subset of boundaries, the **decision points**, which we denote by the set $D = \{i \mid T(p_i) = T(p_{i+1})\}$. The search is thus reduced from the full space $\mathbb{P}$ to a much smaller, more meaningful space of size $2^{|D|}$.

Algorithm: Genetic Search for Optimal Partitioning. We formalize the pruned problem and apply a Genetic Algorithm (GA) to explore the space of candidate solutions. A partitioning strategy is encoded as a binary chromosome of length $k = |D|$, where each bit corresponds to a decision point and represents either a "join" (*0*) or "split" (*1*).

The GA evolves a population of these chromosomes. The process begins with an initial population that includes the structure from the main evolutionary stage alongside other random variants. The fitness of each chromosome is determined by decoding it into a full pass pipeline string and measuring its performance on the target program. Using standard operators—selection, crossover, and mutation applied to the decision points—the algorithm iteratively updates the population. This approach allows systematic exploration of the reduced search space and identification of high-performing structural configurations.

5 Experiments

We evaluate our framework by addressing four Research Questions (RQs):
(RQ1) How does our framework's overall performance compare to LLVM -Oz?
(RQ2) How does it compare against other state-of-the-art auto-tuners?
(RQ3) Is the synergy-guided search beneficial?
(RQ4) What is the contribution of the final structural refinement stage?

5.1 Experimental Setup

Datasets. We use programs from the CompilerGym benchmark suite [13], partitioned into a training set of 19,603 programs for offline synergy mining and a disjoint test set of 335 programs for evaluation. The detailed composition is shown in Table 2.

Baselines for Comparison. We compare our framework against two categories of baselines:

- **LLVM -Oz:** A standard, highly-engineered optimization level.

Table 2. 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 [10]	149	32
	poj104-v1 [25]	7,000	0
	tensorflow-v0 [1]	415	90
Curated	cbench-v1 [15]	0	11
	mibench-v1 [17]	0	40
	chstone-v0 [19]	0	12
	npb-v0 [4]	0	121
Total	–	**19,603**	**335**

- **Auto-Tuning Methods:** We evaluate several representative auto-tuning methods originally designed for linear pass sequences: OpenTuner [2], GA [16], CFSAT [27], TPE [5], CompTuner [30], RIO [9], and BOCA [8].

Evaluation Metric. Our primary metric is the **Average OverOz (%) Improvement**, which measures the additional percentage reduction in LLVM IR instruction count relative to the `opt -Oz` baseline. It is calculated as:

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

where $\mathcal{P}_{\text{test}}$ denotes the set of test programs (with $|\mathcal{P}_{\text{test}}|$ being the number of test programs), $I_{\text{Oz}}(p)$ is the instruction count after `-Oz`, and $I_{\pi^*}(p)$ is the count from the evaluated method π^*. A higher value indicates a more effective optimization.

Environment. Our framework is built using LLVM 18.1.6. We use the `opt` tool to apply optimization pipelines and extract instruction counts. All experiments were conducted on a server with an AMD EPYC 7763 64-core processor. To ensure comparability across methods, all baseline auto-tuning methods operate on the same search space consisting of the transformation passes extracted from this LLVM version. This uniform search space ensures that all methods are evaluated under the same conditions.

5.2 RQ1 & RQ2: Overall Performance Comparison

To address our first two research questions, we conduct a comprehensive performance evaluation of our framework against both the industrial-strength LLVM `-Oz` baseline (RQ1) and a suite of state-of-the-art auto-tuning methods (RQ2).

Results. Table 3 presents the performance of all evaluated methods, measured by the Average OverOz (%) Improvement metric. **Answering RQ1**, our framework shows a notable and consistent performance improvement over the highly-tuned `opt -Oz` baseline, achieving an average additional instruction count reduction of **13.62%**. **Answering RQ2**, the comparison against other auto-tuners reveals a stratification of performance levels. Our framework achieves higher performance than

all other methods across the datasets. CFSAT follows as the second-best method with an average improvement of 8.27%. OpenTuner achieves a slightly positive average (0.28%) but with inconsistent results. In contrast, the remaining baselines (GA, TPE, and RIO) show average performance degradation, suggesting that they may be less suitable for this complex tuning task. Moreover, our framework achieves an average tuning time of only **5s**, slightly faster than CFSAT's 6s.

Table 3. Overall performance comparison based on Average OverOz (%) Improvement. A higher value is better. The best result for each dataset is highlighted in bold. Results for some baselines are pending (–).

Dataset	Ours	CFSAT	OpenTuner	GA	TPE	RIO	CompTuner	BOCA
cbench	**11.84**	8.07	-24.07	-34.65	-115.86	-81.64	-121.42	-83.98
blas	**6.86**	3.92	0.47	-18.19	-20.86	-19.04	-21.35	-18.20
opencv	**8.02**	7.42	7.82	-8.93	-13.83	-13.47	-13.85	-13.28
mibench	**7.99**	5.56	-0.71	-15.88	-146.54	-40.64	-161.58	-158.20
chstone	**21.03**	12.67	2.48	-42.27	-102.80	-68.95	-108.94	-87.47
tensorflow	**8.05**	7.64	6.90	-9.57	-14.36	-14.60	-15.13	-14.01
npb	**31.58**	12.62	9.06	11.14	-36.97	-5.47	-50.01	-28.53
Average	**13.62**	8.27	0.28	-16.91	-64.46	-34.83	-70.32	-42.10

Analysis of Results. The performance differences between the methods can be attributed to their fundamental design choices regarding the search space and optimization strategy, particularly in the context of the LLVM New PM.

The limited performance of most baseline auto-tuners (GA, TPE, etc.) can be explained by several factors. **(1)** Their search algorithms are designed primarily for linear sequences and do not natively respect the New PM's hierarchical grammar. As a result, many of the pass sequences they generate are syntactically invalid and rejected by the compiler, reducing the efficiency of their search. **(2)** These methods tend to focus on pass *selection* rather than precise *ordering*, thereby missing optimization opportunities that depend on sequence interactions. **(3)** The search space of passes includes many detrimental passes that can significantly increase instruction count. Without effective mechanisms to avoid these, GA and TPE often incorporate them, leading to performance degradation.

By contrast, the stronger performance of our framework and CFSAT reflects their shared use of knowledge-driven design. **(1)** Both approaches consider pass selection and ordering simultaneously. **(2)** Both employ an offline knowledge extraction phase, which helps identify and exclude harmful passes, substantially reducing the effective search space.

The performance advantage of our framework over CFSAT underscores the important role of native structural awareness. CFSAT's knowledge base implicitly contains structurally valid pass combinations, as any invalid combinations would have been rejected by the compiler during its offline mining. However, its search algorithm remains fundamentally linear and cannot explore beyond simple sequential compositions of these valid blocks. Our framework, being natively hierarchical, can explore the full, rich space of complex nested structures (e.g., *module(passA, function(passB, passC))*). This unique ability to discover and exploit structure-dependent optimization opportunities, which are invisible to CFSAT's linear search, helps to explain the observed performance differences.

5.3 RQ3 & RQ4: Ablation Studies on Framework Components

To understand the individual contributions of our key design choices, we conduct two distinct ablation studies. The first (RQ3) evaluates the impact of the Synergy Knowledge Graph, while the second (RQ4) quantifies the value added by the final structural refinement stage.

5.3.1 RQ3: The Impact of Synergy-Guided Search.

Experimental Design. To isolate the effect of our knowledge-guided approach, we create a baseline called **Random-GA**. This variant uses the exact same structure-aware Genetic Algorithm as our main framework but is knowledge-blind: its initial population and subsequent mutations are generated by randomly selecting passes from the full set of available passes. To specifically test the impact on search efficiency, we run both our guided framework and Random-GA with a reduced computational budget (smaller population and fewer generations).

Results and Analysis. Table 4 shows that our synergy-guided approach tends to achieve better performance than the Random-GA across all datasets. While our method still reaches a positive average improvement over `-Oz` even with a reduced budget, the Random-GA results in notable performance degradation. Specifically, the average improvement across all datasets is 0.76% for our synergy-guided approach versus -8.29% for Random-GA. This contrast suggests that heuristic guidance plays an important role in navigating such a vast and complex search space. Although Random-GA is structurally-aware, the large number of available passes leads it to often select detrimental ones. In comparison, the synergy-guided approach directs the search more effectively: by seeding the population with higher-quality building blocks and guiding mutations towards beneficial changes, it converges more rapidly to higher-performing regions. These findings provide evidence that the Synergy Knowledge Graph is a valuable component of our framework.

Table 4. Ablation study for RQ3, comparing Synergy-Guided GA with a Random GA under a reduced search budget. Performance is measured by Average OverOz (%) Improvement.

Dataset	Ours (Synergy-Guided)	Random-GA (Unguided)
cbench	-0.93	-13.67
mibench	-8.75	-18.82
chstone	-1.36	-15.65
blas	-2.39	-6.63
opencv	1.86	-3.58
tensorflow	-0.46	-3.70
npb	8.46	-3.96
Average	**0.76**	**-8.29**

5.3.2 RQ4: The Contribution of the Structural Refinement Stage.

Experimental Design. This ablation study quantifies the value added by the final structural refinement stage. We compare the performance of our framework's main search stage (**Main GA**, before refinement) against the final output of the **Full Framework**.

Results and Analysis. Table 5 shows that the refinement stage provides a modest performance improvement, increasing the average OverOz score from 13.08% to 13.62%. Notably, the gain is totally pronounced on the `npb` dataset (+3.83%) while no improvements at all on other dataset. This

behavior also proves that the structures of the pass sequence have different infuluences on different programs.

NPB (NAS Parallel Benchmarks) primarily includes programs related to scientific and parallel computing. These programs typically contain a large number of loop structures (for/while) with deep nesting and high iteration counts. The code size of NPB programs therefore are highly dependent on loop optimizations. The nested structure may enable loop passes and function passes to collaborate effectively, fully leveraging the benefits of loop optimizations with the intra-procedural information obtained from the function-level passes. Thus, the choice of nesting structures can lead to substantial divergence in the instruction counts reduction. For other datasets, they contain fewer loop structures or shallower loop nesting, limiting the overall impact of loop optimizations on instruction counts. The code logic may be more dispersed. They are primarily I/O-, branch-, or system call-intensive, with bottlenecks not residing within loops, leaving little room for additional optimization from nested structures. Thus, the nested structure optimization may have a limited effect on the final instruction counts for these datasets.

These results provide evidence that our approach demonstrates good performance in the cases where programs are parallel and loop-intensive, with the refinement stage acting as a specialized component that can capture additional performance gains on benchmarks where pipeline structure becomes a significant secondary factor.

Table 5. Ablation study for RQ4, quantifying the gain from the structural refinement stage. Performance is measured by Average OverOz (%) Improvement.

Dataset	Main GA	Full Framework	Gain from Refinement
cbench	11.84	11.84	+0.00%
blas	6.86	6.86	+0.00%
opencv	8.02	8.02	+0.00%
mibench	7.99	7.99	+0.00%
chstone	21.03	21.03	+0.00%
tensorflow	8.05	8.05	+0.00%
npb	27.75	**31.58**	**+3.83%**
Average	13.08	**13.62**	**+0.54%**

5.4 Case Study: Structure as a Disambiguation Mechanism

Our quantitative results have demonstrated the performance impact of structural choices. Here, we present a case study on `npb-v0_6.ll` that illustrates an additional role of the hierarchical pipeline: its ability to resolve pass ambiguity, which can be problematic for flat-sequence representations. This case highlights a second mechanism through which structure influences optimization, complementing the execution-order effect discussed earlier.

The Limitation of Flat Sequences: The Ambiguity Problem. A challenge for flat-sequence autotuners arises from passes with identical names but different functions depending on their scope. A representative example is `invalidate<all>`, a meta-pass that invalidates analyses at a specific IR level (Module, CGSCC, Function, or Loop). In a flat sequence, the string "`invalidate<all>`" does not specify scope, making its intended behavior unclear. As a result, a flat-sequence tuner may either omit such passes or adopt a fixed default interpretation, which might not be optimal.

Hierarchical Structure as the Solution. By representing pipelines hierarchically, our framework removes this ambiguity: the nesting explicitly determines the pass's scope. This can be seen in the two pipelines discovered for `npb-v0_6.ll`. While they contain the same sequence of pass names, their different structures yield different semantics and thus different performance outcomes. The initial pipeline from the main GA was a multi-stage, flattened sequence:

```
module(scc-oz-module-inliner),module(function(scalarize-masked-mem-intrin,jump-threading,correlated-propagation,reassociate,slp-vectorizer,vector-combine,correlated-propagation)),module(function(tsan,jump-threading,gvn-hoist,bounds-checking,instsimplify)),module(function(memcpyopt,adce)),module(function(loop(invalidate<all>))),module(strip)
```

Here, the nesting function(loop(...)) indicates that `invalidate<all>` is executed at the Loop level. The refined pipeline, however, converged to a monolithic, nested structure:

```
module(scc-oz-module-inliner,function(scalarize-masked-mem-intrin,jump-threading,correlated-propagation,reassociate,slp-vectorizer,vector-combine,correlated-propagation,function(tsan,jump-threading,gvn-hoist,bounds-checking,instsimplify),memcpyopt,adce),invalidate<all>,strip)
```

In this case, since `invalidate<all>` is a direct child of the module manager, its scope is defined as the Module level.

Analysis: A Different Form of Structural Impact. This example illustrates a type of structure effect distinct from the one shown in our motivation (Table 1). In the earlier case, the effect came from altering the execution order of a fixed set of passes while their semantics remained unchanged. Here, however, the pass sequence is identical, yet the structural change redefines the semantics of `invalidate<all>`: one pipeline applies repeated loop-level invalidations, while the other performs a single module-level invalidation. These strategies impact the compiler's analysis infrastructure in different ways, resulting in the observed 3.31% performance gap. For this program, the refinement stage discovered that the module-level invalidation strategy was more effective.

6 Limitations

Despite the effectiveness of our framework, there are several limitations. 1) **Scope of Synergy Consideration:** Our current approach only accounts for intra-level and inter-level pass synergies, ignoring potential interactions between subtrees. Certain passes may form a high-quality combination that performs well across many programs. Treating such combinations as a subtree and exploring its synergy with other subtrees could further reduce the search space and potentially improve performance. 2) **Refinement Adaptivity:** While the pass sequence itself dominates optimization performance in most cases, the nesting structure can still affect results for a small subset of programs. Our current refinement stage is not adaptive and cannot automatically identify which programs or pass sequences may benefit from structural adjustments. 3) **Evaluation Metric:** We primarily use instruction count as the optimization objective. While widely adopted, it does not always reflect actual runtime performance or other metrics such as energy efficiency. Therefore, our pipelines may not be globally optimal under all practical performance criteria.

Addressing these limitations, such as incorporating subtree-level synergies, developing adaptive refinement strategies, and considering more comprehensive evaluation metrics, represents promising directions for future work.

7 Related Work

The automation of compiler optimization, particularly pass selection and phase ordering, has been a central theme of systems research for decades. Approaches have evolved through several distinct

paradigms, each employing increasingly sophisticated techniques to navigate the vast optimization search space. We summarize these prior paradigms in Table 6.

Table 6. Comparative Summary of Prior Auto-Tuning Paradigms.

Paradigm	Core Methodology	Representation
Iterative Compilation [2, 6–9, 16, 21, 24, 27, 30]	Empirically searches the optimization space using heuristics (e.g., GA, Simulated Annealing) on a per-program basis.	Linear Sequence
ML / Reinforcement Learning [3, 13, 14, 18, 22, 23, 28, 29]	Learns a predictive policy or cost model to map program features to effective optimizations, avoiding exhaustive search.	Linear Sequence
Large Language Models [11, 12, 20, 26]	Fine-tunes a foundation model to infer optimization strategies directly from source code, capturing deep semantic patterns.	Linear Sequence

As summarized in the table, a critical thread across all prior paradigms—from heuristic search to machine learning and LLMs—is their foundational reliance on a **linear representation of the optimization pipeline**. This modeling choice, while perfectly suitable for legacy compiler infrastructures, is fundamentally incompatible with the hierarchical and grammatically-constrained architecture of the LLVM New Pass Manager. The core issue is that a linear representation lacks the expressive power to describe nested structures. Consequently, these methods are incapable of guaranteeing the generation of syntactically valid pipelines, which prevents them from being directly and reliably applied to the New PM's constrained search space. Furthermore, they are inherently "structurally-blind," meaning they cannot reason about or explore the performance implications of different valid hierarchical arrangements.

Our work addresses this **gap**. We are the first to propose an auto-tuning framework designed natively for the challenges and opportunities of modern, hierarchical compilers. By introducing a formal grammar and a forest data structure, we provide a robust mechanism for representing and manipulating these complex pipelines. Our knowledge-guided, co-evolutionary genetic algorithm treats the pipeline structure as a first-class entity. This ensures that every candidate solution is valid by construction and enables a holistic search that optimizes both the pass sequence and its hierarchical structure simultaneously, unlocking performance opportunities that are inaccessible to previous-generation, linear auto-tuners.

8 Conclusion

In this work, we presented a comprehensive framework for auto-tuning compiler passes in the LLVM New Pass Manager, addressing both the syntactic and structural challenges introduced by its hierarchical design. We first formalized the space of valid optimization pipelines using a context-free grammar and a forest-based data structure, providing a foundation for algorithmic manipulation

that guarantees syntactic correctness. Building upon this representation, we introduced a structure-aware Genetic Algorithm guided by an offline-mined Synergy Knowledge Graph, enabling the efficient discovery of high-performance pass sequences. Furthermore, we designed an optional heuristic refinement stage to capture subtle performance differences caused by varying valid nesting structures. Our evaluation on seven benchmark datasets using LLVM 18.1.6 demonstrates that the proposed framework can achieve an average of 13.62% additional reduction in instruction count compared to the standard `opt -Oz` optimization level, outperforming existing auto-tuning approaches. These results validate that integrating structural awareness, knowledge-guided search, and targeted refinement provides tangible benefits over both traditional linear-sequence auto-tuners and standard compiler optimization levels.

In summary, this work highlights the importance of considering hierarchical structure as a first-class factor in compiler auto-tuning and provides a general methodology for systematically exploring and exploiting this complex, constrained search space.

9 Data Availability

A replication package, including code and data used in our experiments, is available at https://github.com/Mind4Compiler/NPMCT/ 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] 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] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

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

[6] 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*.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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

[13] 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.

[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] 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.

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

[21] Michael R Jantz and Prasad A Kulkarni. 2013. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 1–10.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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).

[27] 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.

[28] 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.

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

[30] 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.