# $\mathcal{BO}3$ (Beyond-O3): Workload-Specific Compilation Pipelines for Multi-Level Compilers

by

Megan Kuo

A Thesis
Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2026 by the
Graduate Supervisory Committee

Aviral Shrivastava, Chair
Aman Arora
Jianping Zeng

ARIZONA STATE UNIVERSITY
May 2026

# ABSTRACT

Deploying diverse machine learning workloads across hardware targets increasingly relies on multi-level compiler stacks built on infrastructures such as MLIR. While production compilers typically rely on fixed, expert-designed optimization pipelines, hierarchical progressive lowering makes pass ordering especially consequential. Reordering passes can alter dispatch partitioning and backend routing, effectively changing the downstream tuning space rather than merely improving code quality within a fixed one. This leaves substantial workload-dependent performance untapped.

We first show empirically that performance varies substantially across compilation pipelines. No single fixed pipeline generalizes across models and tensor shapes. We then investigate *why* progressive lowering amplifies phase-ordering effects compared to single-level compilers. Through structural and backend-oriented diagnostics, including dispatch topology, code-generation strategy-mix, and instruction proxy metrics, we identify *tuning-space regime shifts*: discrete changes in downstream compilation structure induced by upstream pass reordering.

These findings inform the design of $\mathcal{BO}3$ (Beyond O3), a legality-aware three-stage reinforcement learning (RL) framework for automated pass ordering in IREE. $\mathcal{BO}3$ combines backend-configuration screening with policy learning over preprocessing-stage pass sequences, using graph-based program observations and dynamic action masking to respect structural legality constraints.

Evaluated on 9 workload instances from 7 model families, $\mathcal{BO}3$ discovers workload-specific pass sequences that outperform the default IREE `-O3` pipeline, achieving a geometric-mean speedup of 5.66×. Ablation studies show that graph-structured observations improve solution quality, while dynamic legality masking improves rollout robustness.

ACKNOWLEDGMENTS

This thesis benefited from the guidance and support of many people. I am profoundly grateful to my advisor, Professor Aviral Shrivastava. It is no exaggeration to say that his words of encouragement and support carried me through the moments when I was ready to step away from this project. His mentorship has been a true gift, and I am honored to have worked under his guidance.

I would also like to extend my sincere thanks to my supervisory committee members, Aman Arora and Jianping Zeng, for their valuable time, technical suggestions, and support throughout this process. To my labmates and friends, thank you for your constant encouragement and the rich discussions that made the demanding days of research so much more rewarding.

I owe a profound debt of gratitude to my parents for their unconditional mental and financial support. Your sacrifices, endless patience, and belief in me provided the foundation I needed to pursue and complete my graduate studies.

Finally, and most importantly, I give my ultimate gratitude to God. Through every challenge, my faith has been a light in the dark, guiding my path and providing the spiritual strength and motivation necessary to finish this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

## Research Background

Machine learning systems are now deployed across a broad range of application domains, including computer vision, natural language processing, speech, recommendation, and scientific computing. As these workloads are mapped onto increasingly diverse hardware platforms, compiler support has become a critical part of the deployment stack. Unlike traditional scalar or control-dominant programs, machine learning programs are dominated by tensor computations, structured linear algebra, layout-sensitive transformations, and hardware-dependent execution behavior. Consequently, effective compilation for machine learning workloads requires bridging a large semantic gap between frontend program representations and backend executable code.

Conventional single-level intermediate representations, such as LLVM IR, are highly effective for low-level optimization and code generation, but they are not designed to naturally express high-level tensor semantics, structured operators, or transformation intent specific to machine learning workloads. To address this limitation, modern compiler infrastructures increasingly adopt multi-level intermediate representations (MLIR), where programs are represented and transformed across several abstraction levels rather than flattened prematurely into a single low-level form.

This design has enabled a broad ecosystem of extensible compiler stacks spanning machine learning and hardware domains, including systems such as IREE, Torch-MLIR, and CIRCT (LLVM Project, 2026b; Liu *et al.*, 2022; LLVM Project, 2026a;

CIRCT Team, 2026). In these systems, compilation proceeds through a sequence of transformations across dialects and abstraction levels, gradually lowering high-level tensor operations into increasingly concrete, target-aware representations. This compilation style, commonly referred to as *progressive lowering*, improves modularity, extensibility, and retargetability. At the same time, it makes optimization decisions at early stages more consequential, because those decisions influence the program structure that later stages will observe and optimize.

This property is especially important in MLIR-based machine-learning compilers such as IREE. In such compilers, lowering is not merely a translation from one representation to another; it also determines what structural information is preserved, exposed, or discarded along the way. Once a program has been lowered past a certain representation boundary, later stages must optimize under the structure that remains. As a result, the effectiveness of downstream code generation depends not only on backend decisions, but also on how the program was transformed before dispatches were formed and lowered.

## Terminology

To reduce ambiguity, this thesis uses the following terminology throughout. A *pass* denotes a single IR transformation, and *phase ordering* refers to the problem of selecting and ordering such transformations into a pipeline. *Dispatch partitioning* refers to the process by which higher-level computation is partitioned into executable regions, called *dispatches*, that serve as the basic units of subsequent backend lowering and code generation. *Backend knobs* denote discrete backend code-generation parameters or flags, such as tiling, unrolling, masking, or vectorization choices. A *knob anchor* is a combination of fixed backend knobs used while searching over upstream pass sequences. Finally, a *tuning-space regime shift* denotes a discrete downstream change

in dispatch structure or backend routing induced by upstream pass reordering.

## Problem Statement

Although progressive lowering provides important benefits in compiler structure and expressiveness, it also introduces a difficult optimization problem. In MLIR-based compilation, transformations applied at early stages can substantially affect the optimization opportunities available later. A pass applied earlier or later in the lowering flow can change dispatch formation, layout materialization, fusion opportunities, and even the backend lowering strategy ultimately selected for a workload. In this setting, pass ordering is not merely a local matter of improving IR quality; it can alter the downstream optimization problem itself.

In production compilers, this challenge is commonly managed through fixed, expert-designed optimization pipelines, such as a default `-O3`-style pipeline.[1] Such pipelines are attractive because they are deterministic, maintainable, and easy to deploy. However, in MLIR-based machine-learning compilation, a single static pipeline is often fragile. A pass sequence that is effective for one workload may be clearly suboptimal for another. Even within the same model family, changes in tensor shape can shift which pass ordering is most profitable.

This difficulty arises because progressive lowering is hierarchical. Early transformations do not simply improve or degrade code quality within a fixed search space. Instead, they can reshape the structure seen by later compilation stages. For example, an upstream pass may change how computation is partitioned into dispatches, which backend strategy a dispatch is routed into, or how much layout materialization overhead must later be introduced. These downstream structural changes are central to the phase-ordering problem studied in this thesis.

[1] In this thesis, the default IREE optimization pipeline is used as the baseline in all evaluations.

Existing autotuning approaches are not well matched to this setting for two main reasons. First, traditional parameter tuners Ansel *et al.* (2014); Chen *et al.* (2018); Zheng *et al.* (2020); Mullapudi *et al.* (2016) generally assume that the program structure is largely fixed and that the primary search space lies in backend schedules or code-generation parameters. While effective in those settings, they are not designed to handle discrete structural changes induced by pass reordering in multi-level compilers. Second, greedy heuristics based on immediate IR-local signals are often unreliable here, because the true effect of an early transformation may emerge only after several later lowering stages, through changes in dispatch formation, layout obligations, and backend routing Agakov *et al.* (2006); Kulkarni and Cavazos (2012).

This thesis studies the pass-ordering problem in MLIR-based progressive lowering, with a particular focus on the IREE compiler. The central view adopted in this work is that pass ordering is difficult not only because optimization effects are delayed, but because upstream pass reordering can induce discrete downstream changes in the effective tuning space. Rather than merely tuning within a fixed optimization regime, pass ordering can change the regime itself.

Accordingly, the core research question of this thesis is the following:

> How can compiler pass ordering be automated in MLIR progressive lowering when legality constraints evolve across compilation, and upstream pass choices can induce workload-dependent downstream regime shifts?

### Challenges in Progressive Lowering

Several characteristics of progressive lowering make this problem fundamentally more difficult than classical formulations of compiler phase ordering.

First, legality is state-dependent. In a multi-level compiler, the set of valid transformations changes as the IR evolves. A pass that is legal and useful at one point in

the lowering flow may become invalid, ineffective, or counterproductive later. Therefore, an automated pass-ordering method must reason not only about profitability, but also about the changing legality structure of the search space.

Second, optimization effects are long-horizon. Early transformations can influence how computation is partitioned into dispatches, which layouts are materialized, and which backend lowering strategies become applicable. These effects may not be visible through immediate local simplification signals. As a result, short-horizon heuristics can miss transformations whose value appears only after later lowering stages.

Third, workload sensitivity is high. Modern machine learning workloads differ substantially in operator composition, tensor shapes, dataflow structure, and memory behavior. Convolution-heavy vision models, transformer-based models, encoder-decoder architectures, and detection workloads do not expose the same profitable transformation patterns. Moreover, even within a fixed architecture family, changes in input shape can change divisibility, dispatch structure, and layout behavior, thereby altering the best pass ordering.

Fourth, the search space is constrained but still combinatorially large. Real compiler pipelines expose multiple candidate passes, repeated applications, and order-dependent interactions. Exhaustive search is infeasible, while manual search requires significant compiler expertise and often produces brittle, workload-specific pipelines. In practice, this leaves substantial performance headroom unexplored.

Taken together, these characteristics show that pass ordering in MLIR progressive lowering cannot be treated as a simple extension of traditional phase-ordering formulations. An effective approach must reason over structured compiler state, respect evolving legality constraints, and optimize for downstream end-to-end effects rather than only immediate local changes.

## Thesis Overview

This thesis presents $\mathcal{BO}3$, a legality-aware 3-stage RL framework for automated pass ordering in IREE. The framework is instantiated in the IREE compiler at the preprocessing stage, where transformations operate on whole-program tensor IR before dispatch boundaries are introduced. This placement is important because preprocessing-stage transformations can reshape the program structure that downstream dispatch creation and backend code generation will later observe.

The design of $\mathcal{BO}3$ is motivated by the observation that, under progressive lowering, pass ordering often changes the downstream tuning space rather than merely tuning within a fixed one. To address this, $\mathcal{BO}3$ decomposes the optimization problem into two levels. The first level performs screening over backend code-generation configurations to identify a small set of good *knob anchors*. Each retained anchor fixes a set of backend knobs against which upstream pass search is performed. The second level then trains a legality-aware RL agent to search over preprocessing-stage pass sequences conditioned on the selected anchor.

To represent compiler state, the framework encodes the current IR as an SSA def-use graph together with compact global and contextual features. This graph-based representation allows the policy to reason over operator structure, producer-consumer relationships, and non-local dependencies that affect pass profitability. To avoid invalid exploration, the action space is dynamically filtered through legality masking, so that the policy only considers transformations that are valid for the current compiler state.

The thesis first develops the motivation for pass ordering by showing that fixed pipelines fail to generalize reliably across workloads, tensor shapes, and stage placements. It then presents the $\mathcal{BO}3$ framework, describes the RL formulation and com-

piler integration, and evaluates the method on a set of workload instances drawn from multiple model families. The experimental results show that $\mathcal{BO}3$ discovers workload-specific pass sequences that outperform the default IREE `-O3` pipeline, achieving substantial end-to-end latency improvement on every instances.

**Contributions.** This thesis makes two main contributions:

- We investigate why pass ordering has a substantial impact on the final performance in multi-level compilers (e.g., IREE) as compared to single-level compilation flow (e.g., LLVM-style pipelines). We identify *tuning-space regime shifts* as an important source of phase-ordering sensitivity in MLIR/IREE progressive lowering.

- We present $\mathcal{BO}3$ (Beyond -O3): a legality-aware three-stage RL framework that combines backend-configuration screening with graph-based pass search under dynamic legality masking and terminal rewards to discover good workload-specific compilation paths.

We show that $\mathcal{BO}3$ consistently outperforms the default IREE `-O3` pipeline across 9 workload instances from 7 model families, achieving a 5.66× geometric-mean speedup using only bounded post-training rollouts from the learned policy.

# Chapter 2

# BACKGROUND AND RELATED WORK

## MLIR and Progressive Lowering

The Multi-Level Intermediate Representation (MLIR) was introduced to address a central limitation of conventional compiler infrastructures: a single intermediate representation is often insufficient to naturally express the range of abstractions required by modern domain-specific compilation. In machine-learning compilation in particular, programs are more naturally described in terms of tensor operators, structured linear algebra, layout-sensitive transformations, and staged lowering decisions than as a flat low-level instruction stream. If these abstractions are lowered too early into a single low-level representation, important structural information may be obscured before the compiler has had an opportunity to exploit it effectively.

MLIR addresses this issue through a multi-level design. Instead of forcing all transformations into one universal representation, MLIR organizes compilation through multiple dialects, each of which captures a particular abstraction level or domain-specific view of computation. This allows transformations to be applied at the level where they are most meaningful. For example, high-level tensor rewrites can be applied while semantic structure is still explicit, whereas lower-level code generation decisions can be deferred until the program has been transformed into a more target-aware form.

A central consequence of this design is the widespread use of *progressive lowering*. Under progressive lowering, compilation proceeds through a sequence of representations rather than through a single abrupt lowering step. High-level operations are

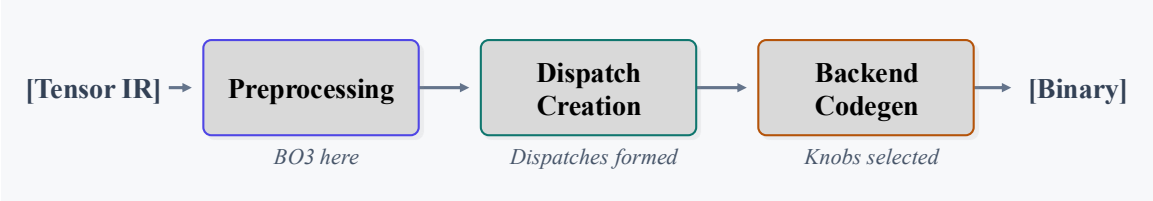



Figure 2.1: IREE's three-stage compilation pipeline. Preprocessing operates on whole-program tensor IR before dispatch boundaries exist; dispatch creation partitions the IR into self-contained executable regions; backend code generation independently lowers each dispatch under a target-specific configuration. BO3 targets the preprocessing stage (purple outline), where transformations can reshape the program structure observed by the two downstream stages.

gradually rewritten into increasingly concrete forms until they reach backend-specific representations. This style improves modularity and retargetability, but it also makes optimization decisions more path-dependent. Each lowering stage both exposes some lower-level structure and discards some higher-level information. Once a representation boundary has been crossed, later stages must optimize under the structure that remains.

For this reason, progressive lowering changes the nature of the optimization problem itself. A transformation applied early in the pipeline can alter what later stages are able to recognize, partition, fuse, materialize, or route. Accordingly, the pass-ordering problem in progressive-lowering compilers must be understood not only as a search over transformation sequences, but also as a search over sequences that shape the downstream optimization context.

### IREE Compilation Flow

IREE provides a concrete and practically relevant setting for studying this problem. It is an MLIR-based compiler and runtime stack for deploying machine-learning workloads across multiple hardware targets, including CPUs and GPUs. Its compilation flow is explicitly organized around progressive lowering, making stage-wise structure exposure and backend specialization first-class concerns.

In this thesis, the IREE compilation flow is viewed in terms of three major stages, as shown in Figure 2.1.

#### *Preprocessing Stage*

The preprocessing stage operates on whole-program tensor IR before any dispatch boundaries have been introduced. At this point, the compiler still has access to relatively high-level structural information. Transformations in this stage include operations such as layout conversion, padding-related rewrites, attention-related rewrites, canonicalization, and operator reformulation. Because these transformations occur before dispatch creation, they can reshape the program structure that downstream stages will observe.

This stage is particularly important because it is where upstream pass ordering has the greatest leverage. A change made here does not merely affect a local region of IR. It can change how later stages partition the program, what forms become recognizable to backend-specific experts, and how much layout materialization overhead is later introduced.

### *Dispatch-Creation Stage*

In the dispatch-creation stage, IREE partitions the program into executable regions called *dispatches*. A dispatch is the basic unit that later backend code generation will lower and optimize independently. The number of dispatches, their internal composition, and the dataflow relationships between them strongly influence the downstream optimization opportunity.

This stage is important because dispatch partitioning defines the structural interface between earlier tensor-level rewriting and later backend lowering. Once dispatch boundaries are formed, subsequent optimization must proceed within those boundaries. Therefore, upstream transformations that change dispatch partitioning do not merely improve code quality inside a fixed structure; they can change the structure being optimized downstream.

### *Backend Code Generation*

After dispatches are formed, each dispatch is lowered independently using a backend-specific code-generation pipeline. At this stage, IREE selects target-specific lowering behavior based on dispatch structure together with backend code-generation parameters. In this thesis, these discrete backend settings are referred to as *knobs*.

The key point is that preprocessing and dispatch creation determine the structure presented to this stage. Backend code generation operates on the dispatches and layouts produced by earlier compilation choices.

### Representation Choice and Progressive-Lowering Sensitivity

A distinctive property of MLIR-based compilation is that semantically equivalent computations may be represented in multiple forms during lowering. These repre-

sentation choices matter because downstream analyses and backend routing decisions often depend not only on semantics, but also on how those semantics are encoded in the IR.

Within IREE, one representative example is the distinction between named `linalg` operations and equivalent `linalg.generic` forms. A computation may remain in a named form that is easy for later stages to pattern-match, or it may be generalized into a more explicit form that exposes indexing maps, iterator structure, and scalar-body semantics that is easier for fusion and computation simplification. Although the computation itself is unchanged, the downstream consequences can differ substantially depending on when this conversion occurs.

This asymmetry is important. A named operation can be generalized at any time, but once subsequent rewrites have altered the IR, recovering the original named structure may no longer be straightforward. As a result, the same pass can have different downstream effects depending on where it is inserted in the lowering flow. Applied at one stage, it may expose structure that later canonicalization can exploit. Applied at another, it may instead remove information that downstream routing relies on. This stage dependence is one of the central motivations for the regime-shift analysis developed later in the thesis.

### IREE CPU Lowering Strategies as Regime Proxies

To interpret the downstream consequences of pass reordering, this thesis uses several concrete IREE CPU lowering behaviors as proxies for different backend regimes. Among these, two dispatch-lowering strategies appear prominently throughout the analysis: *Double-tiling* and *MMT4D*.

*Double-tiling*

Double-tiling is a general-purpose lowering path. It operates on the original tensor layout and emphasizes multi-level tiling and vectorization to improve locality and execution efficiency across a broad range of contraction-like workloads. Because it does not rely on an explicit tiled global layout, it is broadly applicable. However, this generality can come at a cost. The generated inner kernels may still require on-the-fly data rearrangement such as transposition, unpacking, or lane shuffling, to match the access patterns expected by the target hardware. For example, the kernel may read the right-hand-side tile in the layout given by memory, which is often row by row. But the hardware may want the same data in a different form, such as grouped by columns or by vector lanes. So the kernel must reorder the data after loading it. This reordering adds extra instructions and register usage without doing useful multiply-accumulate work. It is called on-the-fly because the conversion happens while the kernel is running, not ahead of time in memory.

*MMT4D*

MMT4D is a more specialized path for matmul-like computation. In contrast to Double-tiling strategy, MMT4D moves more of the data reorganization into memory. The operands are stored in a tiled layout that is closer to what the inner kernel wants, so the kernel does less rearrangement while it runs. This reduces on-the-fly transpose or shuffle work in the critical loop, at the cost of materializing a more specialized layout.

### *Pack/Unpack Tradeoffs and Encoding Dispatches*

The main tradeoff in MMT4D-tiling is that the tiled layout must be materialized. This often introduces pack and unpack operations. If these layout transformations can be fused or amortized effectively, the specialized path can be highly beneficial. If not, layout-conversion overhead can dominate runtime.

Related to this, IREE may introduce additional *encoding dispatches* whose role is to materialize layout conversions needed by tiled data formats. In this thesis, the presence, absence, or relative frequency of such dispatches is treated as a useful structural signal. Together with the observed strategy mix between Double-tiling and MMT4D, these effects provide a concrete way to interpret routing-level changes induced by upstream pass ordering.

## Compiler Phase Ordering

Compiler phase ordering has long been recognized as a difficult optimization problem. Modern compilers expose many transformations whose effects are order-dependent, non-commutative, and difficult to predict in isolation. A pass that is profitable in one context may be neutral or harmful in another, and its value may depend strongly on which transformations preceded it.

In classical settings, the phase-ordering problem is often framed over a relatively stable IR. The main challenge is combinatorial: many pass sequences exist, exhaustive search is infeasible, and within-pass interactions/dependencies are difficult to model. Production compilers therefore commonly rely on fixed expert-designed pipelines such as `-O2` or `-O3`, which trade workload-specific optimality for robustness, determinism, and maintainability.

The difficulty becomes more pronounced under progressive lowering. In this set-

ting, the representation being optimized is not stable throughout compilation. A pass can affect which operations survive into later stages, how data layout propagates, how computation is partitioned into dispatches, and which backend lowering strategies remain applicable. Thus, phase ordering in progressive-lowering compilers is not simply a search over different sequences acting on a fixed object. It is a search over sequences that can reshape the downstream optimization landscape itself.

This distinction is central to the present thesis. The problem is not only to find a locally profitable transformation order, but to do so in a setting where legality evolves with the state and where downstream structural consequences can dominate immediate local simplification effects.

### Autotuning and Structural Search for Tensor Programs

A large body of prior work has studied autotuning for tensor programs and machine-learning compilation, especially in systems that expose explicit schedule spaces or backend parameter spaces. Frameworks such as OpenTuner provide reusable infrastructure for high-dimensional program-configuration search. In the machine-learning domain, TVM introduced operator-level schedule search over tensor programs, and later work used learned cost models to accelerate exploration. Ansor extended this direction with hierarchical search and automatic schedule generation for broader operator classes. Related ideas also appear in Halide auto-scheduling and in graph-level substitution search systems such as TASO.

These works have demonstrated that automated search can be highly effective when the main degrees of freedom lie in backend schedule choices, such as tiling, unrolling, vectorization, or memory placement. In those settings, the program structure being optimized is often treated as fixed or mostly fixed, and the search problem is to identify a good parameterization of that structure.

The optimization problem studied in this thesis differs in an important way. The focus here is not primarily on schedule tuning within a fixed structure, but on *pass ordering under progressive lowering*. In this setting, upstream transformations can alter dispatch structure, layout materialization, and backend routing. The issue is therefore not only how to tune a given downstream regime, but how earlier compilation choices can change which downstream regime exists at all.

For this reason, existing autotuning systems are highly relevant as motivation for performance-driven search under bounded budgets, but they do not directly solve the problem considered here. Their success highlights the value of automated search, while also making clear the need for methods that can reason about structural compiler state rather than only backend parameters.

### Reinforcement Learning for Compiler Optimization

Reinforcement learning (RL) is a natural fit for compiler optimization when decisions have delayed consequences. In a compiler pipeline, the choice of one transformation often affects the utility of future transformations, so the optimization problem is inherently sequential rather than purely one-step.

Prior work has explored RL for compiler phase ordering and related optimization decisions. AutoPhase used RL in LLVM to select pass sequences for High-Level Synthesis programs. NeuroVectorizer applied deep reinforcement learning in LLVM to choose loop vectorization factors and interleaving decisions. MLGO showed that learned optimization policies can replace hand-written heuristics in LLVM and be integrated in a deployment-oriented way, with initial case studies on inlining and later production use extending to register allocation. CompilerGym further provided a standardized experimental environment for compiler optimization research.

Recent work has also begun to study RL in MLIR settings. MLIR RL introduces

an RL environment for automatic code optimization over MLIR `linalg` programs, focusing primarily on loop-level transformation decisions. In its evaluation on full deep-learning models, however, it remains behind the PyTorch compiler; for example, using the paper's reported speedups over an unoptimized MLIR baseline, MLIR RL is about 16.2× slower on ResNet-18, 4.1× slower on MobileNetV2, and 6.0× slower on VGG. The paper attributes much of this gap to the absence of architecture-specialized kernels and stronger Conv2D/Matmul lowering in the current action space.

Unlike most prior RL-for-compilers settings, this thesis targets pass ordering in IREE's progressive-lowering pipeline, where an early pass can change later dispatch partitioning, backend selection, and layout/materialization behavior. The key difficulty is therefore not only choosing profitable actions, but reasoning about structural regime shifts under evolving legality constraints.

### Research Gap

Existing compiler autotuning methods do not fully match the optimization problem induced by MLIR progressive lowering.

Classical phase-ordering research establishes that pass ordering is difficult because transformations interact in order-dependent ways and the search space is combinatorial. However, under progressive lowering, these interactions become more consequential. A pass can affect not only local IR quality, but also downstream dispatch partitioning, layout propagation, and backend routing, thereby changing the optimization regime seen by later stages.

Tensor-program autotuning systems show that automated search can be highly effective when optimizing schedule parameters or backend knobs under a fixed or mostly fixed program structure. Reinforcement-learning-based compiler optimization further shows that learned sequential policies can outperform static strategies when

optimization effects are delayed. Yet neither line of work directly addresses the combined setting induced by progressive-lowering compilers such as IREE.

What remains insufficiently addressed is the setting in which:

1. the search space is *structural*, not merely parametric, because pass ordering can alter dispatch partitioning, layout flow, and backend routing;

2. legality is *state-dependent and evolving*, so the feasible action set changes as lowering proceeds;

3. optimization quality depends on *downstream regime shifts*, meaning that a profitable sequence may be one that changes what later stages are able to optimize, rather than one that simply improves a local proxy in the current IR.

This leaves a clear gap for MLIR/IREE compilation. Existing schedule autotuners largely assume a downstream object that is already fixed enough to tune, while prior phase-ordering formulations generally do not model multi-stage legality together with regime-sensitive structural change. The thesis addresses this gap by studying automated pass ordering in a progressive-lowering setting where the optimization target is itself shaped by the compilation trajectory.

# Chapter 3

# MOTIVATION: WHY FIXED PIPELINES FAIL TO GENERALIZE

When a pass pipeline delivers a large speedup, a natural interpretation is that it simply improves local optimization quality within an otherwise fixed compilation flow. In IREE, this interpretation is incomplete. Under progressive lowering, each stage both exposes lower-level structure and discards higher-level information. These choices are not neutral: an early transformation can change what later stages are able to fuse, partition, materialize, and route. As a result, pass ordering in IREE does not merely search for a better point within a fixed downstream optimization space. It can instead change the downstream optimization problem itself.

Figure 3.1 shows that across 16 workloads, the best-found pipeline can achieve a geometric-mean speedup of 5× over the default `-O3`. These best-found pipelines were obtained by enumerating approximately 150 available passes across IREE's compilation stages and searching pass combinations per model over 48 hours.

This chapter develops the central empirical motivation of the thesis. The key observation is that, under progressive lowering, upstream pass reordering can induce discrete downstream changes in dispatch structure, layout behavior, and backend routing. This thesis refers to these changes as *tuning-space regime shifts*. Under such shifts, the compiler is not simply exploring better settings within the same downstream space; it is entering a different downstream space altogether.

This view helps explain why fixed pipelines transfer poorly. If pass ordering only changed local IR quality, then one would expect a single expert-designed pipeline to generalize reasonably across workloads, and one would also expect local profitability proxies to provide useful guidance. The evidence in this chapter shows otherwise.

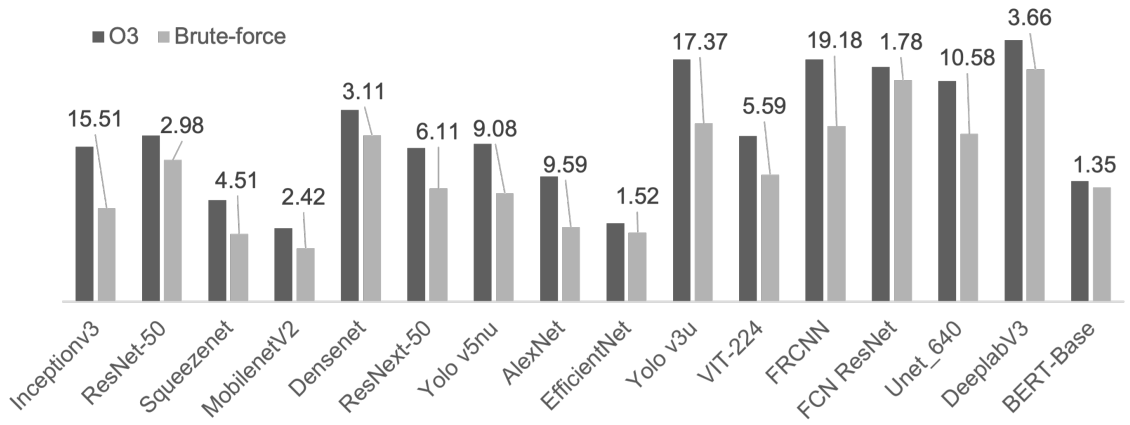



Figure 3.1: Performance headroom from workload-specific pass ordering over the default IREE `-O3` pipeline. Each bar pair shows the `-O3` baseline runtime (gray) and the best-found pipeline runtime (white) for a given model; annotations show the speedup ratio. The gap demonstrates that fixed pipelines leave substantial workload-dependent performance untapped.

This chapter presents three complementary observations: (i) different models need different pipelines, (ii) different tensor shapes need different pipelines, and (iii) the same pass, applied at different stages, can change the downstream problem in qualitatively different ways.

### Different Models Need Different Pipelines

Figure 3.2 shows that pass pipelines are not uniformly good or bad across models. Instead, the ranking of candidate pipelines changes substantially across workloads. Each column corresponds to a model, each row to a candidate pass pipeline, and each cell reports speedup over the default `-O3` baseline. The black outline marks the best-performing pipeline for each model.

The key observation is that the identity of the best pipeline changes across models.

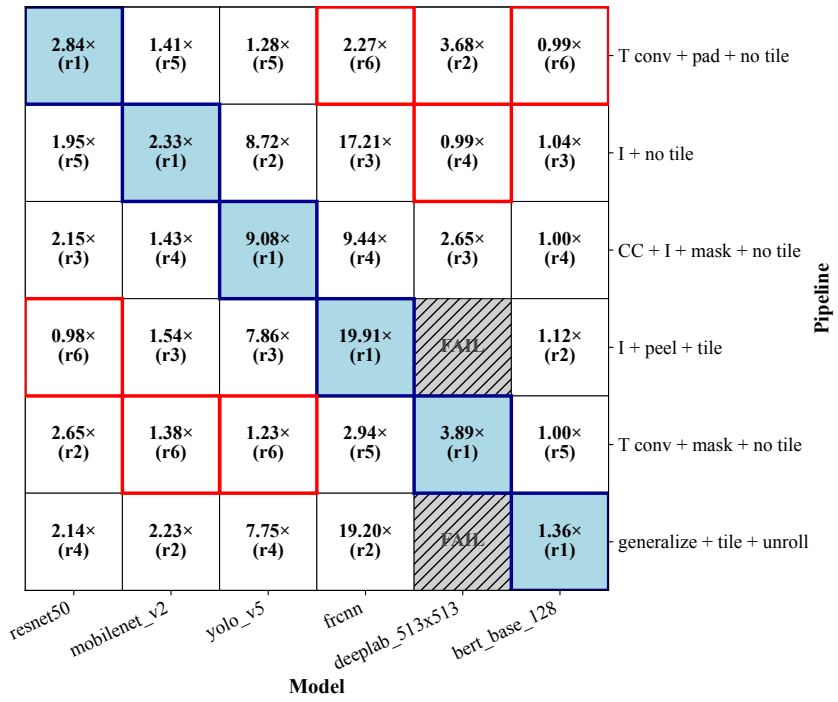



Figure 3.2: Cross-model heterogeneity of pass pipelines. Each cell reports speedup over the default `-O3` baseline, and the black outline marks the best-performing pipeline for each workload. The best pipeline changes across models, and some sequences are invalid on specific workloads (marked FAIL).

For example, `I + peel + tile` is the best pipeline for `frcnn` at 19.91×, yet it drops to near-baseline on `resnet50` (0.98×) and fails on `deeplab_513x513`. Conversely, `T + mask` is best for `deeplab_513x513` at 3.89×, but falls to sixth place on `yolo_v5` at only 1.23×. Thus, the same pass sequence can be optimal, weak, or even infeasible depending on the model.

This behavior goes beyond simple magnitude variation. Some pipelines are not merely suboptimal, but structurally infeasible on certain workloads, as indicated by the FAIL entries in Figure 3.2. The immediate conclusion is that no single fixed pipeline can be assumed to transfer reliably across architectures. The broader implication, developed later in this chapter, is that pass ordering interacts with workload

Table 3.1: Shorthand pass bundles used in Figure 3.2. Each token denotes a preprocessing-stage transformation bundle from the pass vocabulary.

| **Token** | **Expansion** | **Intuition** |
|---|---|---|
| CC | convert-conv-to-channels-last | convert NCHW layout to NHWC |
| T | transpose-convolution | transpose convolution operators |
| I | convert-conv2d-to-img2col | rewrite convolution into matmul form |

structure at a level deeper than ordinary parameter sensitivity.

### Different Tensor Shapes Need Different Pipelines

The previous section shows that pass ordering is model-dependent. This fragility persists even within the same architecture when only tensor shape changes. Figures 3.3a and 3.3b evaluate cross-shape portability by taking the best-found pipeline for each shape and running it on the other shapes of the same model family. Performance is normalized by the best speedup for the target shape, so the home-shape optimum is always 1.0×.

Two patterns are immediately visible. First, the best pipeline for one shape is often not the best for other shapes in the same family. Second, the loss from transferring a pipeline across shapes is non-trivial. For UNet, the pipeline optimized for input size $256 \times 256$ loses roughly 23% of the available speedup when evaluated on $384 \times 384$ (0.768× of that shape's best), and still underperforms on $640 \times 640$ (0.812×). For ViT, the best pipeline at $224 \times 224$ transfers poorly to $160 \times 160$ (0.839×) and also degrades on $512 \times 512$ (0.896×).

These degradations are substantial even within a fixed model family. They rule out the simpler strategy of tuning one representative shape and reusing that pipeline

for the rest of the family. Pass ordering in IREE is therefore shape-sensitive as well as architecture-sensitive.

### The Same Pass at Different Stages Changes the Downstream Problem

Previous sections show that different workloads and shapes demand different pipelines, but they do not yet explain what changes structurally when a pipeline is reordered. A natural hypothesis is that instruction-level proxies, such as compute density, memory pressure, or permutation overhead, should be sufficient to characterize these differences. If pass reordering only changed local code quality within a fixed downstream structure, such proxies would be adequate guides.

Table 3.2: Stage placement induces distinct backend regimes. Early placement can re-route dispatches toward a different lowering strategy, while later placement can trigger overhead-dominant regimes even when compute-density proxies improve.

| | | Structure | | | | Routing | | Instruction Profile | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **Stage** | **Lat. (ms)** | **#Disp.** | **#Encode Disp.** | **#pack/unpack** | **Double%** | **MMT4D%** | **Perm./Disp.** | **Pred./Disp.** | **Mem./Disp.** | **FMA%** |
| InceptionV3 | Baseline | 968 | 183 | 56 | 47 | 76.0 | 17.6 | 20.6 | 2.4 | 15.5 | 67.0 |
| | Preprocessing. | **932** | 183 | 56 | 48 | 77.1 | 16.4 | 21.8 | 2.4 | 15.6 | 89.5 |
| | Dispatch-create | 992 | 127 | 0 | 23 | 93.5 | 0.0 | **395.1** | **29.7** | 23.3 | **94.3** |
| ResNet-50 | Baseline | 1595 | 33 | 2 | 4 | 98.1 | 1.0 | 445.3 | 32.9 | 68.3 | 52.7 |
| | Preprocessing. | **662** | 37 | 3 | 7 | 98.1 | 1.9 | 328.6 | 13.4 | 92.0 | 45.6 |
| | Dispatch-create | 1575 | 33 | 0 | 0 | 99.0 | 0.0 | 409.4 | 31.1 | 55.4 | **77.8** |
| ViT-224 | Baseline | 480 | 211 | 62 | 20 | 71.5 | 28.5 | 11.9 | 2.6 | 5.5 | 37.6 |
| | Preprocessing. | **291** | 226 | 63 | 28 | **40.5** | **59.5** | 25.6 | 3.7 | 11.0 | 31.0 |
| | Dispatch-create | 450 | 214 | 63 | 23 | 39.0 | 61.0 | 14.4 | 2.9 | 7.0 | 38.3 |

Table 3.2 shows that this assumption fails under progressive lowering. To isolate stage-placement effects, we insert `linalg-generalize-named-ops` as a probe pass at 2 different stages: preprocessing and dispatch creation. The knob setting is fixed, so observed differences reflect stage placement rather than feature enablement. We analyze each workload by tracing the causal chain through three layers: (Structure) how dispatch formation changes, (Routing) how backend strategy routing shifts, and (In-

struction Profile) how the resulting instruction profile relates to end-to-end latency.

For each workload below, the causal chain follows the column groups in Table 3.2 left-to-right: a stage-placement decision first alters *Structure* (dispatch count, encoding dispatches, materialization), which determines *Routing* (the fraction of dispatches assigned to each backend strategy), which shapes the *Instruction Profile* (per-dispatch overhead and compute density). Latency is the final outcome. We classify each case by its dominant regime-shift subtype.

**InceptionV3: routing-level regime shift** InceptionV3 provides the most direct evidence of a routing-level regime shift induced by stage placement. At baseline and preprocessing, the compiler materializes 47–48 pack/unpack ops and routes $\sim$17% of dispatches through MMT4D, with latency at 968–932 ms. When generalization is delayed to the dispatch-creation stage, all encoding-related dispatches are eliminated, reducing total dispatches from 183 $\rightarrow$ 127 (this matches the number of encoding dispatches: 56 present at baseline.) Pack/unpack materialization halves from 47 to 23, and MMT4D routing disappears entirely.

The instruction-level consequences follow directly. With all dispatches forced into the double-tiling path, per-dispatch permutation operations rise 19$\times$ and predicate operations rise 12$\times$: the double-tiling path must perform on-the-fly data rearrangement for dispatches that pre-packed MMT4D layouts would have otherwise served. FMA% rises sharply from 67.0% $\rightarrow$ 94.3%, this shows that the higher arithmetic fraction does not translate into lower latency, because it is accompanied by a much more overhead-heavy dispatch regime. Latency regresses from 968 ms to 992 ms. InceptionV3 thus shows a fully traceable chain from stage placement, to structural change at the dispatch boundary to routing collapse to a slower backend regime.

**ResNet-50: partition-level regime shift** ResNet-50 shows that regime shifts are not limited to routing-level changes. Routing-level activity is minimal (4 pack/unpack, MMT4D% nearly unchanged). Yet latency drops by 2.4×.

The structural change is visible in dispatch count (33 → 37) and per-dispatch overhead (e.g., Perm./Disp. drops from 445.3 → 328.6). Early generalization restructures the IR so that dispatch-creation produces dispatches with substantially less rearrangement and predication work, given that they are routed broadly to the same strategies.

Late generalization at dispatch creation eliminates all remaining routing activity (0 pack/unpack, 0 encoding dispatches), raises FMA% to 77.8, but barely changes latency or dispatch count. As with InceptionV3, encoding dispatches that support data-tiling are removed; here, the dominant effect is not routing loss (since workload rarely use MMT4D) but the missed opportunity for dispatch restructuring that early placement enables.

**ViT-224: composition-level regime shift** ViT-224 reveals that regime shifts operate at finer granularity than aggregative strategy-mix percentages capture. Both non-baseline placements flip the strategy from double-tiling-dominant to MMT4D-dominant, with nearly identical percentages and comparable materialization operations (pack/unpack ops). Yet early placement achieves 291 ms while late placement reaches only 450 ms, a 54% latency gap that routing-level and partition-level metrics in Table 3.2 do not explain.

IR inspection reveals 3 differences. Because ViT-224 contains 12 identical transformer blocks, each per-block difference is amplified across the full model.

*(1) Softmax decomposition.* Under early generalization, the softmax computation is decomposed into separate dispatches: two reduction dispatches: `reduction_2364×197`

for max-subtraction and normalization, and one `elementwise_transpose_3×197×768` dispatch. Under late generalization, the same computation remains a single monolithic `softmax_12×197×197` dispatch. The finer decomposition allows the backend to independently tile and vectorize each component; the monolithic dispatch forces a single tiling strategy to handle structurally heterogeneous work (reduction vs. data movement).

*(2) QKV projection routing.* Early generalization routes the fused QKV projection as `matmul_like_197×2304×768` through the MMT4D pipeline. Late generalization instead produces a batched `matmul_like_3×197×768×768` routed through double-tiling. The shape change (fused 2304 v.s. batched $3 \times 768$) and the routing change (MMT4D vs. double-tiling) are jointly determine by the IR structure at the point when dispatch boundaries are drawn.

*(3) Encoding dispatch materialization.* Late generalization introduces an additional data-tiling encoding dispatch for the attention weight tensor (`encode_`$12 \times 197 \times 197 \times f32$), absent under early generalization, reflecting a different materialization decision for attention scores.

These three differences, softmax decomposition, projection routing, and encoding materialization, constitute a *composition-level* regime shift: the aggregate strategy-mix converges, but the decomposition and routing of individual dispatches remain performance-critical.

Taken together, these cases establish two important points. First, pass profitability in progressive lowering is fundamentally stage-dependent: even under a fixed knob setting, the same transformation can help, do little, or hurt depending on where it is inserted. Second, local proxies such as FMA% are insufficient for guidance, because downstream latency depends on structural effects including dispatch partitioning, routing, and layout-materialization behavior rather than on any single local indicator

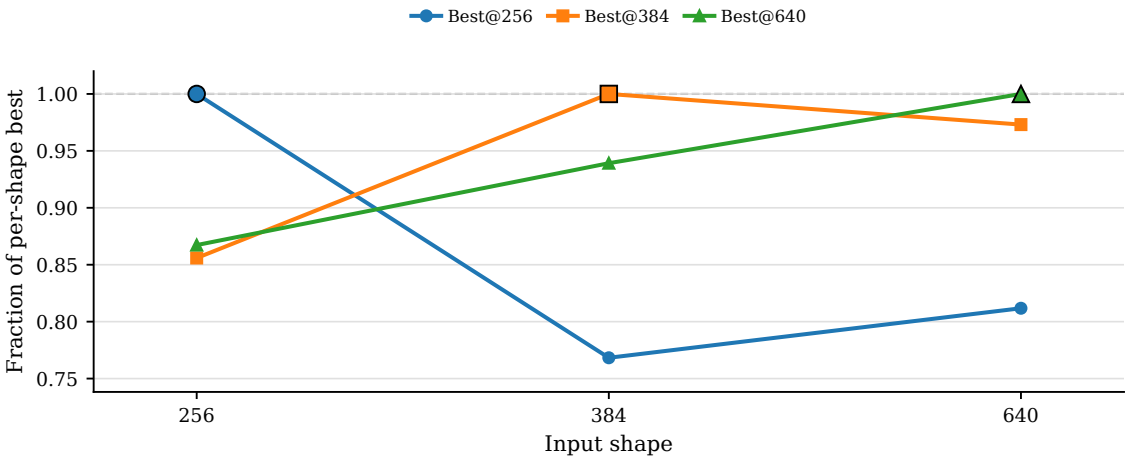
alone.

## Implications for Automated Search

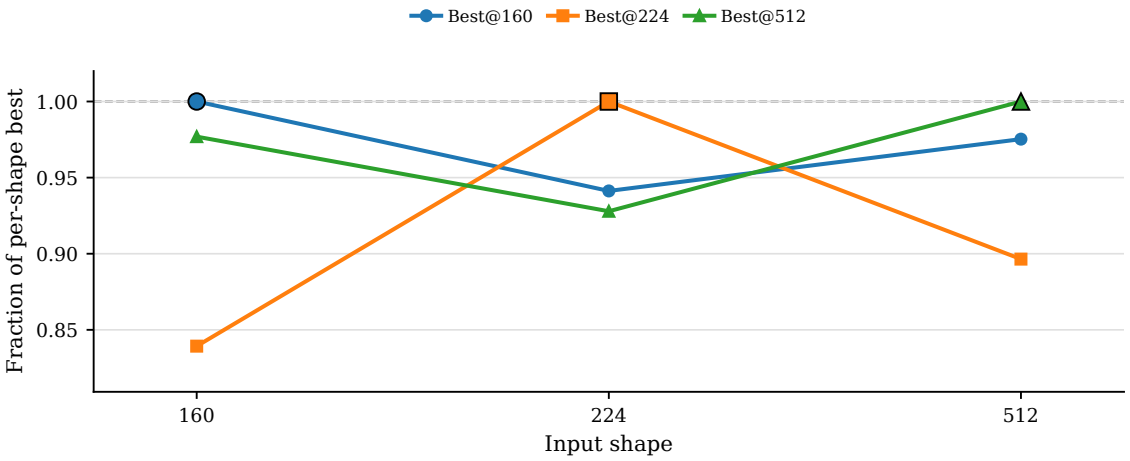
Taken together, the three observations in this chapter motivate the framework design presented in Chapter 4.

The first two sections show that no single fixed pipeline transfers reliably across workloads, motivating workload-specific automated search. We attribute this to the regime shifts identified in the analysis: the same transformation can lead to different downstream realizations depending on where it is applied, and that local proxies such as FMA% can fail to predict end-to-end latency.

These observations motivate an optimization procedure that reasons over long-horizon downstream consequences rather than optimizing local proxies, enforces the evolving applicability of passes as each transformation reshapes the IR, and captures non-local program structure to support regime-aware decisions. The next chapter presents $\mathcal{BO}3$, designed around these requirements.




(a) UNet: portability gaps across resolutions.




(b) ViT: portability gaps across image sizes.

Figure 3.3: Cross-shape fragility within a fixed model family. Each line corresponds to the best pipeline found for one shape, evaluated on all shapes in that family. Solid markers denote the home shape, where the pipeline is best-found by construction (1.0×). The drop away from the home shape shows that different tensor sizes can induce different profitable pass-ordering.

# Chapter 4

# $\mathcal{BO}3$ FRAMEWORK

The empirical observations in Chapter 3 impose three design requirements on an automated pass-ordering framework. First, cross-model and cross-shape sensitivity imply that no fixed pipeline transfers reliably across workloads, so the method must support workload-specific optimization. Second, stage-dependent regime shifts show that local proxies such as FMA% can be anti-correlated with end-to-end latency, so the framework should optimize directly against measured final performance rather than dense proxy-based signals. Third, the regime-shift analysis shows that pass applicability and profitability depend on non-local producer–consumer structure, so the method must represent compiler state structurally while enforcing evolving pass legality.

Figure 4.1 summarizes the overall workflow. $\mathcal{BO}3$ operates in three stages. Stage I screens backend knob configurations and retains a small set of anchors. Stage II trains a pass-ordering policy for each retained anchor. Stage III samples bounded post-training rollouts from each trained checkpoint and selects final candidates under a common high-fidelity evaluation protocol. This decomposition reflects the structure of the optimization problem under progressive lowering: backend knobs define downstream code-generation configurations, while preprocessing-stage pass sequences reshape the upstream IR.

## Stage I: Backend Knob Screening

IREE's backend exposes a discrete set of code-generation parameters, such as tiling, unrolling, masking or peeling behavior, and vectorization flags, referred to

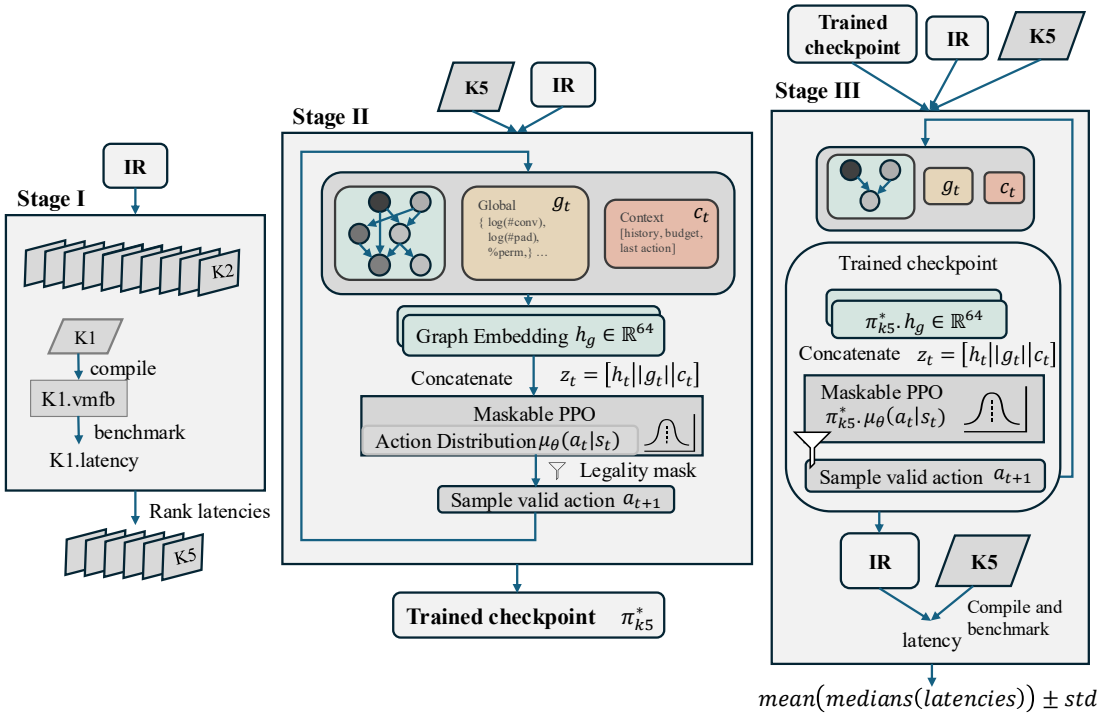



Figure 4.1: Overview of $\mathcal{BO}3$. Stage I screens backend knob configurations and retains the top-$K$. Stage II trains one legality-constrained RL policy per retained anchor. Stage III samples bounded post-training rollouts and re-evaluates under a common high-fidelity protocol.

collectively as *knobs*. Let

$$\mathcal{K}_{\text{defs}} = K_1 \times K_2 \times \cdots \times K_N$$

denote the full discrete knob space, where each $K_i$ is a finite set of choices. A specific assignment $k \in \mathcal{K}_{\text{defs}}$ fixes the backend code-generation configuration for a workload.

A natural alternative would be to place backend knobs and pass actions into one joint RL action space. $\mathcal{BO}3$ does not take that approach. Many backend knobs have no observable manifestation in the preprocessing-stage IR; their effects emerge only in later lowering or backend code generation. Including them directly in the policy action space would therefore create a mismatch between what the policy can observe

and what it is asked to choose.

Stage I instead treats knob combinations as anchors. Each candidate anchor is evaluated on the baseline workload under a coarse benchmarking budget and the top-$K$ configurations, ranked by median latency, are selected for the next stage. In the evaluation presented in Chapter 8, this retained set is intentionally small (three anchors per workload), so that Stage II can focus its search budget on a bounded number of plausible backend regimes.

This design is pragmatic rather than exact. Screening knob assignments on the baseline IR may discard a backend configuration that would become favorable after pass reordering reshapes the program. However, the purpose of Stage I is not to solve the entire optimization problem by itself. Its role is to provide a small set of stable downstream regimes so that Stage II can specialize its search rather than entangling backend and pass decisions in the joint large search space.

### Stage II: Pass Ordering via Reinforcement Learning

For each retained knob anchor $k$, Stage II trains an independent RL policy to search over preprocessing-stage pass sequences. The optimization problem is formulated as an episodic finite-horizon Markov Decision Process (MDP) in which the environment is instantiated under a fixed backend knob anchor, so the policy specializes to one downstream code-generation regime at a time. The full MDP formulation, including the state representation, action space, legality-masking rules, reward function, and training configuration, is developed in Chapter 5.

This section provides a conceptual overview of the key design choices and their motivation; Chapter 5 supplies all implementation-level detail.

### *Why One-Knob-One-Policy*

The one-knob-one-policy design follows directly from the motivation chapter. Cross-workload sensitivity implies that a universal fixed policy is unlikely to transfer reliably. At the same time, backend knobs and pass ordering play different roles: knobs define the downstream lowering envelope, while passes reshape the upstream IR presented to it. Conditioning the search on a fixed anchor therefore reduces interference across incompatible backend regimes and makes learned behavior easier to interpret.

### *Structured Compiler-State Observation*

The policy does not consume the raw compiler IR. Instead, it receives an observation constructed from the current IR through a graph-based encoding pipeline. The observation consists of three parts: (i) an SSA def-use graph, (ii) a compact vector of global module-level features, and (iii) a pass-context vector encoding information such as remaining budget and recent action history. A GraphSAGE-style encoder (Hamilton *et al.*, 2017) compresses the variable-size graph into a fixed-dimensional embedding, which is concatenated with the global and context vectors before being passed to the policy and value networks.

This graph-based representation is motivated by the observation that regime shifts depend on non-local program structure. Pass profitability and pass legality are influenced not only by local operations, but also by producer–consumer relationships, operator composition, and layout-sensitive structure across the IR.

### *Legality-Aware Exploration*

In progressive lowering, the set of valid transformations changes as the IR evolves. $\mathcal{BO}3$ therefore uses a deterministic legality mask at each decision step, restricting the policy to only those actions that are valid for the current compiler state. This mask is not an auxiliary convenience: it is a first-class component of the optimization problem, because legality evolves together with lowering depth.

### *Sparse Terminal Reward*

The analysis in Chapter 3 shows that local IR-level proxies are unreliable as optimization objectives, while intermediate benchmarking after every pass would be too expensive. $\mathcal{BO}3$ therefore assigns the principal reward at episode termination based on end-to-end measured latency, with lightweight shaping terms only to discourage invalid or unnecessarily long trajectories.

### *Policy Optimization*

The legality-constrained policy is optimized using Maskable PPO (Schulman *et al.*, 2017; Huang and Ontanón, 2022). This choice follows from two requirements: the reward is sparse and high-variance because it depends on end-to-end benchmarking, and the action space is dynamically masked by legality constraints. Maskable PPO inherits the update stability of PPO while integrating action masking directly into the policy distribution.

The resulting learned object is a trained *knob-conditioned policy checkpoint* rather than a single fixed pass sequence. This distinction is important for understanding Stage III.

### Stage III: Post-Training Candidate Selection

The output of Stage II is a trained policy checkpoint $\mu^*_{\theta,k}$ for each retained knob anchor. Final candidate selection is intentionally decoupled from training. This avoids conflating policy quality with best-case training-time luck and makes evaluation depend on a common post-training protocol.

For each trained checkpoint, a bounded number of post-training rollouts is sampled by stochastically drawing legal actions from the masked policy distribution until the agent emits `EVAL` or the trajectory terminates. Each rollout produces a candidate pass sequence. These candidate sequences are then compiled and validated under the corresponding knob anchor; invalid candidates are discarded.

All valid candidates are re-evaluated under a common high-fidelity latency protocol using repeated invocations and repeated inner measurements. The final selected candidate is the one with the lowest mean latency across the valid post-training samples.

This design reflects an important methodological distinction. Stage II learns a *distribution* over promising legality-constrained trajectories. Stage III is responsible for final candidate selection under a common evaluation standard. The trained checkpoint should therefore be understood as a generator of candidate sequences rather than as a guarantee that the single best training-time trajectory is the final reported result.

# Chapter 5

# RL ENVIRONMENT DESIGN

This chapter details the reinforcement learning environment used in Stage II of $\mathcal{BO}3$ (Chapter 4). Chapter 4 motivated the high-level design choices; here the discussion turns to their concrete realization: the MDP formulation, observation engineering, legality-masking rules, reward computation, and policy-training configuration. Figure 5.1 illustrates the resulting data path.

## MDP Formulation

Stage II formulates preprocessing-stage pass search as an episodic finite-horizon Markov Decision Process,

$$(\mathcal{S},\ \mathcal{A},\ \mathcal{P},\ \mathcal{R},\ \mathcal{M},\ \gamma), \tag{5.1}$$

where $\mathcal{S}$ denotes the compiler state space, $\mathcal{A}$ the action space of preprocessing-stage passes, $\mathcal{P}$ the transition function induced by applying a pass to the current IR, $\mathcal{R}$ the reward function, $\mathcal{M}$ the legality-mask function, and $\gamma$ the discount factor. Every environment instance is parameterized by a fixed backend knob configuration $k$ selected in Stage I (Chapter 4), so that all compilation and benchmarking within one training run share a single downstream code-generation regime.

## State and Observation

The environment state $s_t$ is the full compiler IR module after $t$ pass applications. The policy does not consume $s_t$ directly. Instead, it receives a structured observation

$$o_t = \big(\mathcal{V}_t,\ \mathcal{E}_t,\ \mathbf{g}_t,\ \mathbf{c}_t\big), \tag{5.2}$$

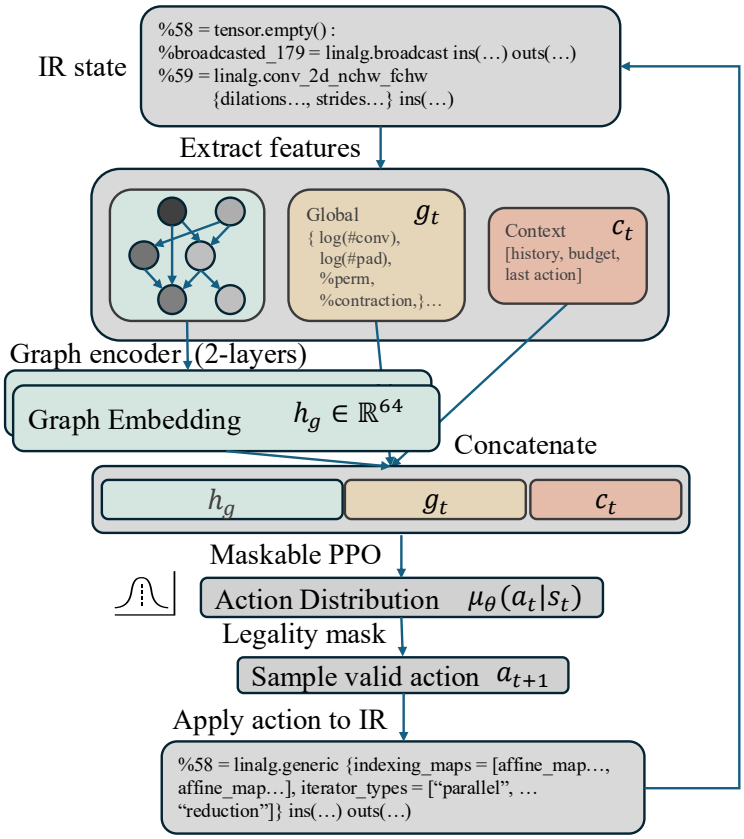



Figure 5.1: Stage II data path. At each step, the current IR is converted to an SSA graph, encoded by GraphSAGE, and passed through the policy network. The legality mask zeros out invalid actions before sampling.

where $\mathcal{V}_t$ and $\mathcal{E}_t$ are the nodes and edges of an SSA def-use graph extracted from the current IR, $\mathbf{g}_t$ is a global feature vector, and $\mathbf{c}_t$ is a pass-context vector.

### *Node Features*

Each operation node $v \in \mathcal{V}_t$ is represented by a compact feature vector designed to preserve distinctions that are most relevant to downstream lowering behavior. The features encode six categories of information:

### State and Observation

The environment state $s_t$ is the full compiler IR module after $t$ pass applications. The policy does not consume $s_t$ directly. Instead, it receives a structured observation

$$o_t = \big(\mathcal{V}_t,\ \mathcal{E}_t,\ \mathbf{g}_t,\ \mathbf{c}_t\big), \tag{5.3}$$

where $\mathcal{V}_t$ and $\mathcal{E}_t$ are the nodes and edges of an SSA def-use graph extracted from the current IR, $\mathbf{g}_t$ is a global feature vector, and $\mathbf{c}_t$ is a pass-context vector.

#### *Node Features*

Each operation node $v \in \mathcal{V}_t$ is represented by a compact feature vector designed to preserve distinctions that are most relevant to downstream lowering behavior. The features encode six categories of information:

1. **Operation-type bucket.** A categorical encoding of the operation class, distinguishing contraction, elementwise, reduction, transpose, padding, and other operation families.

2. **Shape and dynamicity.** Rank, selected dimension sizes, and an indicator of whether any dimensions of the input or output tensors are dynamic.

3. **Topology and usage.** Structural information such as fan-in, total use count, and number of distinct consumer operations.

4. **Computational cost.** A coarse proxy for the amount of arithmetic associated with the operation, estimated from output tensor shapes and reduction dimensions.

5. **Layout and permutation behavior.** Indicators for transposition, broadcasting, and non-identity indexing maps that signal layout-sensitive structure.

6. **Generic-op semantic category.** A recovered semantic label for `linalg.generic` operations, as described below.

*Generic-Op Semantic Recovery*

As discussed in Section 2, generalizing a named `linalg` operation erases its operation-specific name while preserving its semantics. If all generalized operations were treated as undifferentiated `linalg.generic` nodes, structurally important distinctions, for example, whether a generic op corresponds to a convolution-like pattern or a reduction, would be lost.

$\mathcal{BO}3$ therefore recovers semantic categories for generalized operations by inspecting their affine indexing maps and iterator types. For example, an operation whose maps encode a contraction pattern with batch, spatial, and reduction iterators can be classified as convolution-like even though its IR name is `linalg.generic`. This preserves information that would otherwise be lost if the representation relied on raw operation names alone.

*Global Features*

The global feature vector $\mathbf{g}_t$ summarizes module-level statistics that complement the graph embedding. These include log-scale operation counts by category (e.g., convolutions, pads, transposes), the fraction of permutation and contraction operations, and coarse structural-composition signals. This summary helps the policy reason about overall workload composition without requiring every such signal to be inferred through local message passing alone.

### *Pass Context*

The pass-context vector $\mathbf{c}_t$ provides step-dependent search metadata: the one-hot identity of the most recent action, the fraction of the step budget consumed, and a short history window of recent actions. It is kept separate from the graph so that the encoder focuses on IR structure while the policy still has access to search-time context.

### *Budgeted Graph Construction*

Because observation extraction and policy inference are performed at every decision step, graph construction must remain bounded in cost. $\mathcal{BO}3$ imposes fixed caps on the number of nodes ($N_{\max}$=5000) and edges ($E_{\max}$=20000) included in the observation. When the IR exceeds these limits, nodes are retained according to a deterministic priority policy: compute-heavy and layout-relevant operations are kept first, followed by their immediate producers and consumers, before lower-priority nodes are truncated. This ensures that repeated evaluations of the same compiler state produce identical bounded observations, improving reproducibility.

## Graph Encoding

The extracted SSA graph is encoded using a two-layer GraphSAGE (Hamilton *et al.*, 2017) message-passing network with hidden dimension 64. Neighborhood aggregation is performed over SSA edges, and the resulting node embeddings are mean-pooled to obtain a fixed-dimensional graph embedding

$$\mathbf{h}_G \in \mathbb{R}^{64}.$$

The final policy input is formed by concatenation:

$$\mathbf{z}_t = \big[\, \mathbf{h}_G \parallel \mathbf{g}_t \parallel \mathbf{c}_t \,\big]. \tag{5.4}$$

This encoder handles graphs of varying size and topology while producing a representation suitable for a standard policy network.

### Actions and Legality Masking

The action space consists of a discrete vocabulary of preprocessing-stage passes together with a terminal `EVAL` action. `EVAL` terminates the current trajectory and triggers full compilation and benchmarking.

At each step, the environment computes a deterministic binary legality mask

$$\mathbf{m}_t = \mathcal{M}(s_t) \in \{0,1\}^{|\mathcal{A}|}, \tag{5.5}$$

which restricts the valid action subset to $\mathcal{A}_{\text{valid}}(s_t) = \{a \mid \mathbf{m}_t[a] = 1\}$.

The masking rules encode conditions such as the following:

- **IR-structure preconditions:** a convolution-specific pass is masked when no convolution operations remain in the current IR.
- **Per-pass application limits:** each pass may be applied at most a fixed number of times per trajectory.

Masking serves a dual role: it prevents the policy from wasting exploration on structurally meaningless actions, and it aligns the effective search space with the evolving legality induced by progressive lowering.

### Reward

Let $L_0$ denote the baseline `-O3` latency, and let $\mathrm{lat}(s_T)$ denote the measured latency of the compiled artifact at the terminal state. The reward is defined as:

$$r_t = \begin{cases} -\lambda_{\mathrm{err}}, & \text{if } a_t \text{ causes a compile or runtime error,} \\ -\alpha \cdot t, & \text{if } t < T \text{ and the transition is valid,} \\ \log\Big(\dfrac{L_0}{\mathrm{lat}(s_T)}\Big), & \text{if } t = T \text{ and evaluation succeeds.} \end{cases} \tag{5.6}$$

The error penalty discourages trajectories that crash the compiler or runtime. The small progressive step cost biases the policy against unnecessarily long sequences. The terminal log-speedup is the primary optimization signal; the logarithmic form reduces scale asymmetry between large slowdowns and large speedups, which helps stabilize learning. The shaping terms are secondary and are included only to discourage obviously undesirable behavior.

### Policy Optimization with Maskable PPO

#### *Network Architecture*

The policy $\mu_\theta$ and value function $V_\phi$ share a common feature extractor that maps the concatenated observation $\mathbf{z}_t$ (Equation 5.4) through fully connected layers. Two separate heads branch from this trunk:

- a *policy head* that outputs logits $\ell_a$ over the action vocabulary, and
- a *value head* that predicts a scalar state value.

*Masking Integration*

Before the policy distribution is formed, logits corresponding to masked actions ($\mathbf{m}_t[a] = 0$) are set to $-\infty$, so that those actions receive zero probability mass after softmax normalization. Gradients are therefore computed only over valid actions. This prevents the policy from receiving training signal that encourages illegal trajectories and ensures that masking is enforced at the distribution level, not as a post-hoc rejection step.

*Training Objective*

$\mathcal{BO}3$ uses PPO's clipped surrogate objective (Schulman *et al.*, 2017):

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\Big[\min\big(w_t(\theta)\,\hat{A}_t,\ \text{clip}(w_t(\theta),\,1{-}\epsilon,\,1{+}\epsilon)\,\hat{A}_t\big)\Big], \tag{5.7}$$

where $w_t(\theta) = \mu_\theta(a_t \mid s_t) \,/\, \mu_{\theta_{\text{old}}}(a_t \mid s_t)$ is the probability ratio and $\hat{A}_t$ is the generalized advantage estimate. PPO is attractive here for two reasons. First, the clipped surrogate objective bounds how far the policy can drift from the data-collection policy when the same rollout batch is reused across multiple training epochs. This is important because our reward signal, derived from end-to-end latency measurement, is both sparse and noisy: a single unusually fast or slow benchmark run could otherwise cause a disproportionately large policy update. Second, advantages are estimated with GAE, which interpolates between one-step TD estimates and full Monte Carlo returns. Pure Monte Carlo returns are unbiased but exhibit high variance across trajectories, especially when the only informative reward arrives at the final step. TD estimates are lower-variance but depend on critic accuracy, which is poor early in training. GAE balances the two, allowing multi-step credit to propagate back through long pass sequences while dampening trajectory-level noise.

*Training Configuration*

Each Stage II policy is trained under a fixed backend knob anchor for a bounded number of environment steps. The training uses:

- $T_{\text{steps}} = 3{,}000$ total environment steps,
- a rollout buffer of $n_{\text{steps}} = 50$,
- an entropy coefficient of 0.001 to maintain mild exploration,
- standard SB3-Contrib(Raffin *et al.*, 2021) defaults for remaining Maskable PPO hyperparameters.

The resulting learned object is a knob-conditioned policy checkpoint. $\mathcal{BO}3$ is evaluated as a bounded-budget optimization procedure: the checkpoint generates candidate pass sequences through post-training rollout sampling (Chapter 4), and final selection is performed under a common high-fidelity latency protocol.

# Chapter 6

# EVALUATION SETUP

## Experimental Setup

All experiments are conducted on an AMD Ryzen Threadripper PRO 7955WX workstation running Ubuntu 22.04.5 LTS. To ensure reproducibility, this thesis uses `iree-base-compiler` and `iree-base-runtime` version 3.9-rc (snapshot dated 2025-11-25). The reinforcement learning environment and policy training pipeline are implemented using PyTorch 2.9.1, SB3-Contrib 2.7.0, and Gymnasium 1.2.0.

## Model Ingestion

All benchmark models originate from PyTorch and are exported to ONNX using `torch.onnx.export` with `opset-version=18`. The exported ONNX models are then compiled through the IREE toolchain under the selected backend knob configuration and pass sequence.

## Benchmarking Infrastructure

To reduce runtime variance, all benchmarking commands are executed on the IREE `local-sync` device and pinned to a single physical CPU core using `taskset`. Unless otherwise specified, identical benchmarking tools and latency-aggregation procedures are used for the `-O3` baseline, knob-only evaluation, random-rollout reference, beam-search comparison, and RL-selected candidates.

### Evaluation Protocol

All reported candidates are measured under a common latency protocol. Repeated invocations and repeated inner measurements are used to reduce runtime noise and ensure fair comparison across candidate sources. The same protocol is applied to baseline, knob-only, random, beam, and RL-generated candidates.

### Mask-Aware Rollout Execution

Legality masks are applied during both training and post-training rollout sampling. Accordingly, rollout-time inference uses the same legality-aware action interface as training, ensuring consistency between policy learning and candidate generation. The output of Stage II is therefore a trained knob-conditioned checkpoint, and final candidates are generated only through bounded masked rollout sampling followed by common high-fidelity re-evaluation.

# Chapter 7

# EXPERIMENTAL METHODOLOGY

## Evaluation Scope and Claims

This thesis evaluates $\mathcal{BO}3$ as a bounded-budget optimization procedure rather than as a zero-shot predictor that emits a single universally transferable pass sequence. For each workload–knob pair, Stage II produces a trained knob-conditioned policy checkpoint, and final comparison is performed by sampling a fixed number of post-training masked rollouts and selecting the best valid candidate under a common evaluation protocol.

Accordingly, the cross-workload evaluation is intended to measure the reproducible effectiveness and stability of the overall optimization procedure across diverse computational regimes, rather than universal policy transfer or deterministic single-rollout optimality.

## Benchmarks and Tasks

The evaluation suite comprises 7 representative deep learning model families chosen for architectural heterogeneity, including standard convolutional networks, natural language processing models, vision transformers, object detection models, and high-resolution encoder-decoder architectures. The evaluated workload instances are AlexNet, ResNet-50, DenseNet, BERT-Base, ViT-224, YOLOv5, and UNet at three spatial resolutions: $256 \times 256$, $384 \times 384$, and $640 \times 640$.

For each workload instance, Stage I identifies the top-3 backend knob configurations. Stage II then trains an independent knob-conditioned policy checkpoint for

each retained configuration. The resulting evaluation therefore spans $9 \times 3 = 27$ optimization tasks, each defined by a workload instance and a fixed backend regime.

This setup is intended to evaluate whether the learned policies reliably generate strong post-training candidates under bounded rollout budgets across diverse regime-shift settings.

## Evaluation Protocol and Fairness Controls

### *Latency Measurement*

For every compiled candidate, including the `-O3` baseline, knob-only baseline, random-rollout reference, and RL-selected candidate, latency is measured using a two-level aggregation protocol to reduce noise.

Each candidate is benchmarked with repeated inner measurements per invocation and repeated outer invocations. Latency is reported as

$$\text{mean(medians)} \pm \text{std(medians)}.$$

The protocol uses $R_{\text{inner}} = 5$ repetitions per invocation and $R_{\text{outer}} = 10$ invocations.

**Random-rollout reference** For each task, the random baseline samples actions uniformly from the same pass vocabulary under the same legality mask, maximum sequence length, and per-pass limits as the RL environment. Under a common budget of $B_{\text{eval}}$=610 attempted rollouts, accounting for the ∼600 environment interactions consumed by the RL policy during training, plus its 10 evaluation rollouts. The reported random-reference latency is the best measured latency among all valid random rollouts observed within this fixed total budget.

This baseline serves as a bounded-budget reference for unguided search under the same legality-constrained setting.

*RL Candidate Selection*

For each trained knob-conditioned policy, post-training evaluation rollouts are performed by stochastically sampling actions from the masked policy distribution until `EVAL` is emitted or the rollout terminates. All valid candidates are then re-measured using the same latency protocol, and the best latency among these post-training sampled candidates is reported as **RL Found**.

Thus, the main comparison is between the best valid candidate produced by the trained policy under a fixed rollout-evaluation budget and the best valid candidate produced by random rollout under the corresponding bounded budget.

*Rollout Robustness*

In addition to best-found latency, this thesis reports the fraction of sampled post-training rollouts that beat the `-O3` baseline:

$$\mathrm{WinRate}(\tau) = \frac{|\{i \in [1, N_{\mathrm{ep}}] \mid \mathrm{lat}(\sigma_i) < \mathrm{lat}_{O3}(\tau)\}|}{N_{\mathrm{ep}}}.$$

This metric indicates whether the trained checkpoint places substantial probability mass on improving legal trajectories, rather than yielding gains only through a single best-case sample.

*Training Configuration*

Each Stage II policy is trained for a fixed total environment-step budget with a PPO rollout buffer of fixed size. The training configuration uses $T_{\mathrm{steps}} = 3000$ total environment steps, $n_{\mathrm{steps}} = 50$, and an entropy coefficient of 0.001, while other Maskable PPO hyperparameters follow SB3-Contrib defaults.

# Chapter 8

# EXPERIMENTAL CLAIMS

This chapter evaluates $\mathcal{BO}3$ through a sequence of experimental claims. Rather than presenting the results as an undifferentiated collection of tables and case studies, the chapter is structured around four questions: whether $\mathcal{BO}3$ consistently improves over the default pipeline, whether the gains arise from learned components rather than search budget alone, whether the gains are mechanistically explained by regime shifts, and whether the resulting design is robust and practically meaningful.

## Claim 1: $\mathcal{BO}3$ Consistently Outperforms the Default Pipeline

### *Claim*

Across diverse workloads and backend configurations, $\mathcal{BO}3$ discovers workload-specific pass sequences that consistently improve over the default IREE `-O3` pipeline.

### *Experiment*

For each of the 27 optimization tasks (9 workload instances $\times$ 3 knob anchors), Stage II produces a trained knob-conditioned policy checkpoint. A bounded number of post-training rollouts is then sampled, and the best valid candidate is selected under the common high-fidelity latency protocol described in Chapter 7.

### *Results and Analysis*

The main results show that all 27 tasks improve over the default `-O3` pipeline. Across the 9 workload instances, $\mathcal{BO}3$ achieves a geometric-mean speedup of 5.66$\times$ and an arithmetic-mean speedup of 6.89$\times$.

The gains are strongly workload-dependent, which is consistent with the regime-shift analysis developed earlier in the thesis. Two patterns are particularly important.

First, some workloads are *knob-dominant*. For ViT-224, much of the gain is already captured by Stage I: the best knob anchor alone reaches approximately 320 ms relative to the `-O3` baseline, and Stage II provides a further improvement to 287 ms. This indicates that, for some workloads, selecting a strong downstream backend regime accounts for much of the available performance gain.

Second, other workloads are *pass-ordering-dominant*. For UNet-256, the best knob anchor by itself yields only a negligible improvement, whereas the RL-found sequence under the same anchor achieves a much larger speedup. UNet-384, UNet-640, ResNet-50, and YOLOv5 show the same overall pattern at different scales. In these cases, the main gains are not explained by backend parameter selection alone, but by upstream structural changes induced by pass ordering.

*Conclusion*

$\mathcal{BO}3$ consistently improves over the default pipeline across all evaluated tasks. The relative contribution of Stage I and Stage II is workload-dependent, supporting the rationale for the 3-stage design rather than a single-stage optimization procedure.

Claim 2: The Gains Reflect Learned Components, Not Search Budget Alone

*Claim*

$\mathcal{BO}3$'s advantage over simple baselines arises from its learned components, particularly graph-structured observations and legality masking, rather than from search budget alone.

*Experiment 2a: RL vs. Random Rollout*

For each task, a random baseline samples actions uniformly from the same pass vocabulary under the same legality mask and bounded computation budget $B_{eval} = 610$ as the RL system.

*Analysis*

On relatively shallow tasks, such as YOLOv5 and BERT-Base, random rollout and RL-guided rollout can find nearly identical candidates. This suggests that the profitable region is comparatively easy to reach even without learned guidance.

On structurally harder tasks, however, the gap is substantial. The largest differences on UNet-256, UNet-384, and UNet-640, where random search fails to enter the high-performing regime that the trained policy reaches. This indicates that the benefit of learning is most visible when good legal trajectories are sparse or require longer-horizon structural coordination.

*Experiment 2b: RL vs. Beam Search*

To provide a stronger non-random reference, we also compares $\mathcal{BO}3$ against beam search as shown in Figure 8.1. At each depth, beam search expands legal actions from each beam entry and retains the top candidates according to compiled latency.

*Analysis*

On BERT-Base, UNet-256, and YOLOv5, beam search ties the RL candidate. These are cases where the best pass sequence is short and the profitable regime is reachable through comparatively simple structured expansion.

On ResNet-50, ViT-224, and UNet-640, RL reaches lower-latency regimes than

beam search. In the UNet-640 case, beam search finds a shorter sequence, whereas $\mathcal{BO}3$ discovers a longer sequence whose earlier steps are not individually rewarding but reshape the IR so as to enable a better downstream regime.

*Experiment 2c: Component Ablation*

We further isolates the source of $\mathcal{BO}3$'s advantage through component ablation on UNet-640 Top-1. The ablation removes legality masking, the graph encoder, or both, the results are shown in Table 8.2.

*Analysis*

Removing legality masking lowers the rollout win rate, showing that a substantial fraction of trajectories becomes invalid or unproductive without explicit legality control. Removing the graph encoder is more severe: the win rate drops further, and the policy becomes much less effective at identifying productive structural trajectories and appropriate termination points.

Most notably, removing both components reduces RL performance to approximately the same level as beam search. This supports the conclusion that the advantage of $\mathcal{BO}3$ on harder tasks is primarily explained by the learned components rather than by simply spending evaluation budget on search.

*Conclusion*

On easier tasks, several legality-constrained search strategies are sufficient to reach the profitable regime. On harder tasks, $\mathcal{BO}3$ reaches better regimes than random rollout and beam search under the evaluated budget. The evidence indicates that legality masking reduces wasted exploration, while graph-structured state provides the structural information needed to guide search beyond short-horizon local expansion.

Claim 3: Gains Are Mechanistically Driven by Regime Shifts

*Claim*

The performance improvements found by $\mathcal{BO}3$ are not adequately explained by local IR simplification alone. They are mechanistically driven by discrete downstream regime shifts.

*Experiment*

We study UNet-640 under the Top-1 knob anchor and compares three procedures: random rollout, beam search, and policy-guided rollout. This case is particularly useful because backend routing remains fixed, allowing the analysis to isolate structural changes in dispatch decomposition and execution behavior.

**Structural analysis (Table 8.3–8.4).** The three procedures produce qualitatively different dispatch structures despite identical backend routing (all dispatches use the double-tiling path). Table 8.4 summarizes the dispatch-type breakdown for the two candidates that apply img2col decomposition (beam and $\mathcal{BO}3$); IR-level inspection reveals 3 systematic structural differences behind these counts.

Random rollout (`T + EVAL`) transposes convolution operators but barely changes dispatch structure: dispatch count rises from 49 to 55, and latency drops modestly from 18067 ms to 14819 ms.

Beam search applies `I` (img2col), decomposing monolithic convolution dispatches into explicit transpose and matmul components. The dispatch count rises to 186 with 47 matmul dispatches and 47 generic data-rearrangement dispatches. Latency drops to 1 879 ms, attributed to dispatch restructuring, since no routing change occurs.

$\mathcal{BO}3$ finds `canonicalize + CC + I + EVAL`. The two preceding passes normalize ten-

sor expressions and convert the data layout from NCHW to NHWC before img2col decomposition. The total dispatch count is comparable, and the two candidates share largely the same semantic work, but the critical difference is not *what* gets computed, but *how the compiler linearizes and tiles the iteration space* of each dispatch.

*Layout propagation.* IR comparison shows that `CC`'s layout conversion propagates through the entire program, not limited to convolutions. Nearly every dispatch shows permuted dimensions: a broadcast dispatch shaped $102400 \times 64$ under NCHW layout becomes $64 \times 102400$ under $\mathcal{BO}3$; a generic re-arrangement dispatch shaped $320 \times 320 \times 7 \times 7 \times 3$ becomes $3 \times 7 \times 7 \times 320 \times 320$; matmul shapes are also reordered. Because dimension order determines which axis is innermost in memory, this reordering changes the contiguous dimension, the innermost loop, the vectorized dimension, and how working sets land on pages and cache lines, for every dispatch in the program.

*Decomposition changes.* Table 8.4 shows that the layout change also alters how the compiler expresses movement and re-arrangement works. $\mathcal{BO}3$ produces more explicit transpose dispatches ($6 \rightarrow 14$) and fewer broadcast dispatches ($41 \rightarrow 35$); at three resolution levels, beam folds reduction work into generic dispatches (e.g., `generic_64×80×80` where $\mathcal{BO}3$ preserves the named form (e.g., `reduction_128×80×80×64`). However, these dispatch-type shifts are best understood as *symptoms* of the layout change, not as the primary mechanism of speedup; they reflect different lowering paths for the same semantic work, and do not by themselves evidently express which variant executes faster. The hardware counters below identify the actual execution-level mechanism.

**Hardware-counter analysis (Table 8.5).** The counter data identifies the execution-level mechanism behind the layout-induced structural differences.

*Address translation locality* TLB misses drop 6×, the single largest signal in the counter data. The pervasive dimension reordering changes the memory traversal pattern of every dispatch, and the NHWC traversal is dramatically friendlier to translation locality on this microarchitecture.

*SIMD utilization* Packed 256-bit SIMD retirement rises from 813M (beam) to 22B ($\mathcal{BO}3$), a 27× increase. Under NCHW, the channel dimension is not innermost, so the backend must gather across channels to populate SIMD registers. Under NHWC, channels are innermost, making contiguous memory spans align with the SIMD lane width.

*Stall pressure and throughput* Not-complete events drop from 73B to 57B, and IPC rises from 1.15 to 1.6: the more regular data arrival pattern under NHWC keeps the pipeline busier.

*More instructions, fewer cycles* A key interpretive point: $\mathcal{BO}3$ retires *more* total instructions than beam (156B vs. 124B), yet completes in fewer cycles and lower wall time. We can tell from the data that $\mathcal{BO}3$ does not win by doing less work; it wins because the additional instructions are cheap relative to stalls they avoid, and the SIMD throughput they unlock.

**Mechanistic summary.** The causal chain is:

1. **Layout change:** `canonicalize+CC` convert the data layout from NCHW to NHWC, changing the logical tensor dimension order seen by the compiler.
2. **Dispatch decomposition changes:** the layout propagates through the entire dispatch population, reorienting iteration spaces and altering how the compiler expresses data-movement work (visible as rotated dispatch shapes and shifted type composition in the IR).
3. **Loop nest and memory behavior improve:** NHWC places channels inner-

most, making contiguous memory spans align with SIMD lanes, and improving page-level spatial locality across all dispatches.

4. **Hardware counters reflect the change:** TLB misses drop 6×, packed-SIMD activity rises 27×, stall pressure falls, and IPC rises from 1.15 to 1.6.
5. **Execution efficiency dominates instruction count:** despite retiring 26% more instructions, $\mathcal{BO}3$ completes in 11% fewer cycles, yielding the 1 708 ms runtime.

Critically, no per-dispatch proxy captures this mechanism. Per-dispatch permutation and memory counts *increase* under $\mathcal{BO}3$ (Table 8.3), and the dispatch type composition shifts in ways that do not obviously favor either candidate. The speedup emerges from a population-wide layout decision whose consequences are visible only at the hardware-counter level: a regime shift (Definition 1) in which the same semantic computation, decomposed under different dimension orders, produces qualitatively different execution behavior.

*Conclusion*

The UNet-640 case study complements the motivation chapter by showing the mechanism through which $\mathcal{BO}3$'s gains arise. InceptionV3 demonstrated that stage placement can eliminate an entire backend strategy (routing-level shift). ResNet-50 showed dispatch restructuring within the same strategy (partition-level shift). UNet-640 adds a distinct pattern: even when backend routing is fixed (100% double-tiling), partitions are similar, and total dispatch counts are comparable (183 vs. 187), a population-wide layout change produces qualitatively different execution behavior: 27× higher SIMD utilization, 6× fewer TLB misses. And this is not because the compiler does less work, but because it *orients* the same work differently in memory. The common thread across all three cases is that profitable pass ordering reshapes how downstream stages decompose and lower the program, producing effects that no

stage-local proxy can predict.

### Claim 4: Robustness and Design Validation

#### *Rollout Robustness (Table 8.6)*

Because **RL Found** is defined as the best valid candidate among bounded post-training rollouts, it is important to check that the learned checkpoint is not succeeding only through a single lucky sample. We therefore report rollout win rate, defined as the fraction of sampled post-training rollouts that beat the `-O3` baseline.

Across the 27 tasks, the mean rollout win rate is 95.93%. On higher-gain workloads, including the UNet family, YOLOv5, and AlexNet, win rates are generally very high. This supports the interpretation that the learned checkpoints place substantial probability mass on improving legal trajectories rather than memorizing one brittle pass sequence.

#### *Anchor-Ranking Stability*

We also evaluate whether decomposing Stage-I from RL training remains meaningful after pass search. To do so, it compares Top-1 and Worst-1 knob settings on ViT-224 and ResNet-50.

The result is that pass search does not rescue a poorly ranked backend anchor into a top-performing one. On ViT-224, both learned sequences remain much faster under the Top-1 knob than under the Worst-1 knob. On ResNet-50, the best sequence found under the stronger anchor still outperforms what is achievable under the weaker anchor. This indicates that Stage I is not merely an arbitrary preprocessing step; the retained anchors continue to define meaningful downstream performance envelopes after Stage II.

*Practical Cost*

.

$\mathcal{BO}3$ incurs a nontrivial offline autotuning cost. We report a geometric-mean Stage II training time of 5.83 hours per task, with a range of 2.8 to 12.6 hours. $\mathcal{BO}3$ is therefore best viewed as an offline deployment-time autotuning method, with cost comparable to typical tensor-program autotuning budgets Trofin *et al.* (2021); Zheng *et al.* (2020).

*Artifact Size*

Table 8.7 report emitted binary size relative to `-O3`. Under this metric, $\mathcal{BO}3$ produces smaller binaries than `-O3` on several workloads, while increasing size on others. These measurements are not presented as a primary optimization target, but as a practical characterization of the method's cost profile.

*Conclusion*

Taken together, these results support the robustness of the learned checkpoints and the validity of the 3-stage design. They also clarify the intended use case of $\mathcal{BO}3$: offline autotuning for repeatedly executed inference workloads, where the search cost can be amortized against runtime gains.

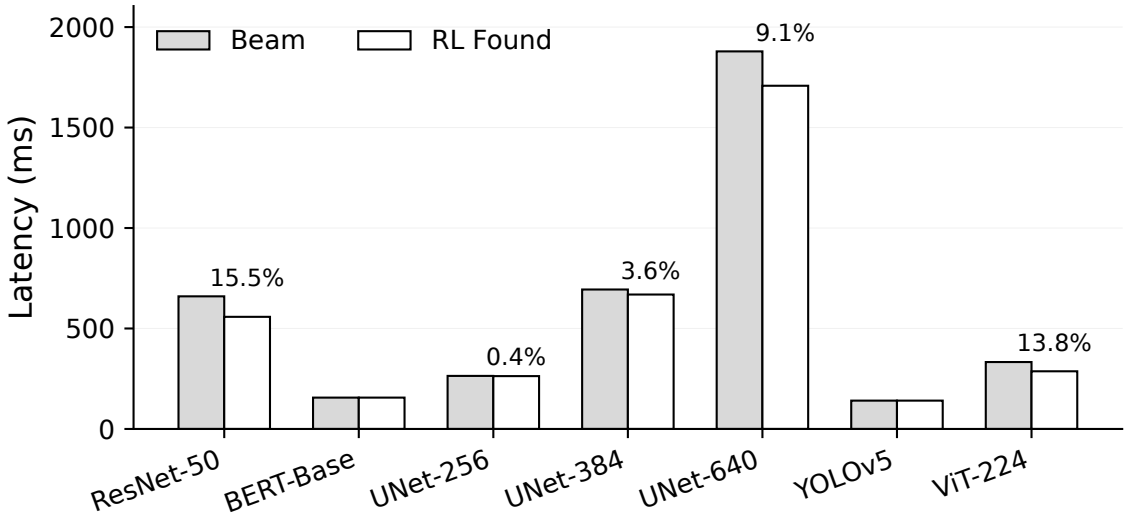



Figure 8.1: RL vs. beam search. On shallow tasks, both tie; on harder tasks, RL reaches regimes that beam search does not discover.

Table 8.1: Main evaluation results across 27 knob-conditioned optimization tasks (9 workload instances × 3 knob anchors). **O3**: default compiler pipeline. **Knob-only**: latency under the same knob anchor without additional pass search. **Random (ms)**: best valid candidate found by legality-constrained random rollout under a fixed post-training evaluation budget. **RL Found**: best valid candidate found by bounded post-training rollout sampling from the trained knob-conditioned policy checkpoint under the same evaluation budget. **Random/RL Spd.**: speedups over `-O3`. **Rollout win rate**: fraction of post-training policy rollouts that beat `-O3` under the same knob anchor. Knobs are denoted as tuples **(U, T, P, V)** representing Unroll, Tile, Vector PProc (m=mask, p=peel, n=none), and Vectorization.

| Model | O3 (ms) | Knob Tuple (U, T, P, V) | Knob only | Random (ms) | RL Found (ms) | Random Spd. | RL Spd. | Rollout win rate |
|---|---|---|---|---|---|---|---|---|
| 3*AlexNet | 3*256 ± 1.91 | Top-1 (T, T, m, F) | 246 ± 0.51 | 55.1 ± 0.49 | **20.5 ± 0.05** | 4.65× | **12.49**× | 100% |
| | | Top-2 (T, T, m, T) | 246 ± 0.61 | 234 ± 0.44 | 20.5 ± 0.03 | 1.09× | 12.49× | 80% |
| | | Top-3 (T, F, m, F) | 247 ± 0.51 | 207 ± 0.32 | 23.7 ± 0.05 | 1.03× | 10.80× | 100% |
| 3*ResNet-50 | 3*1592 ± 4.45 | Top-1 (F, F, m, T) | 1548 ± 1.54 | 659 ± 3.77 | **558 ± 2.17** | 2.42× | **2.85**× | 100% |
| | | Top-2 (T, F, m, T) | 1549 ± 3.43 | 661 ± 2.79 | 560 ± 2.51 | 2.41× | 2.84× | 90% |
| | | Top-3 (F, F, m, F) | 1555 ± 1.02 | 660 ± 2.87 | 564 ± 2.90 | 2.41× | 2.82× | 70% |
| 3*BERT-Base | 3*210 ± 0.61 | Top-1 (F, T, p, T) | 190 ± 0.50 | 156 ± 0.79 | **156 ± 0.34** | 1.35× | **1.35**× | 100% |
| | | Top-2 (F, T, n, T) | 190 ± 0.65 | 157 ± 1.02 | 157 ± 0.42 | 1.35× | 1.35× | 100% |
| | | Top-3 (F, T, n, F) | 191 ± 1.02 | 158 ± 0.62 | 158 ± 0.59 | 1.35× | 1.35× | 100% |
| 3*UNet: 256x256 | 3*2524 ± 6.73 | Top-1 (T, T, n, F) | 2513 ± 5.91 | 2336 ± 1.80 | **263 ± 0.7** | 1.08× | **9.60**× | 100% |
| | | Top-2 (T, T, n, T) | 2516 ± 7.11 | 263 ± 0.94 | 264 ± 0.46 | 9.60× | 9.56× | 100% |
| | | Top-3 (T, T, m, F) | 2515 ± 3.25 | 2337 ± 6.01 | 264 ± 0.95 | 1.08× | 9.56× | 100% |
| 3*UNet: 384x384 | 3*5672 ± 10.10 | Top-1 (T, F, n, F) | 5651 ± 9.75 | 4543 ± 1.57 | **669 ± 3.37** | 1.25× | **8.48**× | 100% |
| | | Top-2 (T, T, n, T) | 5688 ± 16.90 | 5667 ± 13.80 | 683 ± 3.12 | 1.00× | 8.30× | 100% |
| | | Top-3 (F, T, n, F) | 5683 ± 12.60 | 5899 ± 2.63 | 683 ± 2.62 | 0.96× | 8.30× | 100% |
| 3*UNet: 640x640 | 3*18067 ± 29.00 | Top-1 (T, F, p, T) | 16315 ± 18.20 | 14639 ± 4.01 | **1708 ± 4.25** | 1.23× | **10.58**× | 100% |
| | | Top-2 (F, T, p, T) | 16365 ± 23.00 | 15893 ± 8.05 | 1968 ± 1.90 | 1.13× | 9.18× | 90% |
| | | Top-3 (T, T, p, T) | 16366 ± 15.90 | 14636 ± 2.57 | 1971 ± 2.76 | 1.23× | 9.17× | 100% |
| 3*YOLOv5 | 3*1108 ± 3.57 | Top-1 (T, F, p, F) | 1061 ± 3.08 | 993 ± 3.57 | **141 ± 0.50** | 1.12× | **7.86**× | 100% |
| | | Top-2 (T, F, p, T) | 1061 ± 3.53 | 1097 ± 2.59 | 141 ± 0.56 | 1.01× | 7.86× | 100% |
| | | Top-3 (F, F, p, T) | 1056 ± 4.20 | 142 ± 0.80 | 141 ± 0.60 | 7.80× | 7.86× | 100% |
| 3*ViT-224 | 3*1559 ± 2.60 | Top-1 (F, T, p, T) | 319 ± 1.24 | 321 ± 0.965 | **287 ± 1.93** | 4.86× | **5.43**× | 80% |
| | | Top-2 (T, T, p, T) | 320 ± 0.94 | 320 ± 0.68 | 291 ± 1.24 | 4.87× | 5.36× | 80% |
| | | Top-3 (F, T, p, F) | 320 ± 0.57 | 293 ± 1.23 | 291 ± 1.10 | 5.32× | 5.36× | 100% |
| 3*DenseNet | 3*5980 ± 4.14 | Top-1 (T, T, p, F) | 4881 ± 4.65 | 4813 ± 4.65 | **1764 ± 1.79** | 1.27× | **3.39**× | 100% |
| | | Top-2 (F, T, p, T) | 4877 ± 4.23 | 4768 ± 4.91 | 1765 ± 2.14 | 1.25× | 3.39× | 100% |
| | | Top-3 (T, T, p, T) | 4888 ± 7.17 | 4881 ± 2.14 | 1767 ± 3.71 | 1.22× | 3.38× | 100% |

Table 8.2: Component ablation on UNet-640 Top-1. Removing both learned components reduces RL to beam-search-level performance.

| Configuration | Best lat. (ms) | Win rate |
|---|---|---|
| Full BO3 | **1708** | 100% |
| No legality masking | 1708 | 76% |
| No graph encoder | 1876 | 40% |
| No masking + no graph | 1879 | 30%‘ |
| Beam search | 1879 | — |

Table 8.3: UNet-640 case study under the Top-1 knob anchor. Random rollout, beam search, and policy-guided rollout from the trained checkpoint induce qualitatively different downstream regimes.

| **Task** | **Method** | **Found sequence** | **Lat. (ms)** | **#Disp.** | **Double%** | **Mem./Disp.** | **Perm./Disp.** | **FMA rate%** |
|---|---|---|---|---|---|---|---|---|
| 4*UNet-640 Top-1 | O3 / baseline | *default* | 18067 | 75 | 77.44 | 21.47 | 20.29 | 85.71 |
| | Random rollout | `T + EVAL` | 14819 | 77 | 81.12 | 22.38 | 30.64 | 78.32 |
| | Beam search | `I + cse + EVAL` | 1879 | 186 | 75.49 | 27.75 | 45.79 | 92.93 |
| | **BO3** | `canonicalize + CC + I + EVAL` | **1708** | 183 | **88.74** | **52.62** | **147.26** | **96.69** |

Table 8.4: UNet-640 dispatch composition by semantic type under three search procedures (Top-1 knob anchor). All dispatches use the double-tiling path; differences reflect dispatch *decomposition*, not routing.

| Method | Matmul | Generic (rearrange) | Memcpy (slow) | Broadcast | Transpose | Reduction | Others | Total |
|---|---|---|---|---|---|---|---|---|
| Beam | 47 | 47 | 44 | 41 | 6 | 0 | 1 | 186 |
| BO3 | 44 | 44 | 42 | 35 | 14 | 3 | 1 | 183 |

Table 8.5: Execution-side support for the UNet-640 case study. RL further improves on beam by reducing translation/stall pressure while greatly increasing packed SIMD activity.

| **Method** | **CPU cycles** | **Instructions** | **IPC** | **TLB misses** | **Not-complete** | **Pack256 Uops** |
|---|---|---|---|---|---|---|
| Random rollout | 368B | 1T | 3.83 | 311M | 198B | 367M |
| Beam search | 108B | 124B | 1.15 | 6B | 73B | 813M |
| **BO3** | **96B** | **156B** | **1.6** | **1B** | **57B** | **22B** |

Table 8.6: Aggregate evaluation summary. Scope analysis covers 15 workloads, while main evaluation uses 9 workload instances from 7 model families, yielding 27 knob-conditioned tasks.

| **Metric** | **Value** |
|---|---|
| Scope-analysis workloads | 15 |
| Main-evaluation workload instances | 9 |
| Knob-conditioned optimization tasks | 27 |
| Task-level improvement rate (vs. `-O3`) | 100% (27/27) |
| Geometric-mean speedup | 5.66× |
| Mean rollout win rate | 95.93% |

Table 8.7: Per-family Stage II training cost and emitted binary size ratio. Size ratio is computed as `RL` / `-O3` using the emitted `*.so` artifact. Ratios below 1 indicate a smaller binary than `-O3`.

| Workload family | Mean training time (h) | Binary size ratio |
|---|---|---|
| AlexNet | 5.20 | 1.44 |
| ResNet50 | 3.30 | 0.13 |
| BERT | 6.93 | 0.89 |
| UNet-256 | 6.47 | 1.32 |
| UNet-384 | 8.70 | 0.47 |
| UNet-640 | 11.97 | 1.18 |
| YOLOv5 | 2.87 | 0.62 |
| ViT-224 | 5.30 | 0.55 |
| DenseNet | 11.64 | 1.73 |

# Chapter 9

# CONCLUSION AND FUTURE WORK

## Conclusion

This thesis studied compiler pass ordering in MLIR-based progressive lowering, with a particular focus on the IREE compiler. The central argument of this work is that pass ordering in this setting is difficult not only because optimization effects are delayed, but because pass reordering can induce *tuning-space regime shifts*: discrete downstream changes in dispatch structure, layout behavior, and backend routing that alter the optimization space seen by later compilation stages.

To motivate this view, the thesis first presented empirical evidence along three complementary axes. Across model architectures, the identity of the best pass pipeline changed substantially, showing that a pipeline that is highly effective for one workload may be weak or even invalid for another. Within a fixed model family, this fragility persisted across tensor shapes, demonstrating that pass ordering is shape-sensitive rather than merely architecture-sensitive. Across lowering stages, the same transformation could help, do little, or hurt depending on where it was inserted, confirming that profitability in progressive lowering is fundamentally stage-dependent. Taken together, these observations showed that fixed pipelines and local heuristics are poorly matched to the structural and long-horizon nature of pass ordering in progressive-lowering compilers.

Guided by these observations, this thesis presented $\mathcal{BO}3$, a legality-aware bi-level reinforcement learning framework for automated pass ordering. The framework decomposes the optimization problem into two levels. Stage I performs coarse screening

over backend code-generation knobs to identify promising regime anchors, and Stage II trains a separate legality-constrained reinforcement learning policy for each retained anchor to search over preprocessing-stage pass sequences. To make this search feasible in a compiler environment, the framework combines graph-structured SSA state, dynamic legality masking, and sparse terminal reward aligned with end-to-end latency.

Evaluation across 9 workload instances from 7 model families, yielding 27 knob-conditioned optimization tasks, showed that $\mathcal{BO}3$ consistently improves over the default `-O3` pipeline under a bounded-budget evaluation protocol. The results further showed that the value of learning is not uniform across all tasks. On easier tasks, structured non-learning search procedures such as beam search can already reach high-quality candidates. On harder tasks with stronger regime-shift behavior, however, policy-guided search more reliably reaches better-performing legal trajectories under the same bounded search effort. The UNet-640 case study further showed that the strongest candidates are better understood as entering different downstream execution regimes than as monotonically improving any single local proxy.

Overall, this thesis does not claim that one policy transfers universally across workloads, nor that reinforcement learning is always necessary for every pass-ordering task. Its contribution is instead a reproducible optimization procedure that combines legality-aware policy learning, structured compiler-state representation, and backend-regime conditioning to discover workload-specific pass sequences in progressive-lowering compilers. More broadly, the thesis argues that pass ordering in MLIR-based compilation is best understood as a regime-sensitive structural optimization problem rather than as a static ranking problem over locally profitable transformations.

## Technical Insights

Beyond the specific $\mathcal{BO}3$ framework, this thesis suggests several broader technical insights about compiler optimization under progressive lowering.

First, *representation choice is itself an optimization variable.* In a multi-level compiler, the question is not only which transformation to apply, but also which representation should remain available to later stages. The stage-placement study showed that semantically similar transformations can lead to qualitatively different downstream effects depending on when they are applied. This suggests that phase ordering in MLIR is inseparable from representation management.

Second, *good pass sequences often work by changing what later stages are able to optimize.* This is a stronger statement than saying that passes interact. The evidence in this thesis suggests that the most important optimization effect is often not a local improvement in the current IR, but a change in the downstream regime itself. This helps explain why local profitability proxies are frequently unreliable in progressive-lowering settings.

Third, *legality is part of the optimization structure, not merely a constraint to check afterward.* In conventional compiler search formulations, legality is often treated as a side condition. In this work, legality evolved with the IR and shaped the effective search space at every step. This indicates that, in progressive-lowering compilers, legality modeling should be treated as a first-class part of optimization design.

Fourth, *compiler state is inherently structured.* The usefulness of graph-based observations in $\mathcal{BO}3$ reflects that pass effects depend on non-local producer-consumer structure, layout propagation, and the decomposition of computation into dispatches. This makes structured state representations especially natural for compiler optimization in MLIR-like systems.

## Limitations

Several limitations of the current work should be stated explicitly.

First, the current framework does not attempt zero-shot transfer across heterogeneous workloads. Each evaluated task trains a separate knob-conditioned policy checkpoint. This design improves procedural clarity and reduces interference across regimes, but it also incurs substantial per-task training cost and limits immediate reuse across workloads.

Second, the search space is intentionally restricted to preprocessing-stage pass ordering. This choice is well motivated by the strong downstream influence of early transformations, but it leaves the full multi-stage phase-ordering problem unresolved. The present system does not reason jointly about *which stage* to intervene in and *which pass* to apply there.

Third, the empirical claims are tied to a CPU-oriented IREE setting, a particular backend knob space, and the studied benchmark suite. The regime-shift perspective is likely broader than this single instantiation, but additional evidence would be needed before generalizing the quantitative claims to GPUs, accelerators, or substantially different compiler pipelines.

Fourth, $\mathcal{BO}3$ is evaluated as a bounded-budget candidate generator rather than as a deterministic single-shot optimizer. This is an intentional methodological choice, but it means that final result quality still depends on post-training rollout sampling and evaluation budget. In this sense, the framework is better understood as an offline autotuning procedure than as a standard always-on compile-time optimization pass.

Fifth, the current explanatory tools are still incomplete. Although this thesis uses structural diagnostics and hardware-counter evidence to interpret regime shifts, it does not yet provide a full predictive theory of when a given pass sequence will

induce a favorable downstream regime. The current analysis supports the regime-shift hypothesis, but does not fully solve the explanatory problem.

## Future Work

Several research directions follow naturally from the technical structure of this thesis.

### *Cross-Task Transfer and Warm-Start Learning*

The current one-task-one-policy design avoids interference across workloads, but it leaves open whether parts of the learned representation can be reused. A promising direction is to separate *general compiler knowledge* from *task-specific decision making*. For example, future work could pretrain a graph encoder across many workloads, then fine-tune only small policy heads for individual tasks. A stronger variant would be to explore meta-learning or retrieval-based warm starts, where a new workload initializes from previously tuned workloads with similar structural signatures.

### *Multi-Stage Intervention*

This thesis focused on preprocessing-stage pass ordering because early transformations have strong leverage over downstream dispatch formation and backend routing. However, the stage-placement study suggests a broader problem: profitability depends not only on pass identity, but also on *where in lowering* the pass is inserted. A natural extension is therefore to treat stage choice itself as part of the action space. Such a framework would move from "which pass next?" to "which pass, at which stage, under what current representation?"

*Joint Structural Search Beyond Pass Ordering*

The current work searches over pass sequences under a fixed pass vocabulary. A broader extension would be to search jointly over multiple structural compiler decisions, including pass ordering, representation choice, dispatch-formation policy, and backend-expert routing hints. This would move the framework closer to a unified structural autotuner for progressive-lowering compilers.

*Broader Backend and Hardware Coverage*

The present study focuses on CPU-oriented IREE workflows. An important next step is to test whether similar regime-shift phenomena arise on GPU and accelerator backends. This is especially interesting because such targets expose different lowering experts, memory hierarchies, layout sensitivities, and hardware primitives. A useful question is whether the same high-level regime-shift framework remains valid when the relevant downstream distinction is no longer Double-tiling versus MMT4D, but, for example, tensor-core utilization, shared-memory tiling, or asynchronous pipeline structure.

*Hybrid Search Strategies*

The evaluation suggests that learning is not equally necessary across all tasks. On easier tasks, beam search can already be competitive; on harder tasks, policy guidance becomes more valuable. This motivates hybrid strategies that adapt the search procedure to task difficulty. For instance, policy scores could prioritize beam expansion, or a system could begin with symbolic search and switch to learned guidance only when structural signals indicate that the profitable region is sparse or difficult to reach.

### *Richer Compiler-State Modeling*

The current graph representation captures SSA structure, operation categories, layout indicators, and coarse computational features. Future work could incorporate richer dependence information, dispatch graphs, aliasing or memory-region structure, and backend-side feedback features. Another promising direction is hierarchical state modeling: instead of only representing operations, the state could include both operation-level graphs and dispatch-level graphs, allowing the policy to reason at multiple structural scales.

### *Cost-Aware and Multi-Objective Optimization*

This thesis reports training time and artifact size as practical characterization metrics, but does not optimize for them directly. In deployment settings, users may care simultaneously about latency, tuning cost, binary size, energy, memory footprint, or reproducibility. Future systems could therefore formulate the problem as multi-objective optimization rather than latency-only search. This may be especially important if $\mathcal{BO}$3-like systems are to be used in resource-constrained deployment pipelines.

### *Toward Explanatory Compiler Autotuning*

One of the main lessons of this thesis is that strong pass sequences are often better described as inducing a different downstream regime than as locally simplifying IR. A major next step is therefore to build models and diagnostics that can explain *why* a candidate enters a better regime and *when* such a transition is likely to occur. Such explanations would be scientifically useful, but they would also improve practical trust in learned compiler optimizers by making the system's decisions more interpretable

to compiler engineers.

## Closing Remarks

Modern MLIR-based compilers increasingly operate in settings where optimization is Modern MLIR-based compilers operate in highly structural and workload-dependent environments. In this setting, treating pass ordering as a static heuristic or a purely local simplification is no longer sufficient. By reframing pass ordering as a legality-constrained search over regime-shifting transformations, this thesis demonstrated that automated optimization can effectively adapt to these complex conditions.

Ultimately, the value of this work extends beyond any specific reinforcement learning framework or compiler stack. It underscores a necessary shift in how we approach progressive-lowering systems: moving away from static tuning over fixed representations, and toward dynamically shaping the downstream optimization regimes themselves.

# REFERENCES


Agakov, F., E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson and C. Williams, "Using machine learning to focus iterative optimization", in "Proceedings of the International Symposium on Code Generation and Optimization (CGO)", (2006), URL https://dl.acm.org/doi/10.1109/CGO.2006.37.

Ansel, J., S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning", in "Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)", (2014), URL https://dl.acm.org/doi/10.1145/2628071.2628092.

Chen, T., T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning", in "Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)", (2018), URL https://www.usenix.org/conference/osdi18/presentation/chen.

CIRCT Team, "CIRCT tutorial / overview", https://discourse.llvm.org/t/circt-tutorial/80442, accessed 2026-03-05 (2026).

Hamilton, W. L., Z. Ying and J. Leskovec, "Inductive representation learning on large graphs", in "Advances in Neural Information Processing Systems (NeurIPS)", vol. 30 (2017).

Huang, S. and S. Ontanón, "A closer look at invalid action masking in policy gradient algorithms", arXiv preprint arXiv:2006.14171 URL https://arxiv.org/abs/2006.14171 (2022).

Kulkarni, S. and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning", ACM SIGPLAN Notices **47**, 10, 147–162 (2012).

Liu, H.-I. C., M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik and S. Laurenzo, "TinyIREE: An ML execution environment for embedded systems from compilation to deployment", arXiv preprint arXiv:2205.14479 URL https://arxiv.org/abs/2205.14479 (2022).

LLVM Project, "Torch-MLIR project", https://github.com/llvm/torch-mlir, accessed 2026-03-05 (2026a).

LLVM Project, "Users of MLIR", https://mlir.llvm.org/users/, accessed 2026-03-05 (2026b).

Mullapudi, R. T., A. Adams, D. Sharlet, J. Ragan-Kelley and K. Fatahalian, "Automatically scheduling Halide image processing pipelines", ACM Transactions on Graphics (Proc. SIGGRAPH) **35**, 4, URL https://dl.acm.org/doi/10.1145/2897824.2925952 (2016).

Raffin, A., A. Hill, A. Gleave, A. Kanervisto, M. Ernestus and N. Dormann, "Stable-Baselines3: Reliable reinforcement learning implementations", Journal of Machine Learning Research **22**, 268, 1–8, URL http://jmlr.org/papers/v22/20-1364.html (2021).

Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal policy optimization algorithms", (2017), URL https://arxiv.org/abs/1707.06347.

Trofin, M., Y. Qian, E. Brevdo, Z. Lin, K. Choromanski and D. Li, "MLGO: A machine-learning guided compiler optimizations framework", (2021), URL https://arxiv.org/abs/2101.04808.

Zheng, L., C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning", in "Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)", (2020), URL https://www.usenix.org/conference/osdi20/presentation/zheng.