UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Douglas Viana Coutinho

Transfer Learning Across MLIR Dialects

Belo Horizonte
2026

Douglas Viana Coutinho

Transfer Learning Across MLIR Dialects

Submitted Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira
Co-Advisor: Vanderson Martins do Rosário

Belo Horizonte
2026

To my family, for their unwavering support throughout this journey.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Fernando Magno Quintão Pereira, and my co-advisor, Vanderson Martins do Rosário, for their guidance, expertise, encouragement, and constant support throughout this research. Their insights, feedback, and mentorship were invaluable to the completion of this work.

I am deeply grateful to my family for their love, encouragement, and unwavering support throughout my life. This achievement would not have been possible without them.

I would also like to thank the colleagues and friends I met during my academic journey for their companionship, support, and the many hours of discussion, study, and collaboration that made this experience both productive and enjoyable. I am especially grateful to the people in the Compilers Lab and the XNNC team at Cadence Design Systems for their support, valuable discussions, and feedback during this work.

Finally, I would like to thank the Federal University of Minas Gerais (UFMG), as well as its faculty and staff, for providing an environment that values knowledge, scientific inquiry, and academic excellence. The experiences and opportunities I received during my time here have been fundamental to both my professional and personal development, and will remain with me for years to come.

"*The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.*"

(Edsger W. Dijkstra)

Resumo

Aprendizado de máquina tem sido aplicado com sucesso na resolução de tarefas de compilação. No entanto, o treinamento desses modelos exige grandes conjuntos de dados de referência. Essa dependência cria um gargalo em infraestruturas de compiladores como o framework de Representação Intermediária Multi-Nível (MLIR), pois dialetos específicos de domínio frequentemente carecem de dados de treinamento em larga escala. Para contornar essa limitação, demonstramos que a transferência de aprendizado entre dialetos é viável e apresentamos uma cadeia de ferramentas de código aberto para auxiliar desenvolvedores de compiladores MLIR a aplicar essa mesma metodologia em seus próprios compiladores. Utilizando representações de grafos ProGraML e Redes Neurais em Grafos (GNNs), pré-treinamos um modelo em um dialeto base com abundância de dados e o adaptamos para um dialeto-alvo com poucos dados disponíveis. O ponto central para a eficácia dessa metodologia está em identificar quais componentes da GNN podem ser mantidos fixos entre os dialetos e quais precisam ser ajustados para acomodar diferenças semânticas. Avaliamos esse pipeline prevendo o tempo de treinamento e a acurácia de teste de 423.624 arquiteturas de redes neurais do NASBench. Para simular um cenário realista com restrição de dados, pré-treinamos o modelo em 228.757 grafos `stablehlo` e o refinamos com 25.417 grafos `linalg`. O modelo `stablehlo-linalg` resultante alcança um RMSE aproximadamente 50% menor do que um modelo treinado exclusivamente no conjunto limitado de dados `linalg`, além de reduzir o tempo de convergência.

Palavras-chave: MLIR. ProGraML. Redes Neurais em Grafos. Aprendizado por Transferência. Adaptação de Domínio.

Abstract

Machine learning has been successfully used to solve compilation tasks. However, training these models requires large benchmark datasets. This dependency creates a bottleneck in compiler infrastructures such as the Multi-Level Intermediate Representation (MLIR) framework, because domain-specific dialects often lack large-scale training data. To address this limitation, we show that cross-dialect transfer learning is possible and introduce an open-source toolchain to help MLIR compiler developers use the same methodology in their compilers. Using ProGraML graph representations and Graph Neural Networks (GNNs), we pre-train a model on a data-rich baseline dialect and adapt it to a data-poor target dialect. Key to the effectiveness of this methodology is identifying which GNN components can remain fixed across dialects and which require tuning to adapt to semantic differences. We evaluate this pipeline by predicting the training time and test accuracy of 423,624 NASBench neural network architectures. To simulate a realistic data-constrained scenario, we pre-train the model on 228,757 `stablehlo` graphs and fine-tune it on 25,417 `linalg` graphs. The resulting `stablehlo-linalg` model achieves roughly 50% lower RMSE than a model trained solely on the limited `linalg` dataset, while reducing convergence time.

Keywords: MLIR. ProGraML. Graph Neural Networks. Transfer Learning. Domain Adaptation.

List of Figures

List of Tables

List of Abbreviations and Acronyms

DSL	Domain Specific Language
FTE	Fine-tuning Experiment
GGNN	Gated Graph Neural Network
GNN	Graph Neural Network
GRU	Gated Recurrent Unit
IR	Intermediate Representation
MLP	Multilayer Perceptron
MLIR	Multi-Level Intermediate Representation
ProGraML	Program Graphs for Machine Learning
RMSE	Root Mean Squared Error
SSA	Static Single-Assignment

Contents

Chapter 1

Introduction

Machine learning has become an important tool for program analysis and optimization, enabling compilers to learn complex structural and semantic patterns directly from code [Ashouri et al., 2018; da Silva et al., 2021; Fursin et al., 2008; Kulkarni and Cavazos, 2012; Leather and Cummins, 2020; Monsifrot et al., 2002; Stephenson et al., 2003; Trofin et al., 2021]. Stochastic models have been successfully applied to tasks including optimization heuristics [da Silva et al., 2022] and performance prediction [Cavazos and Moss, 2004; Wang and O'Boyle, 2010]. These approaches are effective because they bypass manually designed analytical models and learn directly from large collections of programs.

However, this effectiveness faces two challenges: *availability* and *stability*. The first challenge concerns the availability of training data for machine learning models. This requirement is difficult to satisfy in domain-specific languages (DSLs) and in customizable compiler infrastructures such as the MLIR [Lattner et al., 2021], where new dialects can be created rapidly. As a result, many dialects lack sufficiently large datasets to train accurate stochastic models. The second challenge concerns the stability of these representations. Domain-specific dialects evolve frequently because they serve smaller and more specialized communities, making changes cheaper to introduce. This issue is again particularly evident in the MLIR ecosystem, where several dialects are explicitly designed for extensibility [Lücke et al., 2025]. Even small modifications to an intermediate representation, such as adding or removing operations or types, may invalidate previously trained models and require costly retraining, despite the strong semantic similarity between the old and new representations.

Contributions. This thesis shows that machine learning models can be trained to solve compilation tasks for MLIR dialects that lack large benchmark suites, provided that related benchmarks exist for other, widely used dialects. To enable this transfer, we combine two previous techniques: ProGraML [Cummins et al., 2021], a graph-based representation of programs, and Graph Neural Networks (GNNs) [Scarselli et al., 2009]. In our approach, ProGraML acts as a normalization layer for MLIR, providing a common structural representation that abstracts away many dialect-specific details

while preserving the essential data-flow and control-flow relationships of programs. A GNN is then trained on this representation to learn general structural properties from a data-rich *source dialect*.

The key insight of this work is that it is possible to apply the well-established transfer learning methodology [Pan and Yang, 2010] across MLIR dialects. Transfer learning adapts a model trained on a data-rich *source domain* to a related but data-scarce *target domain*. In our setting, the modular structure of GNNs provides a natural mechanism to support this adaptation by separating dialect-independent knowledge from dialect-specific behavior. Chapter 3 shows that, when adapting a model to a data-scarce *target dialect*, only the components that depend on dialect semantics need to be retrained, while the remaining components can be kept fixed. This design preserves the structural knowledge learned from the source dialect while enabling efficient specialization to the target dialect using only a small number of examples.

Summary of Results: The proposed methodology has been implemented within the MLIR ecosystem. Chapter 4 discusses its evaluation on two different problems: predicting the *training time* and the *test accuracy* of neural network architectures. This evaluation happens over thousands of models present in the NASBench [Ying et al., 2019] suite. Specifically, we demonstrate accurate predictions by pre-training a model on 228,757 programs given in the `stablehlo` dialect of MLIR and fine-tuning it on 25,417 programs from the `linalg` dialect. Our results show that the adapted model reduces errors by approximately 50% on the different problems compared to a model trained solely on the limited target dataset. This result indicates that, at least in the MLIR context, it is possible to solve compilation tasks even in brand new dialects that still lack benchmarks. This possibility follows from three contributions, which this work brings forward:

- **Cross-Dialect Transfer Learning:** The thesis establishes a methodology for repurposing models trained on data-rich MLIR dialects for dialects with limited benchmarking data, leveraging structural similarities maintained throughout the lowering process.
- **Fine-Tuning Strategy:** The thesis identifies architectural components of Graph Neural Networks that capture general program structure across dialects, demonstrating that adapting only dialect-specific layers bridges semantic gaps where zero-shot transfer fails.
- **Tooling:** This thesis provides an open-source transfer-learning pipeline that can be adapted to different source and target dialects with little effort, thanks to the structural uniformity imposed by MLIR across dialects, which ProGraML captures as a graph representation.

1.1 Background

The goal of this work is to show how a machine-learning model trained on a data-rich MLIR dialect can be adapted to a data-scarce dialect. This section provides intuition for the approach, explaining why and how knowledge learned from one dialect can be effectively transferred to another.

1.1.1 The Multi-Level Intermediate Representation

MLIR [Lattner et al., 2021] is a compiler infrastructure designed to represent, analyze, and transform programs through a hierarchy of intermediate representations. It presents many dialects used in various domains, including linear algebra and tensor computations, hardware design, and domain-specific optimizations for machine learning and high-performance computing. Example 1.1 discusses one such dialect.

Figure 1.1: An MLIR program in the `stablehlo` dialect.

```
01 module {
02   func.func @main(%arg0: tensor<1x3x5x5xf32>,
03                   %arg1: tensor<8x3x3x3xf32>,
04                   %arg2: tensor<1x8x3x3xf32>) -> tensor<1x8x3x3xf32> {
05
06     // Convolution: input %arg0 with kernel %arg1
07     %conv = stablehlo.convolution(%arg0, %arg1)
08       dim_numbers = [b, f, 0, 1] x [o, i, 0, 1] -> [b, f, 0, 1],
09       window = {stride = [1, 1], pad = [[0, 0], [0, 0]]}
10       : (tensor<1x3x5x5xf32>, tensor<8x3x3x3xf32>) -> tensor<1x8x3x3xf32>
11
12     // Elementwise multiplication
13     %mul = stablehlo.multiply %conv, %arg2 : tensor<1x8x3x3xf32>
14
15     // Elementwise addition
16     %add = stablehlo.add %mul, %conv : tensor<1x8x3x3xf32>
17
18     // Elementwise maximum
19     %max = stablehlo.maximum %add, %mul : tensor<1x8x3x3xf32>
20
21     return %max : tensor<1x8x3x3xf32>
22   }
23 }
```

Source: Created by the author.

Example 1.1. The program in Figure 1.1 shows a simple MLIR program in the `stablehlo` dialect. The program takes an input tensor and a convolution kernel, computes their convolution, and then applies a sequence of elementwise operations. First, the result of the convolution is multiplied by a second tensor. The program then adds the original convolution result to this product, and finally computes the elementwise maximum between these two intermediate values. The overall computation resembles a small neural-network fragment combining convolution, scaling, and a non-linear activation.

1.1.2 Stochastic Compilation Tasks

Programming language processing systems, such as compilers, analyzers, and interpreters, increasingly rely on stochastic models to solve language-related tasks. This trend is not new: it can already be observed in the work of John Cavazos over 20 years ago [Cavazos and Moss, 2004; Cavazos and O'Boyle, 2005]. However, it has experienced a revival in recent years [Ashouri et al., 2018; Wang and O'Boyle, 2018]. Today, a variety of compilation tasks are addressed using stochastic models, including profile prediction [Moreira et al., 2021], the selection of optimization sequences [da Silva et al., 2021, 2022], and the choice of hardware configurations [Wang and O'Boyle, 2010]. Example 1.2 illustrates one such task.

Example 1.2. Consider a concrete stochastic task: predicting the *training time* of neural network architectures expressed as MLIR programs. This is a practically important task because, as Chen et al. [2018] explains, a reliable predictor can guide architecture search, scheduling decisions, and optimization pipelines without fully training every candidate model. To solve this problem using machine learning, we require a large and representative corpus of programs paired with ground-truth measurements. A typical solution represents each program as a structured object. For instance, a graph derived from its MLIR representation (e.g., Figure 1.1). This representation is then used as input to a predictive model, such as a neural network, which learns to associate program structure and operations with execution costs. Once trained, the model can estimate the training time of unseen programs directly from their representation, avoiding expensive full executions.

1.1.3 The Data Availability Challenge

As already mentioned before, solving compilation tasks via stochastic models requires training data: collections of programs paired with observations about their behavior, such as execution time, resource usage, or other properties of interest. However, obtaining such datasets can be difficult, particularly for new or specialized languages or intermediate representations that suffer frequent updates. In the presence of limited data, stochastic models tend to produce less accurate predictions, which in turn affects the quality of the compilation decisions they support. Example 1.3 illustrates this limitation.

Example 1.3. Suppose we wish to train a Graph Neural Network (GNN) to predict training time, as discussed in Example 1.2. If we train the setup evaluated in Chapter 4 using a dataset of 4,000 programs in the `linalg` dialect of MLIR, we obtain a Kendall's

τ of approximately 0.65 on the prediction task. This metric captures the correlation between predicted and observed rankings: values close to one indicate near-perfect agreement, whereas values close to zero indicate little or no predictive power. Thus, $\tau = 0.65$ suggests that the model captures some useful trends, but still makes many incorrect comparisons. This performance is significantly below what can be achieved with larger datasets. When trained on 400,000 programs, the same model attains $\tau > 0.97$, approaching near-perfect ranking accuracy.

The gap between what a data-starved model can learn and what is ultimately achievable defines the problem addressed in this work. Example 1.3 illustrates that, in the absence of sufficient data, models struggle to capture meaningful program properties. This limitation is pronounced in MLIR, where newly introduced dialects have a small number of available benchmarks. However, MLIR is far larger than any single dialect, and learning can be transferred among them!

1.1.4 Transfer Learning

MLIR encompasses a rich ecosystem of dialects, many of which are substantially better populated than others. Importantly, these dialects are not independent: they are often connected through the compilation pipeline, which progressively *lowers* programs from high-level abstractions to more concrete operations. As a result, different dialects may encode the same computation using different levels of detail, while still preserving similar underlying structure.

This relationship creates an opportunity for transfer learning. Intuitively, a model trained on a data-rich dialect can learn general properties of program structure that remain valid across dialect boundaries, and can therefore be reused when analyzing programs in a data-scarce dialect. This observation is central to our approach. A concrete instance of this relationship can be observed between the `stablehlo` and `linalg` dialects. Example 1.4 illustrates how the same computation can be expressed in both dialects.

Example 1.4. Figure 1.2 shows a side-by-side listing of MLIR code for the same convolution-scale-bias-ReLU block in the `stablehlo` (left) and the `linalg` (right) dialects. Both are used to represent neural network computations; however, `stablehlo` provides a higher-level, more compact representation based on domain-specific operations, whereas `linalg` expresses the same computation using lower-level primitives that make data movement and iteration structure explicit.

Figure 1.2: The same neural network fragment (a convolution, channel-wise scale, bias, and ReLU) expressed in `stablehlo` (left) and `linalg` (right).

```
%conv = stablehlo.convolution ...
```

```
%buf = tensor.empty(...)
%filled = linalg.fill(...)
%padded = tensor.pad(...)
%conv = linalg.conv_2d_nhwc_hwcf(...)
```

```
%mul = stablehlo.multiply ...
```

```
%sbcast = linalg.broadcast(...)
%scaled = linalg.map{arith.mulf}(...)
```

```
%add = stablehlo.add ...
```

```
%bbcast = linalg.broadcast(...)
%biased = linalg.map{arith.addf}(...)
```

```
%max = stablehlo.maximum ...
```

```
%zbcast = linalg.broadcast(...)
%relu = linalg.map{arith.maximumf}(...)
```

Source: Created by the author.

1.1.5 Knowledge Transfer Requires Method

A natural transfer strategy is *zero-shot* learning: for example, training a GNN entirely on `stablehlo` programs and applying it directly to `linalg` programs without any adaptation. In practice, however, this approach fails, as illustrated in Example 1.5. In this example, a model is trained on 400,000 programs written in `stablehlo` and then evaluated on 20,000 programs written in `linalg`, without any customization.

Example 1.5. As reported in Section 4.2.1, the zero-shot model achieves an RMSE of 5,645.5 and an $R^2 = -37.487$ on the training time prediction task, performing far worse than a trivial mean predictor (RMSE = 919.5, $R^2 = 0.0$). The strongly negative R^2 indicates that the model has internalized the output scale and distribution of `stablehlo`, which do not align with those of `linalg`. For instance, a convolution that takes X seconds in `stablehlo` corresponds to a representation that is substantially larger in `linalg`, leading to systematically miscalibrated predictions. Only the ranking signal partially survives ($\tau = 0.276$), suggesting that some structural knowledge is transferable in principle, even though the absolute predictions are unreliable.

In Example 1.5, `stablehlo` plays the role of the *source dialect*, while `linalg` is the *target dialect*. Because these notions are central to this work, Definition 1.6 formalizes them.

Definition 1.6 (Source and Target Dialects)**.** Given two MLIR dialects S and T such that programs in T are obtained by lowering those in S, we call S the *source dialect* and T the *target dialect*. We assume that a large labeled corpus exists for S, but only a small labeled corpus is available for T.

This thesis shows that although zero-shot transfer fails, the structural knowledge learned from the source dialect can be adapted to the target dialect. Chapter 3 presents

a methodology to achieve this adaptation by retaining the components of the model that capture general program structure, while fine-tuning only those components that are sensitive to dialect-specific features using a small target corpus.

1.2 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 reviews the relevant literature, covering the history of machine learning in compilers, the MLIR compiler infrastructure, program representations for machine learning, Graph Neural Networks for code analysis, and transfer learning and domain adaptation. Chapter 3 presents the proposed methodology, detailing the MLIR-to-ProGraML conversion pipeline and the GNN fine-tuning strategy. Chapter 4 describes the experimental setup and evaluates the methodology across seven research questions. Chapter 5 concludes with a summary of findings, a discussion of limitations, and directions for future work.

Chapter 2

Literature Review

The application of machine learning to compiler optimization is a research direction with over two decades of history. Since the mid-1990s, researchers have explored the use of statistical and learning-based techniques to address two fundamental problems in optimizing compilers: *optimization selection*, which concerns choosing which transformations to apply to a given program, and *phase ordering*, which concerns determining the sequence in which those transformations should be applied [Ashouri et al., 2018]. These problems are difficult because the effect of a given optimization depends on the target program, the hardware platform, and the interactions among previously applied transformations. The resulting search space grows combinatorially, rendering exhaustive exploration infeasible and motivating the use of data-driven approaches to learn effective heuristics from empirical observations [Wang and O'Boyle, 2018].

The evolution of this field can be understood along two complementary axes: the *models* used to make optimization decisions, and the *representations* used to characterize programs. Early work relied on classical machine learning methods such as logistic regression [Cavazos and Moss, 2004], decision trees [Monsifrot et al., 2002], and genetic algorithms [Stephenson et al., 2003], operating on hand-crafted feature vectors extracted from source code or compiler intermediate representations [Fursin et al., 2008]. These approaches demonstrated that learned heuristics could match or surpass the performance of manually tuned compiler strategies, establishing the viability of the paradigm. Over time, the field progressed toward richer representations and more expressive models. The introduction of learned embeddings for compiler instructions [Ben-Nun et al., 2018] and graph-based program representations such as ProGraML [Cummins et al., 2021] replaced manually designed features with continuous vector spaces that capture structural and semantic properties of programs. In parallel, deep learning architectures, including Graph Neural Networks [Li et al., 2015; Scarselli et al., 2009], enabled models to reason directly over the relational structure of code, leading to improved performance on tasks such as device mapping [Cummins et al., 2017], throughput prediction [Mendis et al., 2019a], and vectorization [Mendis et al., 2019b]. As Leather and Cummins [2020] observe, a defining advantage of deep learning in this context is that large neural networks can operate on raw or lightly pro-

cessed program representations, shifting the burden from manual feature engineering to end-to-end learning.

Despite these advances, a recurring limitation of machine-learning-based compilation is its dependence on large labeled datasets [Moreira et al., 2021; Wang and O'Boyle, 2018]. Most existing work assumes that a sufficiently large and representative corpus of programs, paired with ground-truth measurements, is available for training. This assumption is difficult to satisfy in modern compiler infrastructures such as MLIR [Lattner et al., 2021], where new domain-specific dialects can be introduced rapidly and often lack established benchmark suites. The challenge is compounded by the instability of these representations: as dialects evolve, previously collected data may become outdated, requiring costly re-collection and retraining. This thesis addresses the data-scarcity bottleneck by introducing a cross-dialect transfer learning methodology, which draws on foundational work in transfer learning [Pan and Yang, 2010] and domain adaptation [Kirkpatrick et al., 2017] to repurpose models trained on data-rich dialects for use in data-scarce ones.

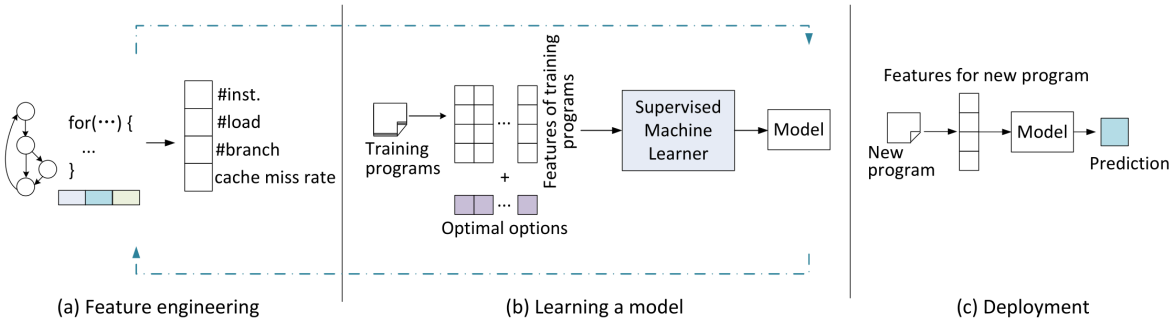
This chapter surveys the literature relevant to this contribution. Section 2.1 reviews the historical development of machine learning in compilers, tracing the progression from early heuristic induction to modern neural approaches. Section 2.2 introduces the MLIR compiler infrastructure, its design principles, and the dialect ecosystem on which this thesis builds. Section 2.3 discusses program representations for machine learning, from hand-crafted features to learned embeddings and graph-based encodings. Section 2.4 covers Graph Neural Networks and their application to code analysis. Finally, Section 2.5 reviews transfer learning and domain adaptation, with emphasis on techniques for Graph Neural Networks.

2.1 Machine Learning in Compilers

The idea of using machine learning to guide compiler optimization decisions emerged in the early 2000s and has since grown into a well-established research direction. Three comprehensive surveys chart its development. Ashouri et al. [2018] catalogue over 200 publications spanning 25 years, organizing them along the axes of application characterization, machine learning model, prediction type, and target platform. Wang and O'Boyle [2018] provide a complementary perspective, emphasizing the interplay between feature engineering, model selection, and deployment strategies, and illustrating how the same three-step pipeline, namely, characterizing the program, learning a model, and deploying it inside the compiler, underpins virtually all work in the area (Figure 2.1). Leather and Cummins [2020] offer a retrospective that traces

the field from its origins in iterative compilation through supervised learning and into the deep-learning era, concluding with a vision of reinforcement-learning agents that control every optimization decision in the compiler.

Figure 2.1: Three-step machine-learning compilation pipeline.



Source: Reproduced from Wang and O'Boyle [2018].

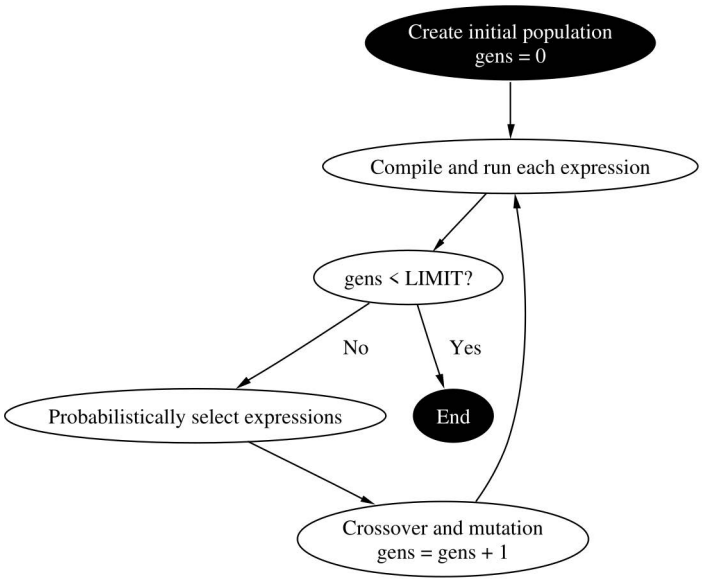
The precursor to machine-learning-based compilation was *iterative compilation*: defining a space of optimization strategies and using search to find the best one for each program. Although effective, with speedups of up to 2.23× [Leather and Cummins, 2020], the search must be repeated for every new program and may require thousands of compile-execute cycles, making it impractical for general use. This cost motivated the shift toward *predictive* approaches: instead of searching at deployment time, one trains a model offline on the results of previous searches and uses it to predict good optimization choices for unseen programs in a single compilation.

One of the earliest studies to adopt this predictive perspective was the work of Monsifrot et al. [2002], who framed the loop-unrolling decision as a classification problem. They represented each loop by six static features, namely, number of memory accesses, arithmetic operations, loop body size, control statements, array-element reuses, and iteration count, and used the OC1 oblique decision-tree learner to induce a rule that predicts whether unrolling a given loop will be beneficial on the target architecture. By comparing the execution times of unrolled and original loops and labeling each instance as *improved* or *degraded*, the authors constructed training sets for two different processors, an UltraSPARC and an IA-64 machine. Their results showed that the learned heuristics outperformed the default unrolling strategy of the GNU Fortran compiler and, critically, that a heuristic trained on one architecture performed poorly when applied to the other. This finding underscored a point that would recur throughout the literature: compiler heuristics are inherently architecture-dependent, and manually tuning them for every new target is unsustainable.

Stephenson et al. [2003] introduced *Meta Optimization*, a methodology that uses genetic programming (GP) to automatically search the space of compiler *priority functions*. The key insight of their work is that many compiler heuristics are controlled by a single cost or priority function, a scoring expression that ranks the options available

to a compiler algorithm. For example, in register allocation, a priority function determines which variable to spill; in list scheduling, it determines which ready instruction to emit next. Rather than hand-tuning these expressions, the authors evolved them using GP, starting from a population of candidate expressions and iteratively selecting, recombining, and mutating them according to a fitness metric derived from the execution time of compiled benchmarks. Figure 2.2 summarizes this evolutionary search process. They demonstrated the approach on three optimizations: hyperblock formation (achieving an average speedup of 23%, with peaks of 73%), register allocation, and data prefetching. In each case, the evolved priority functions matched or exceeded the performance of manually designed ones. The system could operate in two modes: producing application-specific heuristics through feedback-directed optimization, or evolving general-purpose heuristics by training across multiple benchmarks.

Figure 2.2: Genetic programming workflow used in Meta Optimization. Candidate priority functions are evaluated, selected, recombined, and mutated over successive generations to evolve compiler heuristics.



Source: Reproduced from Stephenson et al. [2003].

Cavazos and Moss [2004] applied supervised learning to a different kind of decision: not *how* to optimize, but *whether* to optimize at all. Their target was instruction scheduling in the Jikes RVM just-in-time compiler for Java. Because scheduling is expensive (accounting for over 10% of total compilation time) and often yields no benefit on a given basic block, they sought to *filter* the blocks presented to the scheduler, applying it only where it is predicted to help. Each basic block was characterized by 13 inexpensive static features, including fractions of instructions belonging to categories such as loads, stores, floating-point operations, and hazards, and labeled according to whether scheduling reduced its estimated execution cost. Using the Ripper rule-set induction algorithm, they induced a compact set of if-then rules that could be evaluated

in negligible time. The resulting filter retained over 90% of the benefit of scheduling every block while requiring less than 25% of the scheduling effort. The work demonstrated that even simple, cheap-to-compute features can yield highly effective learned heuristics, an observation that remains relevant today.

In a subsequent study, Cavazos and O'Boyle [2005] used genetic algorithms to automatically tune the parameters controlling method inlining in Jikes RVM. Inlining is a particularly consequential optimization in dynamically compiled languages: it reduces method-call overhead and exposes opportunities for further optimization, but overly aggressive inlining increases code size, instruction-cache pressure, and compilation time. The authors showed that the optimal inlining parameters depend not only on the program and the hardware but also on the compilation scenario (e.g., a full optimization pass versus adaptive hot-spot compilation) and that the default parameters shipped with Jikes RVM were far from optimal in several configurations. By encoding the inlining parameters as a genetic-algorithm chromosome and evolving them over a training set of benchmarks, they achieved a 17% average reduction in total execution time on SPECjvm98 and 37% on the DaCapo benchmark suite. The work reinforced the argument that the "one size fits all" approach to compiler heuristics is inadequate and that machine learning provides a practical path toward automatic, platform-aware specialization.

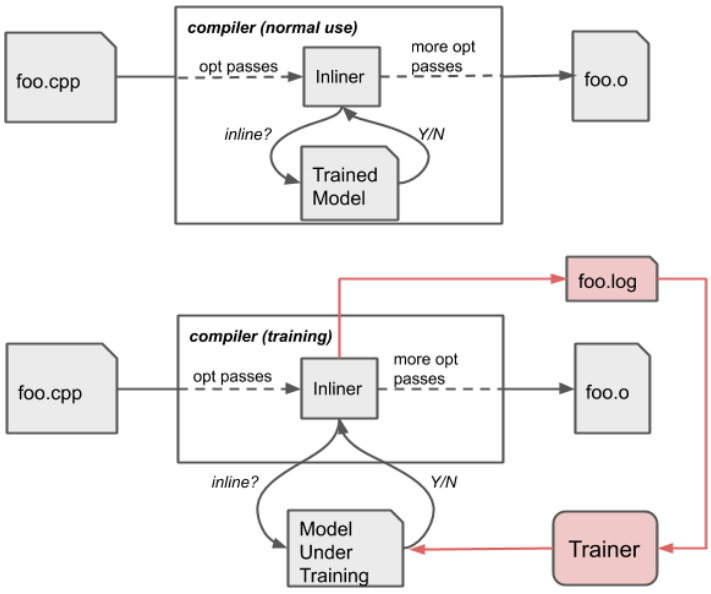
Fursin et al. [2008] brought many of these ideas together in MILEPOST GCC, one of the first open-source, machine-learning-enabled compiler frameworks. MILEPOST integrated iterative compilation, static program features extracted from GCC's intermediate representation, and supervised learning to predict which compiler flags to enable for a given program. The project received wide attention and demonstrated the practical feasibility of the approach, although its reliance on hand-crafted, fixed-length feature vectors limited the expressiveness of its program characterizations.

Beyond these foundational works, machine learning has been applied to a growing range of compilation tasks. Cummins et al. [2017] used deep neural networks for end-to-end prediction of OpenCL device mappings and thread coarsening factors, replacing hand-crafted features with automatically learned representations of program source code. Mendis et al. [2019a] developed Ithemal, a neural network that predicts the throughput of x86 basic blocks directly from their byte representations, achieving higher accuracy than analytical models such as IACA and llvm-mca. In a related line of work, Mendis et al. [2019b] applied imitation learning to compiler auto-vectorization, training a model to mimic an expert policy derived from exhaustive search. da Silva et al. [2022] survey the landscape of program representations for predictive compilation, noting the progression from hand-crafted feature vectors through learned embeddings to graph-based encodings, and highlighting the trade-offs between expressiveness and generality. In the domain of static profiling, Moreira et al. [2021] introduced VESPA,

a machine-learning model that predicts branch frequencies and basic-block execution counts from static program features, enabling profile-guided optimization without runtime profiling. da Silva et al. [2021] contributed AnghaBench, a suite of one million compilable C benchmarks designed to support large-scale code-size-reduction experiments, addressing the chronic shortage of diverse training data in the field.

A milestone in the adoption of machine learning by production compilers was the MLGO framework of Trofin et al. [2021]. MLGO is widely regarded as the first full integration of a machine-learned optimization policy in an industry-strength compiler. The framework replaces the hand-tuned heuristics of the LLVM inliner with a neural network trained via reinforcement learning (Policy Gradient) or Evolution Strategies, achieving up to 7% code-size reduction compared to LLVM's `-Oz` optimization level. A central design principle of MLGO is the strict separation between *training*, which occurs offline on a large representative corpus, and *deployment*, in which the trained model is embedded as statically linked native code in the compiler binary, incurring no additional runtime dependencies (Figure 2.3). This architecture ensures that the compiler's behavior remains deterministic and that build pipelines need not change. Importantly, the authors demonstrated that a model trained on one code corpus generalizes well to other targets and remains effective even after months of active development on those targets, a property that is essential for practical adoption.

Figure 2.3: Overview of the MLGO framework, illustrating the separation between offline model training and deployment within the compiler.



Source: Reproduced from Trofin et al. [2021].

Collectively, these works establish a clear trajectory: from manual heuristic tuning through offline supervised learning to deep reinforcement learning deployed in production compilers. A recurring theme across the literature is the dependence on large, labeled datasets. As Wang and O'Boyle [2018] emphasize, the quality of a learned

model is bounded by the quality and quantity of its training data, and both feature engineering and training-data generation often need to be repeated when the compiler targets a new architecture or representation. This dependence is precisely the bottleneck that motivates the present thesis: in the MLIR ecosystem, new dialects can be introduced rapidly, but the benchmark suites needed to train accurate models for those dialects may not yet exist. The following sections review the specific technical foundations on which our solution builds: the MLIR framework, program representations, graph neural networks, and transfer learning.

2.2 The MLIR Compiler Infrastructure

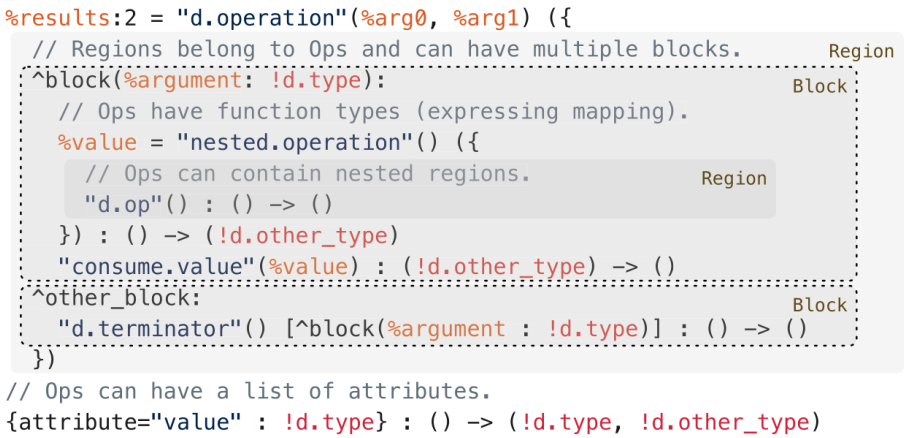
The Multi-Level Intermediate Representation (MLIR) [Lattner et al., 2021] is a compiler infrastructure designed to support multiple domain-specific abstractions within a single, extensible framework. Unlike traditional compiler IRs that provide a single fixed level of abstraction (e.g., LLVM IR, often characterized as "C with vectors"), MLIR allows compiler developers to define custom *dialects*: collections of operations, types, and attributes tailored to a specific domain or abstraction level. This design makes it possible to represent computations ranging from high-level tensor operations in machine learning frameworks to low-level hardware instructions within the same infrastructure, and to lower progressively between these levels through a series of well-defined transformations.

Design and core abstractions. Lattner et al. [2021] organized the design of MLIR around three overarching principles. *Parsimony* dictates that the system should provide as few built-in concepts as possible, making everything else customizable; in practice, this means that "instruction," "function," and "module" are all represented uniformly as *operations* (Ops), the single unit of semantics in MLIR. *Traceability* requires the system to retain rather than recover information: source locations, type annotations, and high-level structure are preserved throughout the compilation pipeline rather than being discarded and later reconstructed through fragile analyses. *Progressivity* holds that premature lowering should be avoided: high-level structure (e.g., loop nests, tensor semantics) is maintained as long as it is useful for analysis or optimization, and is removed only when the compilation has reached a level of abstraction where it is no longer needed.

An MLIR program is organized as a hierarchy of nested structures. Operations are grouped into *blocks*, each of which ends with a terminator operation that defines control-flow successors. Blocks are grouped into *regions*, which are attached to

operations and provide the nesting mechanism of the IR. Values flow between operations in Static Single Assignment (SSA) form, and MLIR uses *block arguments* rather than ϕ-nodes to represent values at control-flow join points. This hierarchical design, in which operations contain regions, regions contain blocks, and blocks contain operations, allows a single program to mix different levels of abstraction: a high-level tensor operation can coexist with low-level arithmetic instructions in the same module, provided they reside in different regions or dialect scopes. Figure 2.4 illustrates this recursive organization of MLIR's core abstractions.

Figure 2.4: Recursive organization of MLIR, where operations contain regions, regions contain blocks, and blocks contain operations.



Source: Reproduced from Lattner et al. [2021].

Dialects and progressive lowering. A central consequence of MLIR's design is that the compilation pipeline becomes a sequence of *dialect-to-dialect lowerings*, each of which replaces operations in one dialect with semantically equivalent operations in a lower-level dialect. For example, a machine-learning model expressed in the `stablehlo` dialect can be lowered to the `linalg` dialect (which makes iteration structure and data movement explicit), then further to the `scf`, `memref`, and `arith` dialects (which express loops, memory buffers, and scalar arithmetic), and finally to the `llvm` dialect for code generation. Each step refines the representation while preserving the essential computation.

This progressive lowering creates the structural relationships between dialects that this thesis exploits. Because lowering preserves computational semantics while expanding syntactic detail, programs at different levels of the pipeline represent the same underlying computation through different vocabularies of operations. As discussed in Section 1.1.4, a model trained on programs at one level of abstraction can therefore transfer structural knowledge to programs at a different level, provided that the representation captures the shared data-flow and control-flow relationships.

The fragmentation problem. The original motivation for MLIR arose from the observation that modern software systems, particularly machine-learning frameworks, suffer from severe infrastructure fragmentation. As Lattner et al. [2021] document, the TensorFlow ecosystem alone spanned multiple independent compiler stacks, including XLA HLO, TensorFlow Lite, TensorRT, and nGraph, each with its own intermediate representation, optimization passes, and code generators. A similar pattern exists in traditional compilers, where languages like Swift, Rust, and Julia each implement custom mid-level IRs on top of LLVM because the single LLVM IR abstraction level is insufficient for their high-level optimizations. MLIR addresses this fragmentation by providing a common infrastructure in which each domain can define its own dialect without reinventing the surrounding compiler machinery (parsing, verification, pass management, parallelism).

High-performance compilation with upstream MLIR. Golin et al. [2024] demonstrate a practical instantiation of this multi-level lowering approach for high-performance AI workloads. Their compilation pipeline ingests models from TensorFlow and PyTorch, converts them to the `linalg`-on-tensor representation, and applies a sequence of hardware-agnostic passes, namely packing, tiling, and fusion, before lowering to a hardware-specific dialect (XSMM) that interfaces with an optimized micro-kernel library (libxsmm). The key insight of their work is that high-level optimizations such as cache-aware tensor distribution and operation fusion can be expressed entirely within the upstream `linalg` dialect, while only the final "last-mile" code generation requires a target-specific dialect. This separation achieves over 90% of the performance of hand-written implementations.

Two aspects of this work are relevant to the present thesis. First, it confirms that the `linalg` dialect occupies a central position in the MLIR ecosystem as the common compiler IR to which multiple high-level frameworks converge, validating its use as a target dialect in our cross-dialect transfer experiments. Second, it highlights that compilation heuristics, such as the choice of tile sizes and packing factors, are currently exposed as command-line flags or hard-coded constants, with the construction of automated cost models left as future work. Machine-learning approaches, including the transfer-learning methodology proposed in this thesis, offer a path toward automating these decisions.

Fine-grained transformation control. Lücke et al. [2025] address a complementary challenge: how to give performance engineers precise control over MLIR's optimization pipeline. Their *Transform dialect* represents compiler transformations as first-class IR operations, allowing users to compose, sequence, and parameterize individual transformation steps (tiling, fusion, unrolling, etc.) without writing new compiler

passes or rebuilding the compiler. Transform scripts operate alongside the computational IR as a separate "schedule" that drives the compilation process, analogous to the scheduling languages of Halide and TVM but integrated into a general-purpose compiler framework.

This work is relevant to the broader context of this thesis for two reasons. First, it illustrates the increasing complexity of the MLIR transformation ecosystem: as dialects and passes proliferate, the number of possible optimization configurations grows combinatorially, making manual tuning impractical. This is precisely the setting in which learned cost models and prediction-based optimization become valuable. Second, the Transform dialect embodies MLIR's design philosophy of extensibility and progressive lowering: new transformation operations can be defined for new dialects with minimal effort, just as new dialects can be defined for new domains. This extensibility is a double-edged sword: it enables rapid innovation but fragments the available training data across an ever-growing number of representations, reinforcing the data-scarcity challenge that motivates the present work.

Relevance to this thesis. The MLIR framework provides the structural foundation on which our cross-dialect transfer methodology is built. Three properties of MLIR are essential. First, its uniform abstractions, namely operations, regions, blocks, operands, and results, are shared across all dialects, enabling a single ProGraML conversion pipeline to handle any MLIR dialect without dialect-specific parsers (Section 3.1.1). Second, progressive lowering creates a natural hierarchy of dialects connected by semantics-preserving transformations, which is precisely the relationship that transfer learning exploits: structural knowledge learned at one level of the hierarchy can be adapted to another. Third, the rapid proliferation of new dialects ensures that the data-scarcity problem addressed by this thesis is not a one-time challenge but a recurring and growing need within the MLIR ecosystem.

2.3 Program Representations for ML

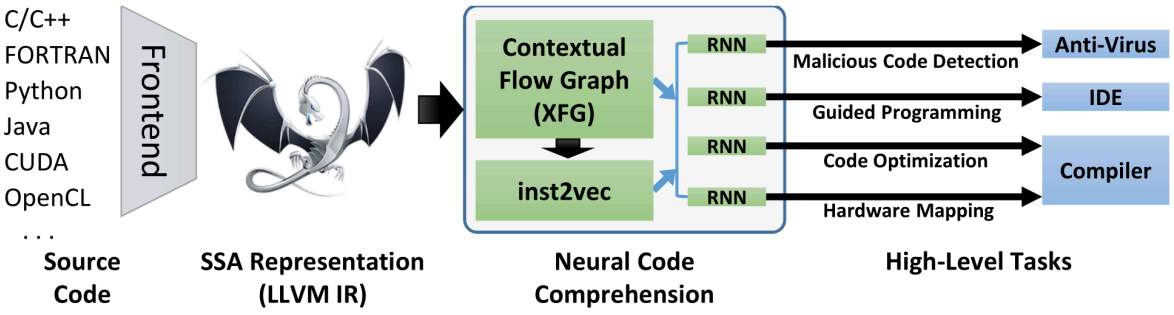
The quality of any learned compiler model is fundamentally constrained by the representation used to describe programs to the learning algorithm. As da Silva et al. [2022] observe in their survey on program representations for predictive compilation, the field has evolved through three broad stages: hand-crafted numerical features, learned token-level embeddings, and structured graph-based representations. Each stage has extended the model's ability to capture program semantics, at the cost of increased computational complexity and data requirements.

Hand-crafted features. The earliest machine-learning approaches to compilation characterized programs using manually selected numerical properties. Cavazos and Moss [2004] represented basic blocks as 13-element vectors of instruction-category fractions, as discussed previously. Fursin et al. [2008] followed a similar strategy in MILEPOST GCC, extracting approximately 55 static features from GCC's intermediate representation, including counts of instructions, branches, basic blocks, and various loop-level statistics. These fixed-length feature vectors were then fed to classical machine-learning algorithms such as decision trees, nearest neighbors, or support vector machines.

Although effective for coarse-grained tasks such as flag selection, hand-crafted features suffer from well-known limitations. First, they are inherently lossy: reducing an entire program to a fixed-length vector discards most structural information, including the data-flow and control-flow relationships between operations. Second, they are fragile: as Cummins et al. [2021] point out, a model based on instruction counts can be trivially confused by the insertion of dead code, which changes the feature vector without altering the program's behavior. Third, designing a good feature set requires substantial domain expertise, and the features must be re-engineered whenever the compiler targets a new intermediate representation or architecture.

Learned embeddings. A significant step beyond hand-crafted features was the introduction of learned distributed representations for compiler instructions. Ben-Nun et al. [2018] proposed *inst2vec*, the first embedding space designed specifically for compiler intermediate representations (Figure 2.5). Rather than processing source code in a particular programming language, inst2vec operates on LLVM IR, the intermediate representation of the LLVM compiler infrastructure. Working at the IR level makes the approach language-independent: programs written in C, C++, Fortran, or OpenCL can all be compiled to LLVM IR and then embedded in the same continuous space.

Figure 2.5: Overview of the inst2vec (Neural Code Comprehension) pipeline.



Source: Reproduced from Ben-Nun et al. [2018].

The central contribution of inst2vec is the *conteXtual Flow Graph* (XFG), a directed multigraph that captures both the data-flow and control-flow context of each

IR statement. Unlike prior code-embedding approaches that defined context as lexicographic proximity (neighboring tokens in the source text), XFGs define context through execution dependence: two statements are contextually related if the execution of one directly depends on the other through data or control flow. Statement pairs extracted from the XFG are then used to train skip-gram embeddings in the style of word2vec, yielding 200-dimensional vectors that capture semantic relationships between instructions. The resulting embedding space exhibits meaningful structure: statements that perform analogous operations on different data types cluster together, and vector arithmetic over embeddings reproduces type-conversion analogies (e.g., integer addition is to floating-point addition as integer subtraction is to floating-point subtraction).

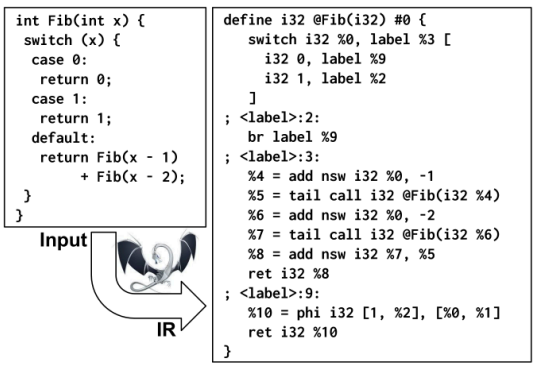
Ben-Nun et al. evaluated inst2vec on three downstream tasks, namely heterogeneous device mapping, thread-coarsening-factor prediction, and algorithm classification, showing that a single LSTM architecture operating on fixed pre-trained embeddings could match or surpass specialized approaches on each task. However, the sequential processing model (an LSTM over the linear sequence of IR statements) fundamentally limits the approach: it cannot capture long-range dependencies across function boundaries or through complex control flow, because the order in which statements appear in the IR text need not reflect their execution dependencies. Moreover, the XFG representation omits information critical for fine-grained analysis, including the order of operands to non-commutative operations and separate vertices for variables and constants.

Graph-based representations: ProGraML. The limitations of sequential models for compiler analysis motivated the development of structured, graph-based program representations. Cummins et al. [2021] introduced *ProGraML* (Program Graphs for Machine Learning), a language-independent, IR-level representation that simultaneously captures the control-flow, data-flow, and call-flow structure of a program as a directed multigraph. ProGraML addresses the shortcomings of both hand-crafted features and prior graph representations (such as XFG and Control-Data Flow Graphs) by incorporating three types of information that earlier approaches omitted: (i) separate vertices for instructions, variables, and constants, enabling per-variable reasoning; (ii) positional attributes on data-flow edges, encoding operand order so that non-commutative operations such as division can be distinguished; and (iii) call-flow edges connecting call sites to function entry points and return statements, supporting inter-procedural analysis.

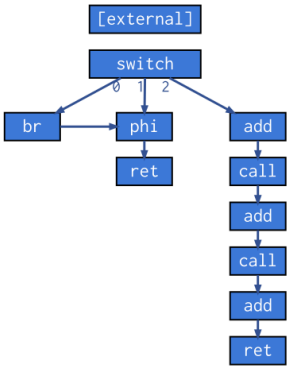
The ProGraML graph is constructed from a compiler IR in three stages. First, a *control-flow* layer is built by inserting a vertex for each instruction and connecting sequential and branching control dependencies. Second, a *data-flow* layer introduces vertices for variables and constants and edges that encode use-def chains, with posi-

tional attributes preserving operand order. Third, a *call-flow* layer adds edges between call instructions and the entry points of callee functions, as well as return edges from function exits back to call sites. Figure 2.6 depicts the conversion of an LLVM IR program into its corresponding ProGraML representation. The resulting graph can be constructed in $O(|V| + |E|)$ time and is processed by a Gated Graph Neural Network (GGNN) [Li et al., 2015], whose message-passing iterations mirror the transfer functions and meet operators of classical iterative data-flow analysis.

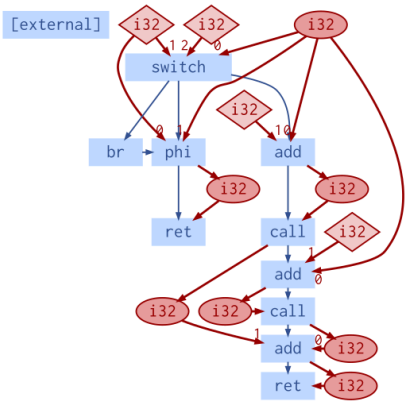
Figure 2.6: Construction of a ProGraML graph from an LLVM-IR implementation of the Fibonacci sequence, illustrating the control-flow, data-flow, and call-flow layers.



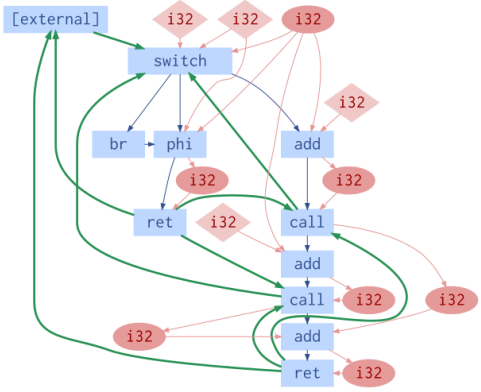
(a) The input program is passed through the compiler front-end to produce an IR. In this example, LLVM-IR is used.



(b) A full-flow graph is constructed of instructions and control dependencies. All edges have position attributes; for clarity, we have omitted position labels where not required.



(c) Vertices are added for data elements (elliptical nodes are variables, diamonds are constants). Data edges capture use/def relations. `i32` indicates 32 bit signed integers. Numbers on edges indicate operand positions.



(d) Functions have a single entry instruction and zero or more exit instructions. Call edges are inserted from call sites to function entry instructions, and return-edges from function exits to call sites.

Source: Reproduced from Cummins et al. [2021].

To evaluate ProGraML's representational power, Cummins et al. introduced the DeepDataFlow benchmark, a dataset of 461 k LLVM-IR programs annotated with ground-truth labels for five classical data-flow analyses: reachability, dominance, data dependencies, liveness, and available subexpressions. On these tasks, a GGNN operating on ProGraML graphs achieved ≥ 0.939 F1 score across all analyses when the number of message-passing iterations was sufficient, significantly outperforming both the sequential inst2vec model and the CDFG graph representation. In partic-

ular, inst2vec and CDFG were structurally unable to perform per-variable analyses (data dependencies and liveness) because they lack separate vertices for variables. On downstream compiler optimization tasks, such as device mapping and algorithm classification, ProGraML reduced test error by 1.20× and 1.35×, respectively, compared to prior representations.

Relevance to this thesis. The ProGraML representation is central to the methodology presented in Chapter 3. Because ProGraML encodes programs as graphs of operations, values, and typed dependencies rather than as dialect-specific token sequences, it provides a natural normalization layer across MLIR dialects. Programs lowered from `stablehlo` to `linalg` to `scf/memref/arith` produce different ProGraML graphs, but those graphs share structural motifs, including chains of producer-consumer data-flow edges, control-flow sequencing within basic blocks, and inter-procedural call edges, that a Graph Neural Network can learn to recognize. This structural overlap is what makes cross-dialect transfer learning possible: a GGNN pre-trained on ProGraML graphs from one dialect can transfer its learned message-passing behavior to graphs from a different dialect, provided that the dialect-specific components of the model (primarily the embedding layer) are appropriately adapted. We extend ProGraML from LLVM-IR to MLIR in Section 3.1.1, building on the same three-stage construction (control flow, data flow, call flow) while leveraging MLIR's uniform abstractions of operations, regions, and blocks to support multiple dialects through a single conversion pipeline.

2.4 Graph Neural Networks for Code

Programs are naturally expressed as graphs: compilers reason about code through control-flow graphs, data-flow graphs, and call graphs, all of which encode relational structure that sequential models such as LSTMs can capture only indirectly. Graph Neural Networks (GNNs) provide a principled framework for learning over such relational data, and their application to compiler analysis tasks has yielded substantial improvements over both hand-crafted features and sequence-based representations. This section reviews the GNN foundations on which the model used in this thesis is built.

The Graph Neural Network model. Scarselli et al. [2009] introduced the Graph Neural Network (GNN), the first neural architecture designed to operate directly on arbitrary graph-structured data, including cyclic, directed, and undirected graphs. The model is based on iterative state propagation: each node v maintains a hidden state vector $\mathbf{h}_v$ that is repeatedly updated as a function of the node's own label and the

states and labels of its neighboring nodes through a shared local transition function f. Formally, the state of each node is defined by

$$\mathbf{h}_v = f\big(\mathbf{l}_v,\ \mathbf{l}_{\mathrm{co}(v)},\ \mathbf{h}_{\mathrm{ne}(v)},\ \mathbf{l}_{\mathrm{ne}(v)}\big)\,, \tag{2.1}$$

where $\mathbf{l}_v$ is the label of node v, $\mathrm{co}(v)$ denotes the set of edges incident to v, and $\mathrm{ne}(v)$ denotes its neighboring nodes. The hidden states are iteratively updated until convergence to a fixed point. A separate output function g then maps each node's converged state to a task-specific prediction.

To guarantee the existence of a unique fixed point, Scarselli et al. require the local transition function f to be a contraction map, meaning that it shrinks the distance between successive state estimates by a factor strictly smaller than one. Under this assumption, the Banach fixed-point theorem guarantees convergence to a unique solution from any initialization. Learning is performed using the Almeida-Pineda algorithm, which computes gradients at the converged fixed point without storing the intermediate states generated during the iterative process.

The contraction-map requirement, while theoretically elegant, imposes a practical limitation: it restricts the class of functions f can represent and makes it difficult for the network to propagate information across long distances in the graph. Li et al. [2015] illustrate this limitation with an example showing that information propagation in contraction-map GNNs may require a number of iterations that grows exponentially with the graph diameter. This challenge motivated the development of Gated Graph Neural Networks.

Gated Graph Neural Networks. Li et al. [2015] proposed the Gated Graph Neural Network (GGNN), a modernized variant that replaces the contraction-map constraint with two key changes: (i) the state update uses a Gated Recurrent Unit (GRU) cell instead of a generic contraction function, and (ii) the recurrence is unrolled for a fixed number of steps T and trained end-to-end via backpropagation through time, rather than relying on convergence to a fixed point.

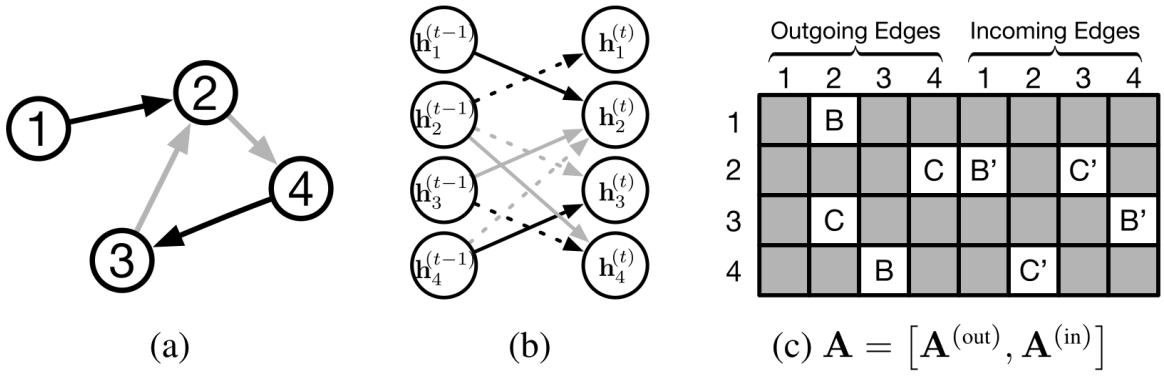
At each iteration t, the GGNN first aggregates incoming messages from neighboring nodes through an edge-type-specific linear transformation. For a graph with multiple edge types (e.g., forward and backward edges, different relation types), the transformation matrix $\mathbf{A}$ is structured so that each edge type and direction has its own learned parameters, while the overall sparsity pattern mirrors the graph's adjacency structure. The aggregated message is then used to update each node's hidden state via the GRU cell:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_v^{\top} \left[\mathbf{h}_1^{(t-1)\top} \cdots \mathbf{h}_{|V|}^{(t-1)\top}\right]^{\top} + \mathbf{b}, \tag{2.2}$$

$$\mathbf{h}_v^{(t)} = \mathrm{GRU}\big(\mathbf{a}_v^{(t)},\ \mathbf{h}_v^{(t-1)}\big)\,, \tag{2.3}$$

where $\mathbf{A}_v$ selects the relevant rows of the global parameter matrix for the edges incident to node v. The GRU gates (update gate $\mathbf{z}$, reset gate $\mathbf{r}$) allow the network to selectively retain or overwrite information at each step, enabling effective propagation over long distances without the contraction-map bottleneck. Figure 2.7 illustrates the GGNN message-passing architecture, including typed edges, recurrent state propagation, and the parameter-tying scheme used to support multiple edge types.

Figure 2.7: GGNN architecture illustrating typed edges, one propagation step, and the parameter-tying structure used for edge-type-specific message passing.



Source: Reproduced from Li et al. [2015].

Li et al. also introduced *node annotations*, initial feature vectors that mark nodes with task-specific information (e.g., flagging the source and target nodes in a reachability query). For output, the model supports both per-node classification and graph-level prediction; the latter is achieved through a gated attention mechanism that soft-selects which nodes are relevant to the graph-level task. The authors extended the architecture to *Gated Graph Sequence Neural Networks* (GGS-NNs), which chain multiple GGNN modules to produce sequential outputs such as logical formulas. On program verification tasks, specifically classifying heap data structures from memory graphs, the GGS-NN matched the accuracy of a system built on extensively hand-engineered features, demonstrating that GGNNs can learn meaningful program abstractions directly from graph structure.

Application to compiler analysis. The GGNN architecture has been particularly successful in compiler-related tasks. Cummins et al. [2021] adopted GGNNs as the learning model for ProGraML graphs (discussed in Section 2.3), achieving ≥ 0.939 F1 on five classical data-flow analyses and setting new state-of-the-art results on device mapping and algorithm classification. Their adaptation introduced position-aware message passing, in which a learned gating vector modulates messages according to operand order, allowing the network to distinguish non-commutative operations such as subtraction and division. Mendis et al. [2019b] applied a related graph neural network

to compiler auto-vectorization, representing LLVM-IR programs as graphs with edge types that encode operand positions and augmenting the structure with a priori vectorization opportunities. Their model was trained via imitation learning to replicate an expert policy derived from exhaustive search, achieving competitive vectorization decisions without manual heuristic design.

In an earlier line of work, Cummins et al. [2017] demonstrated end-to-end deep learning for heterogeneous device mapping and thread coarsening in OpenCL, using language models over source-code tokens. While effective, these sequence-based approaches lack the structural inductive bias that graph models provide, and subsequent work consistently showed that graph representations outperform sequential ones on tasks that require reasoning about data flow and control flow.

Pre-trained models for code. A parallel line of research has explored the large-scale pre-training of Transformer-based models on code corpora. Feng et al. [2020] introduced CodeBERT, a bimodal pre-trained model for programming and natural languages that learns token-level representations from large code-documentation corpora using masked language modeling and replaced-token detection objectives. Guo et al. [2021] extended this approach with GraphCodeBERT, which incorporates data-flow information during pre-training by adding graph-based objectives that require the model to predict which variables are connected by data-flow edges. GraphCodeBERT demonstrated that even a Transformer architecture benefits from explicit structural signals: it outperformed CodeBERT on downstream tasks such as code search, clone detection, and code translation.

These pre-trained models achieve strong performance by leveraging massive code corpora and transfer learning across programming tasks. However, they operate at the source-code or token level and are not directly applicable to compiler intermediate representations, where the relevant structure is defined by the IR's operation semantics rather than by surface syntax. In contrast, the GGNN-based approach adopted in this thesis operates on IR-level graphs (ProGraML), making it naturally compatible with different MLIR dialects and enabling the component-level transfer learning strategy described in Chapter 3.

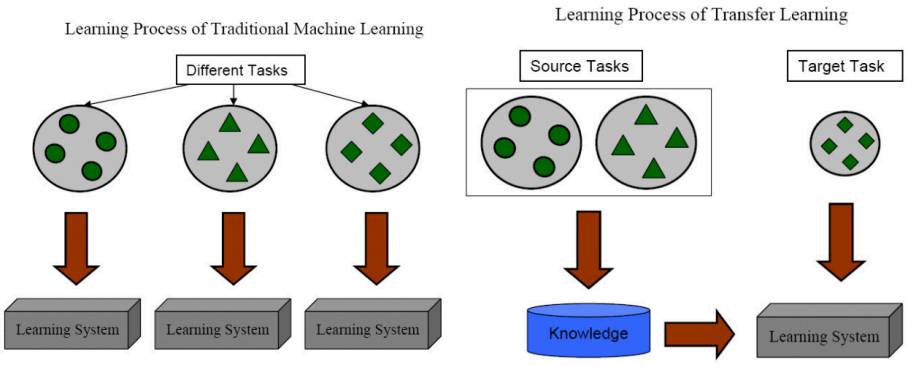
Relevance to this thesis. The GGNN architecture is central to the transfer-learning methodology presented in this work. As described in Section 2.3, ProGraML graphs capture the control-flow, data-flow, and call-flow structure of MLIR programs, and a GGNN processes these graphs through iterative message passing using edge-type-specific transformations, positional gating, and GRU-based state updates. The GGNN's parameters decompose naturally into functionally distinct groups, namely, the embedding lookup table, the message-passing layers (edge transformations, position MLP,

and GRU cell), and the readout MLP, each of which can be independently frozen or fine-tuned during cross-dialect adaptation. This modular decomposition enables the selective fine-tuning experiments (FTE1–FTE8) evaluated in Chapter 4, where we study which parameter groups encode universal program structure and which must adapt to dialect-specific semantics.

2.5 Transfer Learning and Domain Adaptation

A central premise of most machine learning algorithms is that the training and test data are drawn from the same feature space and the same distribution. When this assumption is violated, for instance, because the target domain has too few labeled examples or because the input distribution has shifted, models trained in the standard way perform poorly. Transfer learning addresses this problem by leveraging knowledge acquired in a data-rich *source domain* to improve learning in a related but data-scarce *target domain* (Figure 2.8) [Pan and Yang, 2010]. This section reviews the theoretical foundations of transfer learning, the problem of catastrophic forgetting that arises during sequential adaptation, and recent techniques for parameter-efficient adaptation of Graph Neural Networks.

Figure 2.8: Traditional machine learning versus transfer learning.



(a) Traditional Machine Learning (b) Transfer Learning

Source: Reproduced from Pan and Yang [2010].

Foundations and taxonomy. Pan and Yang [2010] provide the standard formal framework for transfer learning. They define a *domain* as a pair $\mathcal{D} = \{\mathcal{X}, P(X)\}$ consisting of a feature space $\mathcal{X}$ and a marginal distribution $P(X)$, and a *task* as a pair $\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$ consisting of a label space and a conditional distribution. Transfer

learning is then defined as the use of knowledge from a source domain $\mathcal{D}_S$ and task $\mathcal{T}_S$ to improve the predictive function f_T in a target domain $\mathcal{D}_T$ where $\mathcal{D}_S \neq \mathcal{D}_T$ or $\mathcal{T}_S \neq \mathcal{T}_T$.

Pan and Yang organize the field around three fundamental questions: *what* to transfer, *how* to transfer, and *when* to transfer. They further identify three settings based on the relationship between source and target: *inductive* transfer learning (different tasks, labeled target data available), *transductive* transfer learning (same task, different domains, no target labels), and *unsupervised* transfer learning (different tasks, no labels in either domain). Within these settings, four families of approaches are distinguished: instance transfer (re-weighting source examples), feature-representation transfer (learning a shared representation that reduces domain divergence), parameter transfer (discovering shared model parameters or priors), and relational-knowledge transfer (mapping relational structures between domains).

In the context of this thesis, cross-dialect transfer learning is most closely aligned with the *transductive* setting and, more specifically, with *domain adaptation*: the task is the same across dialects (predicting training time or test accuracy of the same neural network architectures), but the domains differ because different MLIR dialects produce ProGraML graphs with different operation vocabularies and structural distributions. A notable departure from the standard transductive formulation is that a small amount of labeled target-domain data is available (10% of the dataset), enabling supervised fine-tuning rather than purely unsupervised adaptation. The transferred knowledge is encoded in the shared parameters of the GGNN, particularly in the message-passing layers, while the dialect-specific parameters (the embedding lookup table) are adapted to the target domain using the available labeled examples. In terms of transfer mechanism, the approach follows the *parameter transfer* strategy: pre-trained model parameters that capture general program structure are selectively preserved, while a subset of parameters is retrained on the target dialect.

Catastrophic forgetting. A fundamental challenge in sequential learning is *catastrophic forgetting*: the tendency for a neural network to abruptly lose knowledge of previously learned tasks when its parameters are updated to optimize a new task [Kirkpatrick et al., 2017]. Kirkpatrick et al. formalized this problem and proposed *Elastic Weight Consolidation* (EWC), an algorithm that selectively slows down learning on weights that are important to previously learned tasks.

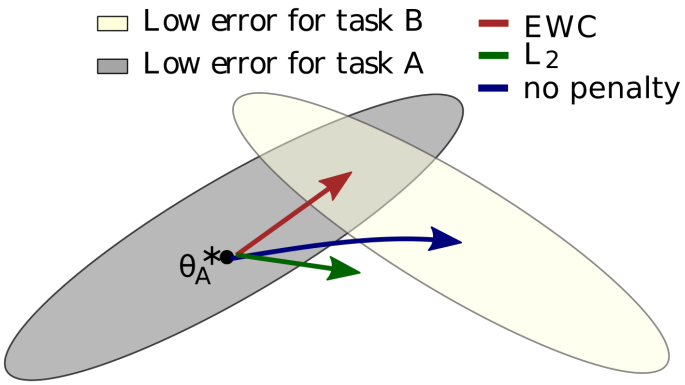
The key insight of EWC is that neural networks are typically overparameterized: many configurations of weights yield equivalent performance on a given task. This means that, when the network needs to learn a second task, there likely exists a region in parameter space where both tasks perform well. The challenge is to find this region without overwriting the parameters critical to the first task (Figure 2.9). EWC achieves

this by adding a quadratic penalty to the loss function during training on the new task:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i \left(\theta_i - \theta^*_{A,i}\right)^2, \tag{2.4}$$

where $\mathcal{L}_B$ is the loss for the new task, $\theta^*_{A,i}$ are the parameters learned on the old task, F_i is the diagonal of the Fisher information matrix (measuring how important each parameter θ_i is to the old task), and λ controls the relative importance of old versus new knowledge. Parameters with high Fisher information are strongly anchored to their old values, while parameters with low Fisher information are free to change. Kirkpatrick et al. demonstrated that EWC enables continual learning on permuted MNIST tasks and Atari 2600 games, whereas standard gradient descent and uniform regularization both fail.

Figure 2.9: Illustration of catastrophic forgetting and Elastic Weight Consolidation (EWC) in parameter space. EWC guides optimization toward solutions that preserve performance on a previously learned task while adapting to a new one.



Source: Reproduced from Kirkpatrick et al. [2017].

The phenomenon of catastrophic forgetting is directly relevant to the fine-tuning experiments in this thesis. As shown in Section 4, full fine-tuning (FTE4) of a GGNN pre-trained on `stablehlo` leads to performance close to the scratch-trained baseline, indicating that the unconstrained parameter updates overwrite the useful structural representations learned during pre-training. Our selective freezing strategy (FTE2, FTE3, FTE6, FTE7) can be understood as an extreme form of the EWC principle: rather than using a soft quadratic penalty, we set the effective regularization strength to infinity for selected parameter groups, preventing any change to the components that encode transferable program structure.

Parameter-efficient fine-tuning for Graph Neural Networks. Li et al. [2024] introduced AdapterGNN, a parameter-efficient fine-tuning (PEFT) method designed specifically for Graph Neural Networks. Unlike Transformer-based models, where PEFT techniques such as LoRA and prefix tuning have been extensively studied, GNNs

present distinct challenges: they lack the self-attention layers targeted by most PEFT methods, and larger GNN models do not necessarily generalize better, contrary to the scaling trends observed in large language models.

AdapterGNN addresses these challenges by inserting lightweight *adapter modules*, small bottleneck networks with batch normalization and learnable scaling, into the GNN architecture. During fine-tuning, only the adapter parameters and the final prediction head are updated, while the pre-trained GNN backbone remains frozen. On molecular property prediction benchmarks, AdapterGNN outperformed full fine-tuning by 1.6% in chemistry and 5.7% in biology while tuning only 4–5% of the total parameters. Li et al. provided a theoretical justification through generalization bounds: by reducing the effective number of tunable parameters, PEFT shrinks the hypothesis space and thus tightens the generalization bound, mitigating the overfitting that arises when large models are fine-tuned on small datasets.

This finding resonates with the results of this thesis. Our experiments show that configurations with more frozen parameters (e.g., FTE2 and FTE6, which freeze the edge and positional MLPs) consistently outperform full fine-tuning (FTE4 and FTE8), supporting the principle that constraining the adaptation to a small number of parameters preserves pre-trained knowledge and improves generalization in the low-data regime.

Parameter freezing in Graph Neural Networks. Blake et al. [2021] investigated which components of a GNN architecture should be frozen during training in the context of reinforcement learning for locomotion control. Their method, *Snowflake*, applies to GNNs based on the Gated Graph Neural Network architecture and addresses the observation that GNN performance degrades rapidly as the dimensionality of the input and output spaces grows. Through a systematic analysis, the authors traced this degradation to overfitting in specific parts of the network: the encoder, decoder, and message-passing functions. They found that the best performance was achieved by freezing these components entirely and training only the GRU update function.

This result is particularly relevant to the present work, as it provides independent evidence that, in GNN architectures with GRU-based message passing, the state-update mechanism (the GRU cell) captures the most task-specific information and benefits most from adaptation, while the message-routing components (edge transformations, positional encodings) encode more general structural knowledge that can be safely preserved. Our fine-tuning experiments arrive at a complementary conclusion from the opposite direction: whereas Snowflake freezes the message-passing layers and trains the GRU, our best-performing configurations (FTE6, FTE7) freeze the edge and positional MLPs while allowing the GRU to adapt, and additionally retrain the embedding layer to accommodate the new dialect vocabulary. Both findings point to the same

architectural insight: GNN message-passing components learn reusable relational patterns, whereas the state-update mechanism captures domain-specific dynamics. This observation suggests that the decomposition may be a general property of GRU-based graph neural networks.

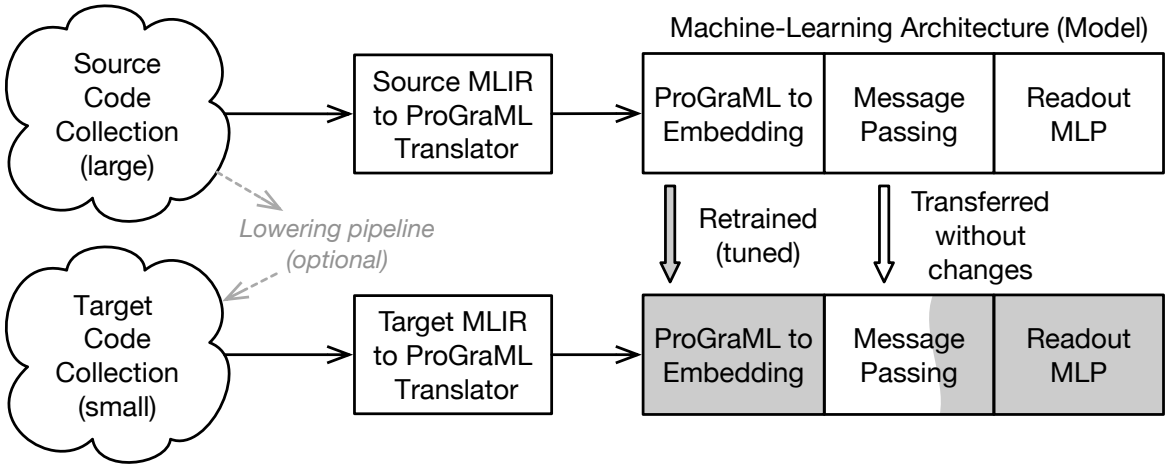
Relevance to this thesis. The transfer-learning methodology of this thesis draws directly on the foundations reviewed in this section. From Pan and Yang's taxonomy, it adopts the transductive parameter-transfer framework, applying it to the novel setting of MLIR compiler dialects. From the catastrophic forgetting literature, it inherits the insight that unconstrained fine-tuning can destroy pre-trained representations, motivating the selective freezing strategy evaluated in Chapter 4. From AdapterGNN and Snowflake, it borrows the empirical finding that GNN architectures decompose naturally into transferable and task-specific parameter groups, and that freezing the former while adapting the latter yields the best trade-off between knowledge preservation and domain adaptation. The contribution of this thesis is to bring these ideas together in the context of compiler intermediate representations, demonstrating that the same principles apply when the domain shift is not between molecular datasets or locomotion tasks but between different levels of a compiler's lowering pipeline.

Chapter 3

Transfer Learning Across Program IRs

The methodology proposed in this thesis is based on two components: a common graph representation that maps programs from any MLIR dialect into the same structured format, and a graph neural network training strategy that transfers learned knowledge across dialects through selective fine-tuning. Figure 3.1 provides an overview of the proposed cross-dialect transfer pipeline. Programs from both the source and target dialects are first converted into ProGraML graphs and then mapped to vector representations through an embedding lookup stage. These representations are processed by a GGNN model that is pre-trained on a large source corpus and later adapted to the target dialect through selective fine-tuning of different parameter groups.

Figure 3.1: Cross-dialect transfer pipeline. Programs are mapped from MLIR to ProGraML graphs and node embeddings. A GGNN is pre-trained on a large source corpus and adapted to a smaller target corpus via selective fine-tuning.



Source: Created by the author.

A key design decision in this pipeline is the choice of *where* in the compilation stack to extract the program representation. The original ProGraML implementation [Cummins et al., 2021] operates on LLVM-IR, a low-level representation optimized for code generation. While effective for many tasks, LLVM-IR aggressively lowers high-level constructs into primitive operations: loops become sequences of comparisons and

conditional branches, multi-dimensional arrays are flattened into raw pointer arithmetic, and structured parallelism is reduced to opaque runtime calls. Much of the original program structure is lost in this process. By extracting ProGraML graphs from MLIR instead, the pipeline preserves high-level semantic information, such as tensor types, structured loop nests, and domain-specific operations, that would otherwise be discarded. This richer representation gives the GNN access to the structural and semantic cues that are most relevant for predicting program properties.

This chapter is organized as follows. Section 3.1 introduces the ProGraML representation and describes how it is extended from LLVM-IR to MLIR, including the conversion tool that implements this extension. Section 3.2 presents the GGNN architecture and the selective fine-tuning strategy used for cross-dialect transfer.

3.1 ProGraML: From Programs to Vectors

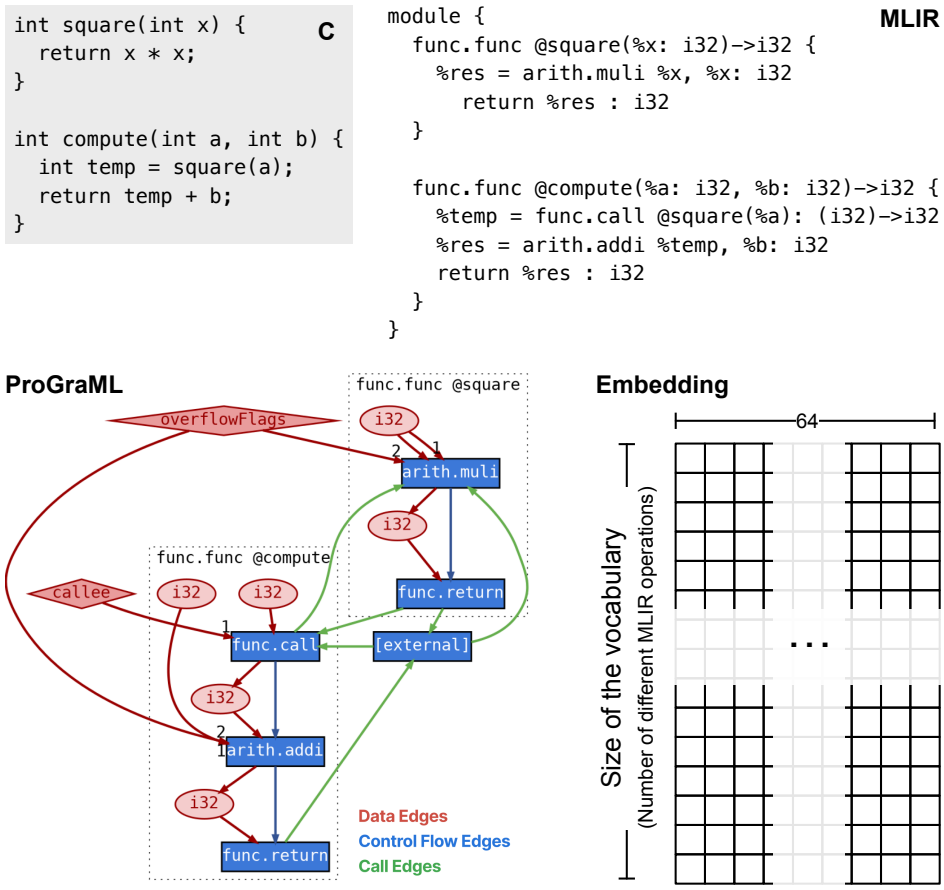
A key requirement for transfer learning across dialects is a representation that is expressive enough to capture program semantics yet dialect-agnostic enough to be applied uniformly across IRs. We adopt ProGraML [Cummins et al., 2021] (Program Graphs for Machine Learning), a graph-based program representation originally proposed for LLVM-IR that we extend to MLIR. Definition 3.1 revisits this concept.

Definition 3.1 (ProGraML Graph)**.** A ProGraML graph is a directed multigraph $G = (V, E)$ where nodes $v \in V$ represent fundamental program components (*operations*, *variables*, or *constants*), and edges $e \in E$ are typed as one of three semantic relations: *control flow*, *data flow*, or *call flow*. Data-flow edges carry a position attribute encoding operand order.

Unlike representations that treat programs as flat token sequences or abstract syntax trees, ProGraML explicitly encodes the two relational structures compilers reason about: execution order (control flow) and value dependencies (data flow). This makes it well suited for learning tasks that depend on execution cost, since both the type of computation and its dependencies are directly captured in the graph.

Example 3.2. Figure 3.2 shows a C program lowered into MLIR and converted into a ProGraML graph, where operations, variables, and constants become nodes connected by control-flow and data-flow edges. Node labels are then mapped to vector representations via an embedding lookup stage, described in Section 3.1.2.

Figure 3.2: A four-stage diagram showing the progression from a C code snippet to its MLIR representation, then to a ProGraML graph, and finally to an embedding lookup table.



Source: Created by the author.

Handling Structure and Control Flow. The most important aspect of translating MLIR into ProGraML is assigning edge types so the graph preserves execution semantics. As shown in Figure 3.2, three categories of edges are introduced:

Data flow edges: These edges are derived from MLIR's SSA use-def chains and represent dependencies between operations and values. For an operation such as (`%res = arith.addf %a, %b`), edges are created from `%a` and `%b` to the operation node, and from the operation node to `%res`. Operand positions are preserved through edge attributes, allowing the model to distinguish argument roles when order matters.

Control flow edges: Model execution order and control flow relationships. Within a basic block, operation nodes are connected sequentially to represent program order. Across blocks, terminator operations such as `cf.cond_br` connect to the entry operation of successor blocks, preserving branch structure, loop back-edges, and alternative paths.

Call flow edges: Captures inter-procedural structure to represent interactions across function boundaries. When a call operation such as `func.call` is encountered, edges connect the call operation to the entry node of the callee function, and edges from the callee's return points connect back to the call site.

3.1.1 Mapping MLIR Dialects to ProGraML

We chose ProGraML as the core program representation because it provides a convenient graph abstraction over MLIR programs. However, the dialect-agnostic nature of our approach does not stem specifically from ProGraML itself, but rather from the generic abstractions provided by MLIR. MLIR represents programs using common concepts such as operations, regions, blocks, operands, and results, which are shared across dialects. Consequently, a single conversion pipeline from MLIR into ProGraML graphs can be reused for different dialects without requiring dialect-specific parsers or graph encoders. In contrast, transferring across unrelated compiler IRs typically requires custom frontends and representation-specific adaptations for each IR.

The MLIR-to-ProGraML mapping. The conversion establishes a direct correspondence between MLIR's core abstractions and ProGraML's node types:

Operations → Instruction nodes: Each MLIR operation (e.g., `arith.addf`, `linalg.matmul`, `stablehlo.convolution`) becomes an instruction node in the graph, labeled with its opcode. This label later serves as the key for the embedding lookup (Section 3.1.2).

SSA Values → Variable nodes: Each SSA value, whether an operation result or a block argument, becomes a variable node, labeled with its data type (e.g., `f32`, `tensor<1x8x3x3xf32>`). Data-flow edges connect variable nodes to the operations that consume them and from the operations that produce them, preserving operand positions.

Attributes → Constant nodes: MLIR operations carry static metadata through named attributes (for example, the stride and padding of a convolution, or the permutation of a transpose). Each distinct attribute value is mapped to a constant node and connected to its parent operation via a data-flow edge. This is an extension beyond the original ProGraML design, which handles constants only as immediate values in LLVM-IR instructions. By surfacing attributes as explicit graph nodes, the representation exposes configuration parameters that may influence execution cost.

Handling the hierarchical structure. A central challenge in the conversion is the mismatch between MLIR's recursive, hierarchical structure and ProGraML's flat graph representation. MLIR programs are organized as a tree: a module contains functions, functions contain regions, regions contain blocks, and blocks contain operations, which may themselves contain nested regions (e.g., loop bodies, conditional branches). ProGraML, by contrast, is a directed multigraph with no hierarchical nesting.

The conversion tool resolves this mismatch through a recursive traversal. For each function, the tool visits every block and processes its operations sequentially, creating instruction nodes and connecting them with control-flow edges to establish intra-block ordering. When an operation contains nested regions, such as an `scf.for` loop or an `scf.if` conditional, the tool recurses into each region, processes the contained blocks, and introduces control-flow edges between the parent operation and the entry point of each nested region. For operations recognized as loops (via MLIR's `LoopLikeOpInterface`), additional back-edges are created from the region's exit to the loop header, preserving loop structure in the graph. Inter-block control flow is handled by connecting each block's terminator to the entry nodes of its successor blocks, as determined by MLIR's block successor information.

Normalization across dialects. Different MLIR dialects may express the same computation using very different syntactic structures, as Example 1.4 showed. ProGraML normalizes these differences into a common graph structure that exposes data and control dependencies. Still referring to Example 1.4, although the operation vocabularies of `stablehlo` and `linalg` differ, equivalent programs often induce similar data-flow topologies: both dialects produce graphs of computation nodes connected by directed data edges, with analogous producer-consumer relationships between operations. A GNN trained to reason over these dependencies in one dialect can therefore transfer part of this structural knowledge to another dialect. Importantly, this observation is not unique to ProGraML: other representations built on top of MLIR's generic abstractions could also support cross-dialect transfer learning, provided that they preserve structural relationships between operations.

Implementation. The conversion pipeline is implemented as a C++ tool built on top of the LLVM/MLIR framework.[1] The tool uses MLIR's generic operation interfaces (in particular, `FunctionOpInterface` for identifying function boundaries, `CallOpInterface` for resolving call targets, and `LoopLikeOpInterface` for detecting loop structures) rather than dialect-specific APIs. This design ensures that the converter works with any MLIR dialect that implements these standard interfaces, without requiring modi-

[1] The complete source code, including the MLIR-to-ProGraML converter and the Python machine-learning experiments, is publicly available at https://github.com/lac-dcc/sphinx.

fications for each new dialect. The output is a serialized Protocol Buffer conforming to the ProGraML graph schema, which can be directly loaded by the PyTorch Geometric data pipeline used in the training experiments. The tool supports both single-file conversion and multi-threaded batch processing of entire datasets.

3.1.2 Node Embeddings

Neural networks operate on continuous vectors rather than discrete symbols. To bridge this gap, each node v is assigned an initial state vector $h_v^0 \in \mathbb{R}^d$ through an embedding lookup. We maintain vocabularies of MLIR operation opcodes and data types observed during training. For operation nodes, the embedding is indexed by the operation opcode (e.g., `linalg.matmul`, `stablehlo.convolution`); for variable and constant nodes, it is indexed by the data type (e.g., `f32`, `i64`). These vectors are learned jointly with the rest of the model and encode semantic information for each node category.

Example 3.3. Consider a node labeled `stablehlo.maximum`, which maps to index 17 in the vocabulary. The embedding lookup table assigns it an initial feature vector:

$$h_v^0 = [0.12, -0.30, \dots] \in \mathbb{R}^d$$

Similarly, a value node such as `tensor<?x32x32x16xf32>` is mapped using its type. As illustrated in Figure 3.2, these vectors define the initial hidden states of nodes, which, together with the graph structure, form the input to the GNN.

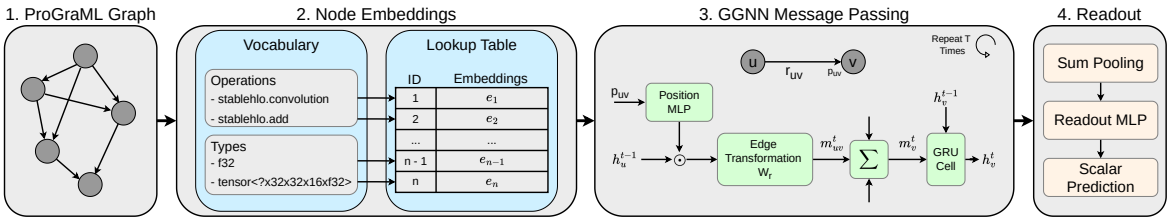
The opcode vocabulary is dialect-specific: `stablehlo` and `linalg` programs may share type embeddings while using different operation embeddings. This distinction is relevant during fine-tuning, as different components of the model may require adaptation to the target dialect (Section 3.2).

3.2 Graph Neural Network Model and Fine-Tuning

Given the ProGraML graph of a program, our goal is to predict a scalar property (training time or test accuracy) by aggregating information from all nodes. We use a Gated Graph Neural Network (GGNN) [Li et al., 2015] following the architecture of Cummins et al. [2021] (see Definition 3.4). Figure 3.3 provides an overview of the

prediction model used throughout this work, including the embedding stage, message-passing iterations, and readout. To study transfer across dialects, we fine-tune different parameter subsets of the model.

Figure 3.3: Overview of the GGNN prediction model. ProGraML graph nodes are mapped to embeddings and updated through T message-passing iterations using edge-type transformations, positional encodings, and a GRU cell. Final node states are pooled and processed by a readout MLP to predict scalar properties such as training time or test accuracy.



Source: Created by the author.

Definition 3.4 (Gated Graph Neural Network)**.** A Gated Graph Neural Network (GGNN) is a graph neural network in which node states are updated iteratively through a message-passing process. At each iteration t, every node v aggregates messages from its neighbors via typed, position-aware edge functions, and updates its hidden state h_v^t using a Gated Recurrent Unit (GRU) cell. After T iterations, the final node states are used by a task-specific output layer.

Message Passing. As illustrated in Figure 3.3, the GGNN updates node states through repeated message-passing iterations. At each iteration t, the hidden state of each node is updated by aggregating messages from its incoming neighbors. For each edge type r (control, data, call, and their reverse directions), a learned edge-type-specific linear transformation is applied. A separate Position MLP uses the edge position attribute to generate a gating vector, allowing the model to modulate messages according to operand order. For an edge (u, v) of type r, the message is computed as:

$$m_{u \to v}^t = W_r \left(h_u^{t-1} \odot g(p_{uv}) \right), \tag{3.1}$$

where W_r is the transformation associated with edge type r, p_{uv} is the edge position attribute, $g(\cdot)$ is the Position MLP, and $\odot$ denotes element-wise multiplication. Messages arriving at node v are summed:

$$m_v^t = \sum_{(u,v) \in E} m_{u \to v}^t. \tag{3.2}$$

The node state is then updated using a GRU cell:

$$h_v^t = \mathrm{GRU}(m_v^t,\, h_v^{t-1}). \tag{3.3}$$

After T iterations, the final node states are aggregated through sum pooling into a graph-level embedding, which is processed by a readout MLP to produce a scalar prediction.

Parameter Groups for Transfer Learning. Although the GGNN architecture shown in Figure 3.3 remains unchanged, its parameters can be grouped into functional components for transfer learning experiments. This grouping allows us to study which parts capture reusable graph-processing behavior and which require adaptation to a new dialect.

Definition 3.5 (Transferable Parameter Groups)**.** We consider the following parameter groups:

- *Embedding layer*: the input lookup table that maps operation tokens and data types to initial node vectors.
- *Message-passing layers*: the Edge transformations, Position MLP, and GRU update cell that propagate and integrate information across the graph.
- *Readout layer*: the final MLP that maps the pooled graph representation to a scalar prediction.

These groups define the freezing configurations evaluated in our transfer experiments. During adaptation, selected parameter groups are kept fixed while the remaining groups are updated on the target dialect corpus.

Pre-training. All parameter groups are first trained jointly on the source dialect corpus until convergence. This stage learns node embeddings, message-passing behavior, and the final prediction mapping from a larger source dataset.

Fine-tuning. To adapt the model to a target dialect, we introduce an embedding table compatible with the target operation vocabulary. Its initialization is evaluated in Section 4.2.5: embeddings are initialized either randomly or from the mean vector of the pre-trained source embedding space. We then fine-tune the model on the smaller target corpus, optionally freezing selected parameter groups such as the Edge transformations, Position MLP, or GRU cell. Across our experiments, we evaluate eight configurations that Section 4.1 describes (FTE1–FTE8). Each configuration varies which groups remain frozen and how embeddings are initialized.

Chapter 4

Evaluation

This chapter evaluates the proposed cross-dialect transfer learning methodology through a series of controlled experiments. All experiments share a common setup described in Section 4.1: a Gated Graph Neural Network pre-trained on a large corpus of `stablehlo` programs and subsequently adapted to the data-scarce `linalg` dialect. Performance is measured on two compilation tasks: training time and test accuracy prediction of neural network architectures from the NASBench dataset. The results are presented in Section 4.2.

4.1 Experimental Setup

This section describes the experimental setup used to investigate the following research questions:

RQ1: Zero-Shot Generalization. Can a model trained exclusively on a source dialect (`stablehlo`) accurately predict performance on a target dialect (`linalg`) zero-shot, or is explicit domain adaptation required to bridge the semantic gap?

RQ2: Data Efficiency. How does the model's quality improve as the number of programs from the target dialect increases?

RQ3: Transfer Effectiveness. Does transferring structural knowledge from a highly populated MLIR dialect (`stablehlo`) yield better predictive accuracy on a low-resource dialect (`linalg`) compared to training from scratch on the latter?

RQ4: Architectural Plasticity. Which components of the message-passing backbone encode universal graph semantics versus dialect-specific logic, and how does partially freezing the GNN impact domain adaptation?

RQ5: Initialization Strategy. How does the initialization strategy for new dialect embeddings (random initialization vs. mean-centering from the source domain) affect the stability and peak performance of the fine-tuning process?

RQ6: Time Efficiency. Does our cross-dialect domain adaptation reduce the computational cost and time-to-convergence required to reach peak performance?

RQ7: Generalization. Can the proposed transfer learning methodology generalize beyond the `stablehlo`–`linalg` dialect pair to other MLIR dialects?

The Prediction tasks. We evaluate the research questions on two compilation tasks:

Training time: Predicts the time required to train a neural network. This regression task is largely determined by explicit structural and data-flow properties of the program graph, such as tensor shapes, operation types, and computational complexity.

Test accuracy: Predicts the final accuracy of the architecture. In contrast to training time, this metric is non-linear and depends on implicit learning dynamics, making it a more challenging task that probes the model's ability to capture deeper structural patterns.

Benchmarks. To evaluate our domain adaptation pipeline, we use 423,624 unique neural network architectures from the NASBench dataset [Ying et al., 2019], which provides ground-truth training time and test accuracy metrics. We lower these architectures into two MLIR dialects to generate program graphs and simulate domain transfer. To emulate a realistic data-scarce setting for a new compiler dialect, we partition the training pool using a 90/10 split across the two dialects:

StableHLO: A high-level dialect for representing machine learning models from frameworks such as JAX and TensorFlow. We use `stablehlo` as the source domain, allocating 90% of the training data to it. This provides a controlled setting where the GNN can learn structural and data-flow patterns prior to transfer.

Linalg: A dialect providing standard linear algebra operations (e.g., matrix multiplication, addition, and transposition). It serves as the target domain, with its training set restricted to the remaining 10% to simulate a data-scarce regime for cross-dialect adaptation.

Hardware & Software. Experiments run on a workstation featuring Ubuntu 24.04.4 LTS, equipped with an NVIDIA GeForce RTX 5090 GPU (32GB VRAM), 128GB of system RAM, and an AMD Ryzen Threadripper 7970X 32-Core processor. The translation from MLIR to ProGraML graphs was performed using our custom C++ toolchain built on top of the LLVM/MLIR framework (version 22.0). The neural network models were implemented in Python using PyTorch (v2.10.0) and PyTorch Geometric (v2.7.0).

Neural Network Architecture. We employ a Gated Graph Neural Network (GGNN) following Cummins et al. [2021] for ProGraML graphs. Node features use learned embeddings of MLIR operations into 64-dimensional vectors, augmented with 32-dimensional sinusoidal positional encodings. Message passing runs for 30 iterations using Gated Recurrent Unit (GRU) updates. Messages are computed with edge-type-specific MLPs across six edge types (forward and backward control, data, and call edges), combined with position-dependent gating. Final node representations are aggregated via sum pooling and passed through a readout MLP to produce a scalar prediction. The StableHLO and Linalg models contain 62,081 and 63,041 trainable parameters, respectively, with the small difference attributable to dialect-specific vocabulary sizes. Targets are normalized with Z-score normalization during training and de-normalized at inference time. Optimization uses AdamW (5.0×10^{-4} learning rate) with weight decay and a ReduceLROnPlateau scheduler driven by validation Kendall's τ. Training uses a batch size of 128, mixed-precision arithmetic, and gradient clipping.

Predictors. We evaluate several predictors, ranging from naive statistical models to neural network ablations, to isolate the effects of our pipeline. For our domain adaptation experiments (FTE 1–8), we load the pre-trained **StableHLO-Source** backbone and introduce the `linalg` dialect. We study two factors: (i) architectural plasticity (layer freezing) and (ii) embedding initialization strategy (random vs. mean-centered).

Mean Predictor: A naive model that always predicts the mean of the target variable from the training set, establishing a lower-bound performance baseline.

Linear Regression: A classical baseline that predicts performance solely from the number of model parameters, highlighting the limitations of relying on a single coarse structural feature.

StableHLO-Source: A GNN trained exclusively on the `stablehlo` source domain. It establishes peak source-domain performance and is evaluated on `linalg` to measure *zero-shot generalization*.

Linalg-Target: A GNN trained from random initialization on the limited `linalg` dataset, representing a data-constrained setting without transfer learning.

FTE 1 (Frozen Backbone): All message-passing layers are frozen; only the randomly initialized dialect embeddings and the readout MLP are actively trained.

FTE 2 (Frozen Edge & Position): Both the Edge and Position routing MLPs are frozen to retain universal relational and spatial semantics. The GRU state-update cells, embeddings, and readout layers are fine-tuned.

FTE 3 (Frozen Edge): Only the Edge MLPs are frozen. The Position MLPs are fine-tuned alongside the GRU cells, embeddings, and readout layers, allowing the spatial logic to adapt to the target dialect.

FTE 4 (Full Fine-Tuning): The `linalg` dialect embeddings are randomly initialized, and all parameters are actively fine-tuned without any freezing constraints.

FTE 5–8 (Mean Initialization): Same as FTE 1–4, but `linalg` embeddings are initialized using the mean vector of pre-trained **StableHLO-Source** embeddings.

4.2 Research Questions

This section evaluates the proposed transfer-learning methodology using the experimental setup described in Section 4.1. The discussion is organized around the research questions introduced earlier. Table 4.1 summarizes quantitative results.

Table 4.1: NASBench Dataset – Training Time (Top, seconds) and Test Accuracy (Bottom, 0–1) – 90% `stablehlo` Pre-training with 10% Linalg Fine-tuning – Test Set Metrics (RMSE, R^2 and Kendall's τ) and Epoch Training Details.

	Baseline				Random Embedding Init				Smart Embedding Init (Mean)				
Metric	Mean Pred.	Lin. Reg.	StableHLO-Source	Zero-Shot	FTE1	FTE2	FTE3	FTE4	FTE5	FTE6	FTE7	FTE8	Linalg-Target
RMSE	919.5	374.6	105.4	5645.5	419.3	116.7	107.1	296.8	125.9	109.9	113.2	**105.2**	290.1
R^2	0.0	0.834	0.987	-37.487	0.788	0.984	0.986	0.894	0.981	0.985	0.985	**0.987**	0.898
Kendall's τ	0.0	0.704	0.976	0.276	0.675	0.958	0.958	0.834	0.924	0.963	0.958	**0.964**	0.830
Epoch Time	–	–	2m34s	–	31s	31s	33s	33s	31s	32s	33s	**35s**	34s
Total Time	–	–	1h42m	–	52m04s	52m21s	55m35s	55m48s	52m20s	54m47s	56m32s	**57m34s**	57m19s
RMSE	0.058	0.057	0.033	0.271	0.054	0.045	0.044	0.046	0.050	0.043	**0.043**	0.045	0.046
R^2	0.0	0.022	0.684	-18.918	0.209	0.461	0.482	0.425	0.318	0.492	**0.498**	0.446	0.413
Kendall's τ	0.0	0.307	0.795	-0.086	0.513	0.683	0.684	0.696	0.504	0.704	**0.721**	0.681	0.658
Epoch Time	–	–	2m33s	–	31s	31s	33s	35s	31s	33s	**35s**	36s	35s
Total Time	–	–	2h33m	–	52m 25s	53m 01s	55m 46s	57m 34s	53m00s	55m03s	**58m04s**	59m48s	56m39s

4.2.1 RQ1: Zero-Shot Generalization

We first investigate whether a model trained exclusively on a source dialect can predict performance metrics for a target dialect without any exposure to target dialect samples. The goal of this evaluation is to establish a lower bound for transfer learning.

Discussion: The Zero-Shot results in Table 4.1 show that direct transfer fails, as Example 1.5 already explained. For training time prediction, the model achieves RMSE $= 5645.5$ and $R^2 = -37.487$, substantially worse than the mean predictor baseline. A similar trend appears for test accuracy prediction. These strongly negative R^2 values

indicate severe miscalibration caused by the mismatch between the distributions of `stablehlo` and `linalg` programs. Nevertheless, some ranking information survives the transfer. For training time prediction, Kendall's $\tau = 0.276$ remains above zero, suggesting that the model captures transferable structural information even though absolute predictions are unreliable. For test accuracy, however, even the ranking signal collapses ($\tau = -0.086$).

Conclusion: At least in our experimental setup, a model trained on one MLIR dialect cannot be used "as-is" to make predictions on another. Without explicit domain adaptation, the model's absolute predictions are effectively useless, often performing worse than a simple mean baseline.

4.2.2 RQ2: Data Efficiency

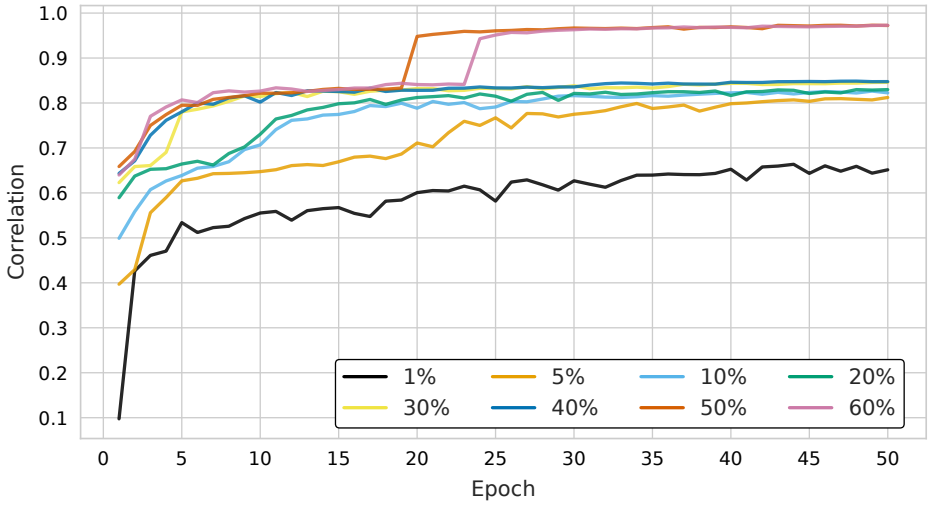
To evaluate the data efficiency of training from scratch, we incrementally scale the volume of target-domain data available to the baseline model. Specifically, we train independent instances of the **Linalg-Target** predictor on subsets corresponding to 1% through 60% of the *total* dataset. The remaining 40% of the data is held constant across all experiments and is split into validation and test sets (20%/20%).

To account for sampling variability, each data regime is evaluated over three independent runs, where the subset is randomly sampled each time. From these three runs, we report the best-performing model according to the validation Kendall's τ. We monitor Kendall's τ rank correlation over training to assess how effectively an uninitialized architecture can learn execution semantics of the `linalg` dialect as data availability increases. While the figures report training and validation performance for clarity, all selected models were also evaluated on the held-out test set, where we observed results consistent with validation trends, indicating good generalization and no major evidence of overfitting.

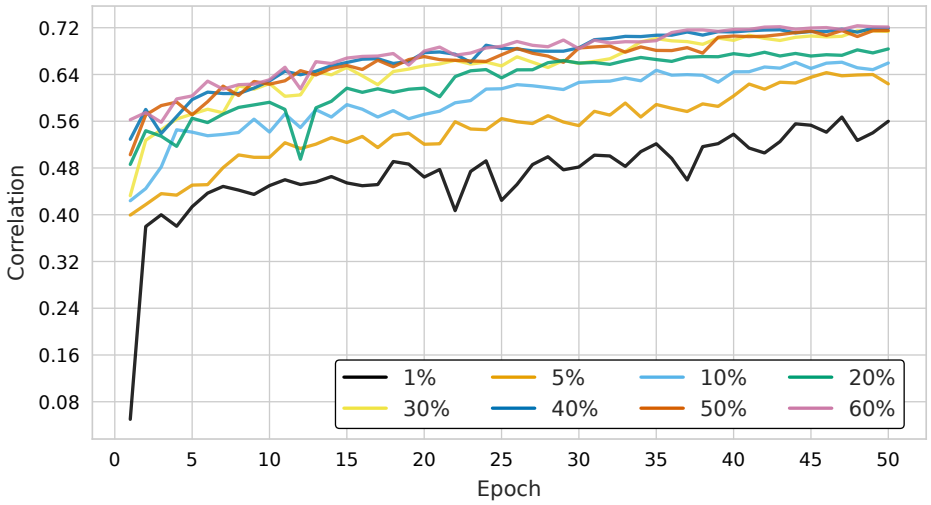
Discussion: Figure 4.1 shows that training from scratch is highly data-dependent. For the training time task (Figure 4.1a), the model exhibits high variance and limited predictive power in extreme low-data regimes (1%–5%), indicating it struggles to infer meaningful graph semantics from sparse supervision. As dataset size increases (10%–40%), performance improves but enters a broad plateau, suggesting the model captures partial structural regularities without fully generalizing.

A sharp improvement is observed in the higher data regimes (50% and 60%), where the model converges to near-perfect rank correlation. This behavior suggests the presence of a *data threshold* beyond which the model can reliably internalize the

Figure 4.1: Validation Kendall's τ (higher is better) as a function of training data size (1%–60%) for the `linalg`-target model trained from scratch. Each point corresponds to the best result across three random subset samples.



(a) Training time prediction.



(b) Test accuracy prediction.

Source: Created by the author.

execution semantics of the target dialect. However, this transition is not necessarily tied to these exact percentages; given the stochastic nature of training and data sampling, similar breakthroughs could occur at intermediate points (e.g., 30% or 40%) under different conditions.

A similar pattern emerges for the test accuracy task (Figure 4.1b), though with a lower absolute ceiling due to the inherently more complex and non-linear nature of the prediction target. While performance steadily improves with additional data, the model remains significantly constrained in low-resource settings, failing to achieve a strong correlation until larger portions of the dataset are available.

Conclusion: Training from scratch on a new MLIR dialect is highly data-inefficient. Competitive performance only emerges after a significant data threshold is met (in our setup, $> 50\%$ of the dataset), leaving the model crippled in the low-resource regimes (1–10%).

4.2.3 RQ3: Transfer Effectiveness

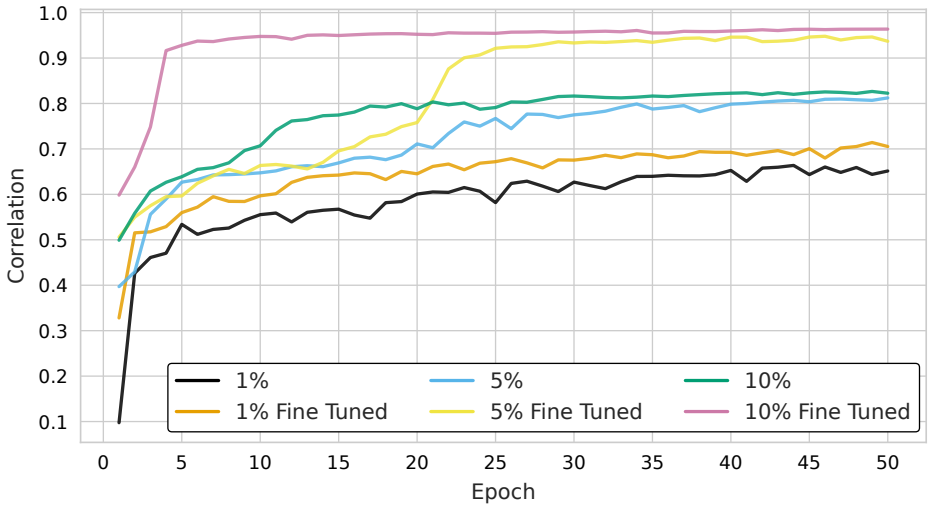
We now evaluate the proposed transfer-learning methodology. A model is first pre-trained on a large corpus H of `stablehlo` programs and then fine-tuned on a smaller set L of `linalg` programs. We compare this approach against models trained from scratch on L under low-resource regimes of 1%, 5%, and 10% of the available data. Unless otherwise noted, fine-tuned results use the FTE6 configuration, which employs smart embedding initialization and partially freezes the message-passing layers (edge and positional MLPs).

Discussion: Figures 4.2a and 4.2b show that transfer learning consistently outperforms training from scratch across all low-resource regimes. For training time prediction, the advantage becomes increasingly pronounced as more target-domain data becomes available: fine-tuned models converge faster and achieve substantially higher Kendall's τ values than scratch-trained models. The gains are smaller in the 1% regime, suggesting that some minimum amount of target data is necessary for effective adaptation.

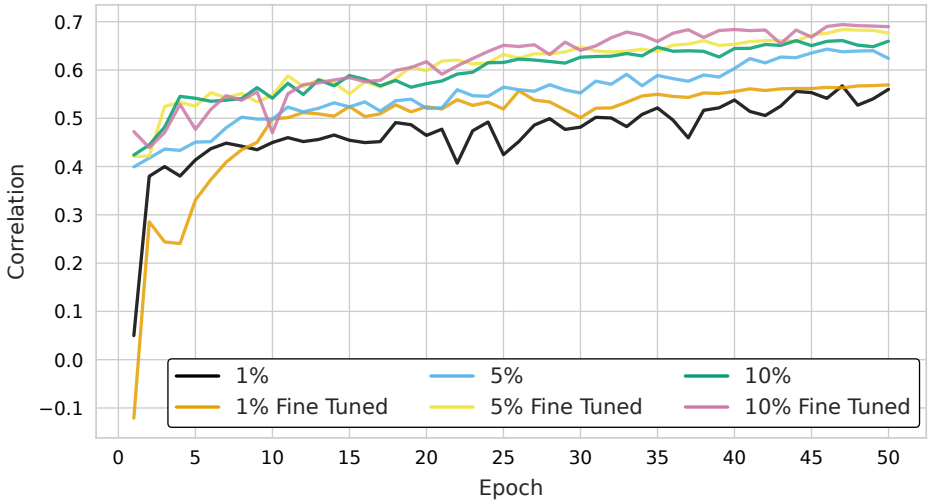
A similar trend appears for test accuracy prediction, although with lower overall correlation values and smaller margins between transfer and scratch training. Nevertheless, fine-tuned models consistently outperform scratch-trained baselines across all regimes. These trends are confirmed by the quantitative results in Table 4.1. For training time prediction, several fine-tuning configurations (e.g., FTE2, FTE3, FTE6, FTE7, FTE8) achieve RMSE and Kendall's τ values close to those obtained by the `stablehlo`-trained model ($\tau = 0.976$), and consistently outperform the `linalg` scratch baseline ($\tau = 0.830$). For test accuracy prediction, fine-tuned models also outperform the `linalg` scratch baseline across R^2 and Kendall's τ, although the margins are smaller and the overall performance remains lower compared to training time prediction.

Conclusion: Transfer learning consistently outperforms training from scratch across all data-scarce scenarios. By leveraging pre-trained structural representations, the model achieves higher predictive accuracy with limited target data, mitigating the data-scarcity bottleneck in MLIR.

Figure 4.2: Comparison of Kendall's τ (higher is better) progression between models trained from scratch on `linalg` and models fine-tuned from `stablehlo` under extreme data-scarce scenarios (1%, 5%, and 10% of total data).



(a) Training time prediction.



(b) Test accuracy prediction.

Source: Created by the author.

4.2.4 RQ4: Architectural Plasticity

We now investigate which components of the GNN encode transferable structural information and which must adapt to dialect-specific semantics. The eight fine-tuning configurations (FTE1–FTE8) differ in two aspects: the network components frozen during adaptation and the initialization strategy for target-dialect embeddings. This section focuses on the effect of freezing strategies; initialization is analyzed separately in RQ5.

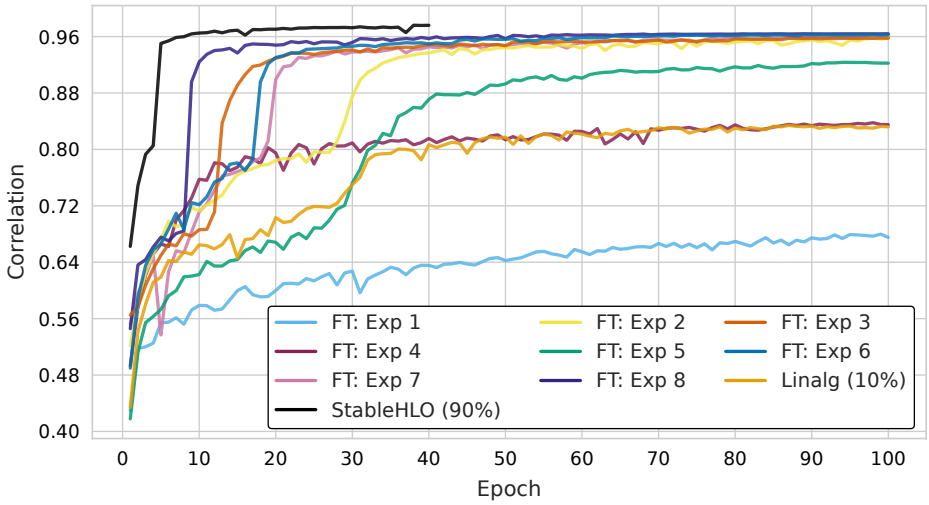
Discussion: Figures 4.3a and 4.3b show the validation Kendall's τ curves across 100 epochs for all eight fine-tuning configurations, alongside the `stablehlo` pre-trained model and the `linalg` scratch baseline, for training time and test accuracy, respectively. For training time (Figure 4.3a), the impact of freezing strategies is clearly differentiated across configurations. FTE1 performs the worst by a large margin ($\tau \approx 0.67$), indicating that freezing the entire message-passing backbone severely limits adaptation. Interestingly, full fine-tuning (FTE4) does not yield strong performance, plateauing at $\tau \approx 0.83$, close to the `linalg` scratch baseline. This highlights a possible loss of useful pre-trained structure during adaptation, consistent with catastrophic forgetting. FTE5 improves substantially over FTE1, reaching $\tau \approx 0.92$, likely due to the more informative initialization, but still falls short of the best-performing configurations. The strongest results are obtained by FTE2, FTE3, FTE6, FTE7, and FTE8, all of which achieve τ values close to the `stablehlo`-trained model. These results indicate that effective transfer requires preserving useful pre-trained structure during adaptation, either through selective freezing or through informed initialization, rather than relying on unconstrained fine-tuning, which can overwrite useful pre-trained representations.

For test accuracy (Figure 4.3b), a similar overall ordering is observed, but with lower correlation values across all configurations. FTE1 and FTE5 perform consistently worse than the other strategies, remaining well below the rest throughout training, indicating that freezing the entire message-passing backbone limits adaptation even when improved initialization is used. Among the remaining configurations, fine-tuned models outperform the `linalg` scratch baseline, although the gap is smaller than for training time prediction. This suggests that, while transfer remains beneficial, its impact is reduced for this more challenging task. Notably, none of the fine-tuning strategies match the performance of the `stablehlo`-trained model, which consistently achieves higher correlation. This may indicate that the representations learned in the source domain are more directly aligned with this prediction task, and are not fully recoverable through fine-tuning on the target-domain data.

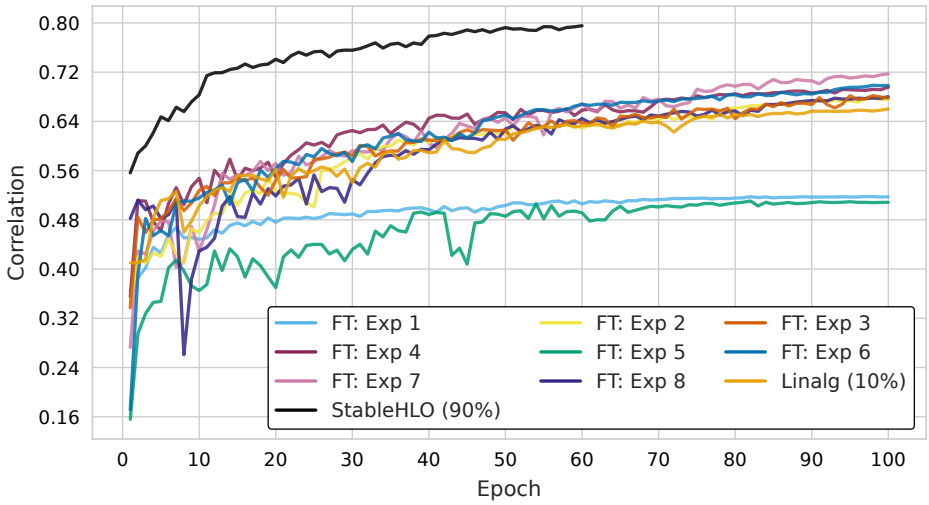
Table 4.1 provides a quantitative summary that reinforces the trends observed in the figures across multiple evaluation metrics. The relative ordering of configurations is consistent across RMSE, R^2, and Kendall's τ, indicating that the observed effects are robust and not specific to a single metric. Across both tasks, configurations that allow adaptation of the message-passing layers consistently achieve the strongest results, while more constrained strategies degrade performance. Importantly, the differences between configurations are more pronounced for training time than for test accuracy, reflecting the greater difficulty of the latter task.

Conclusion: Among the evaluated configurations, FTE6 and FTE7 provide the most favorable trade-off. This indicates that selective freezing is superior to full fine-tuning.

Figure 4.3: Validation Kendall's τ (higher is better) across training epochs for all fine-tuning configurations (FTE1–FTE8), compared with the `stablehlo` pre-trained model and the `linalg` scratch baseline.



(a) Training time prediction.



(b) Test accuracy prediction.

Source: Created by the author.

Adaptation must be balanced; freezing the entire backbone prevents the model from learning dialect-specific logic, while unconstrained fine-tuning leads to "catastrophic forgetting" of the universal program structures learned during pre-training.

4.2.5 RQ5: Initialization Strategy

We now evaluate the impact of embedding initialization on transfer performance. We compare random initialization against mean initialization, where target-dialect embeddings are initialized using the average vector of the pre-trained `stablehlo` embed-

ding space. Configurations FTE1–FTE4 use random initialization, whereas FTE5–FTE8 use mean initialization under equivalent freezing strategies.

Discussion: Figures 4.3a and 4.3b, together with Table 4.1, show that mean initialization consistently improves performance, although the magnitude of the effect depends on the task and adaptation strategy. For training time prediction, initialization has the strongest impact in constrained configurations. For example, FTE5 substantially improves over FTE1, indicating that better initialization can partially compensate for a frozen backbone. Similarly, mean initialization mitigates the degradation observed under full fine-tuning (FTE8 vs. FTE4). In contrast, configurations that already adapt effectively (e.g., FTE6 and FTE7) benefit only marginally.

For test accuracy prediction, the effect of initialization is more modest. Improvements are primarily observed in FTE6 and FTE7, where mean initialization provides small but consistent gains. However, the overall ranking of configurations remains largely unchanged, suggesting that initialization plays a secondary role compared to architectural adaptability for this task.

Conclusion: Smart initialization (mean-centering) is a low-cost insurance policy for model stability. It significantly boosts performance in constrained backbones and mitigates the failures associated with random embedding initialization.

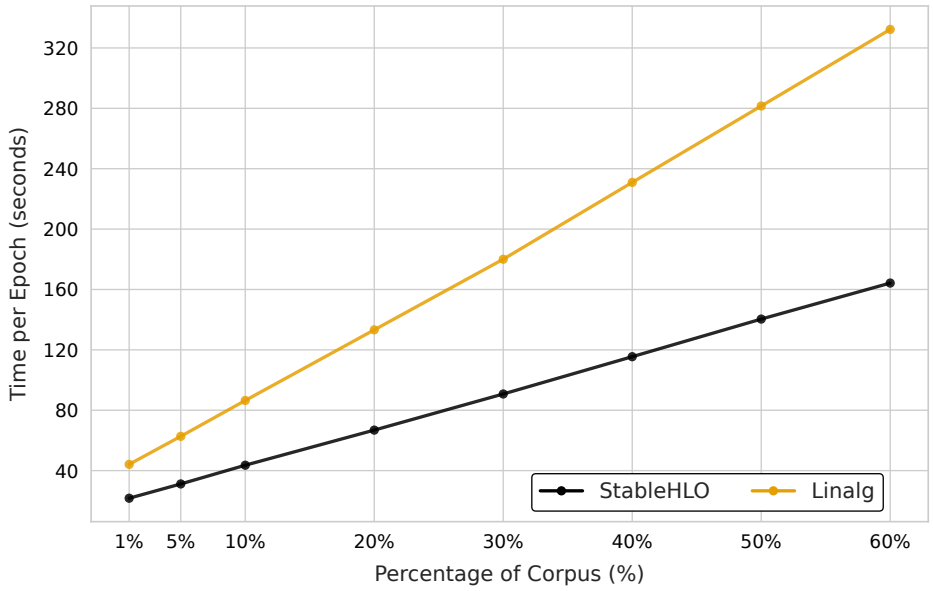
4.2.6 RQ6: Time Efficiency

We now evaluate the computational cost of the proposed transfer-learning pipeline. First, we compare the cost of processing different dialects; then, we analyze the wall-clock time required to reach comparable predictive quality.

Discussion: Figure 4.4 compares the time per epoch for models trained from scratch on `stablehlo` and `linalg` across varying dataset sizes. `linalg` consistently exhibits a higher computational cost, with per-epoch times approximately twice as large as those of `stablehlo`. This difference arises from the lowering process itself: transitioning from `stablehlo` to `linalg` increases the number of operations, resulting in larger and more densely connected program graphs. As a result, message passing becomes more expensive, increasing both computation and memory requirements. This behavior is expected to generalize beyond this setting, as lowering in compiler pipelines typically expands programs into more granular representations, systematically increasing processing costs. In this context, pre-training on higher-level representations and transferring to lower-level dialects can be a practical strategy to reduce the overall

computational cost of learning in lower-level dialects. Although there is a difference in trainable parameters (62,081 vs. 63,041), it is too small to account for the observed performance gap.

Figure 4.4: Time per epoch as a function of dataset size for models trained from scratch on `stablehlo` and `linalg`.



Source: Created by the author.

Table 4.2 complements this analysis by evaluating time efficiency under an iso-accuracy setting. Instead of comparing raw training times, we measure the wall-clock time required to reach fixed fractions (90%, 95%, and 99%) of each model's peak validation Kendall's τ. We use the `linalg` models identified in RQ2 as the first to reach performance levels comparable to the fine-tuned configurations, which correspond to larger target-domain data regimes (50% and 30%, respectively). In contrast, fine-tuned models operate with only 10% of the data. This setup allows us to compare the time required to achieve similar predictive quality rather than equal data budgets, providing a normalized view of convergence efficiency.

For training time prediction, transfer-based approaches achieve comparable performance significantly faster than training from scratch. For example, the `linalg` baseline requires over two hours to reach $\tau \approx 0.96$, whereas FTE configurations reach similar performance levels in under one hour, despite using only 10% of the data. In particular, FTE8 achieves $\tau = 0.957$ in 48 minutes, demonstrating that transfer can substantially reduce the time required to reach high-quality predictions.

For test accuracy prediction, the advantage of transfer is less pronounced but remains evident. While the `linalg` baseline is competitive in absolute time at lower thresholds, transfer-based methods achieve similar or slightly higher peak performance using significantly less data. Configurations such as FTE7 reach the highest observed τ

Table 4.2: Time-to-performance analysis for training time (top) and test accuracy (bottom). Wall-clock time to reach 90%, 95%, and 99% of peak validation Kendall's τ. For fair comparison, baseline models utilize larger target-domain data volumes (50% and 30%, respectively) to reach comparable performance levels, while fine-tuned (FTE) models use only 10%. FTE compute is split into `stablehlo` pre-training (**Pre-Training**) and target-dialect computation (**Linalg**).

		90% of Peak τ				95% of Peak τ				99% of Peak τ				
Predictor	**Train Data (%)**	**Pre-Training**	**Linalg**	**Total**	**Val. τ**	**Pre-Training**	**Linalg**	**Total**	**Val. τ**	**Pre-Training**	**Linalg**	**Total**	**Val. τ**	**Peak Val. τ**
Linalg (Scratch)	50%	–	1h 38m 52s	1h 38m 52s	0.948	–	1h 38m 52s	1h 38m 52s	0.948	–	2h 11m 53s	2h 11m 53s	0.963	0.973
FTE6 (Transfer)	10%	12m 26s	9m 39s	22m 06s	0.896	12m 26s	10m 11s	22m 38s	0.921	30m 16s	23m 42s	53m 58s	0.955	0.963
FTE7 (Transfer)	10%	12m 26s	11m 49s	24m 15s	0.899	12m 26s	12m 24s	24m 50s	0.917	30m 16s	30m 53s	1h 01m 09s	0.950	0.959
FTE8 (Transfer)	10%	12m 26s	5m 10s	17m 37s	0.896	12m 26s	5m 45s	18m 11s	0.924	30m 16s	17m 47s	48m 03s	0.957	0.964
Linalg (Scratch)	30%	–	40m 43s	40m 43s	0.645	–	1h 47m 55s	1h 47m 55s	0.679	–	2h 38m 23s	2h 38m 23s	0.715	0.715
FTE6 (Transfer)	10%	30m 05s	25m 37s	55m 43s	0.641	1h 13m 29s	32m 14s	1h 45m 43s	0.665	1h 54m 24s	51m 07s	2h 45m 31s	0.692	0.699
FTE7 (Transfer)	10%	30m 05s	30m 06s	1h 00m 12s	0.648	1h 13m 29s	43m 29s	1h 56m 58s	0.692	1h 54m 24s	53m 59s	2h 48m 23s	0.713	0.717
FTE8 (Transfer)	10%	30m 05s	26m 31s	56m 37s	0.613	1h 13m 29s	39m 51s	1h 53m 20s	0.647	1h 54m 24s	54m 22s	2h 48m 46s	0.674	0.680

while maintaining comparable end-to-end training times, indicating a favorable trade-off between efficiency and accuracy.

One important consideration is that the discussion above accounts for the full end-to-end cost of pre-training and fine-tuning. In practice, however, pre-training can often be treated as an offline cost, amortized across multiple downstream tasks. Under this perspective, only the fine-tuning stage contributes to the effective deployment cost. This significantly strengthens the efficiency of transfer-based approaches. For instance, in Table 4.2, FTE8 reaches $\tau = 0.957$ in 17m 47s of `linalg` compute, compared to over 2 hours required by the `linalg` scratch baseline to achieve similar performance. Even for test accuracy, fine-tuning times remain competitive while achieving higher peak τ. This highlights that, when amortizing pre-training, transfer not only improves performance but also substantially reduces the practical time-to-solution, particularly in settings where lowering increases the number of operations and program size, amplifying the cost gap between source and target dialects.

Conclusion: Transfer learning reduces both data requirements and time-to-solution. Fine-tuned models achieve high predictive accuracy substantially faster than models trained from scratch, particularly for lower-level dialects with larger graph representations.

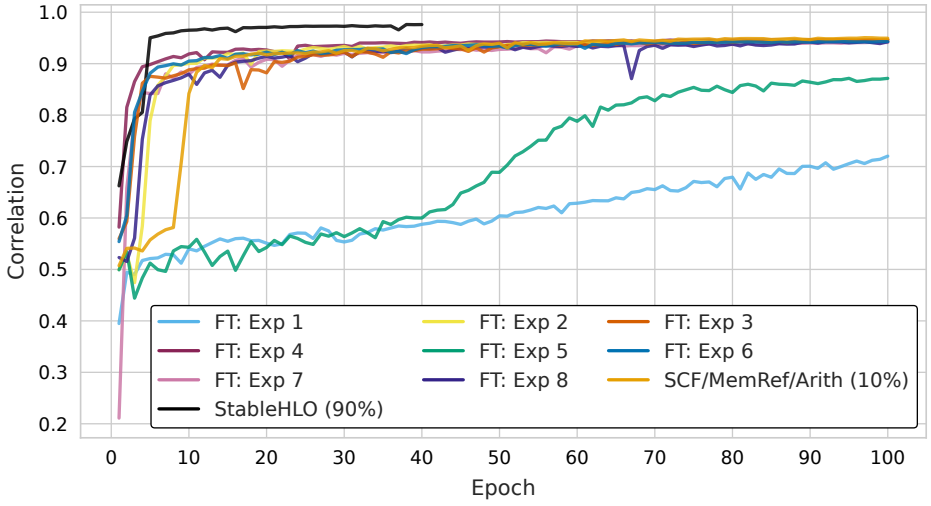
4.2.7 RQ7: Generalization

Thus far, we have demonstrated transfer learning from `stablehlo` to `linalg`. This section evaluates the effectiveness of this approach on dialects more distant from `stablehlo` on the lowering pipeline. Notice that extending our pipeline to a new dialect requires only implementing the corresponding MLIR-to-ProGraML bindings. Because the pipeline is built on MLIR's shared abstractions, dialects that already integrate with

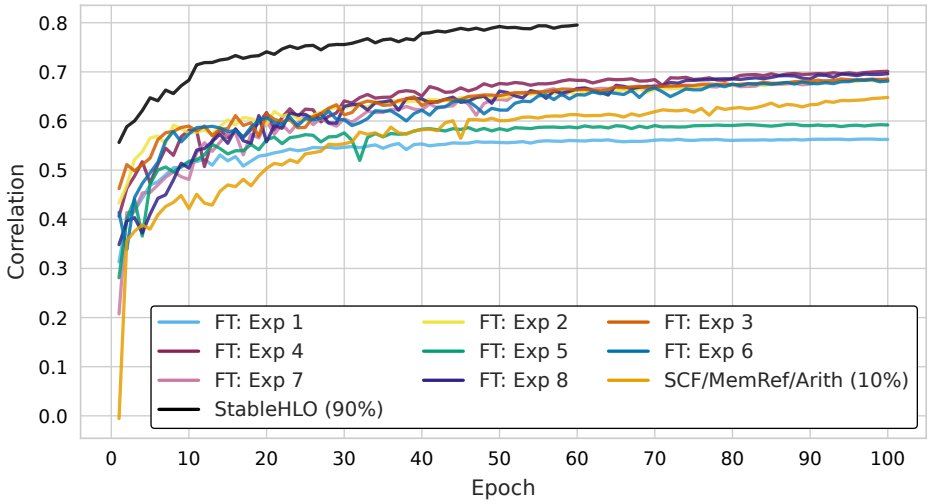
MLIR's infrastructure are supported out of the box.

Discussion: We apply the pipeline to a lower-level MLIR representation comprising the `scf`, `memref`, and `arith` dialects, using the same 90/10 split with `stablehlo` as the source. Figures 4.5a and 4.5b show that transfer learning converges slightly faster for training time prediction and outperforms the scratch baseline for test accuracy, demonstrating that the methodology generalizes to additional MLIR dialects.

Figure 4.5: Validation Kendall's τ across training epochs for all fine-tuning configurations (FTE1–FTE8) applied to the `scf/memref/arith` target dialect combination, compared with the `stablehlo` pre-trained model and baseline.



(a) Training time prediction.



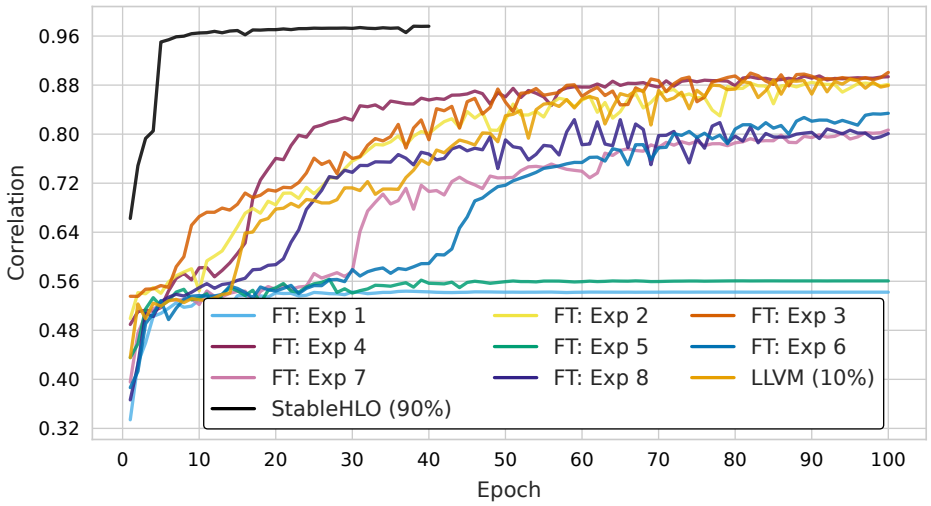
(b) Test accuracy prediction.

Source: Created by the author.

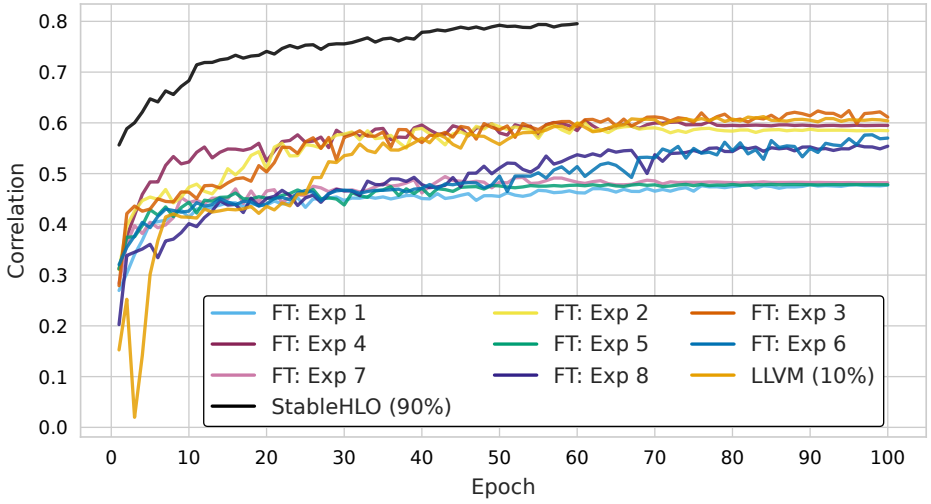
We also evaluated transfer to the `LLVM` dialect, as shown in Figures 4.6a and 4.6b. In this setting, fine-tuning was not successful in improving results compared to the baseline. However, in this case, the gap between dialects is enormous. While `stablehlo`

programs typically contain only a few hundred operations, the corresponding LLVM representations often expand to thousands of operations. This structural mismatch appears to exceed the range over which transfer learning remains effective.

Figure 4.6: Validation Kendall's τ across training epochs for all fine-tuning configurations (FTE1–FTE8) applied to the `LLVM` target dialect, compared with the `stablehlo` pre-trained model and baseline.



(a) Training time prediction.



(b) Test accuracy prediction.

Source: Created by the author.

Conclusion: The methodology generalizes to new dialects with minimal engineering effort and retains benefits for moderately distant targets such as `scf`/`memref`/`arith`. Transfer breaks down for very distant dialects like `LLVM`, where the representation gap becomes too large.

Chapter 5

Conclusion

Applying machine learning to emerging compiler dialects is often constrained by a lack of labeled training data. This thesis addresses that challenge for MLIR through a cross-dialect domain adaptation methodology that combines ProGraML graph representations with a Gated Graph Neural Network and selective fine-tuning. The methodology was evaluated on two prediction tasks, training time and test accuracy, across 423,624 neural network architectures from the NASBench dataset, using `stablehlo` as a data-rich source dialect and `linalg` as a data-scarce target dialect.

The experimental results, organized around seven research questions, yield several conclusions. First, zero-shot transfer from one dialect to another fails: a model trained on `stablehlo` and applied directly to `linalg` produces predictions worse than a trivial mean baseline (RQ1). Second, training from scratch on a new dialect is highly data-inefficient, with competitive performance emerging only after a substantial fraction of the dataset is available (RQ2). Third, transfer learning consistently outperforms training from scratch across all data-scarce regimes, demonstrating that pre-trained structural representations provide a useful initialization even when the operation vocabulary changes entirely (RQ3). Fourth, the effectiveness of transfer depends critically on which model components are frozen during adaptation: selective freezing of the edge and positional MLPs preserves reusable structural knowledge while allowing the GRU and embedding layers to adapt to the target dialect, whereas full fine-tuning leads to potential catastrophic forgetting (RQ4). Fifth, initializing target-dialect embeddings from the mean of the source embedding space improves stability and peak performance, particularly in constrained configurations (RQ5). Sixth, the transfer-based pipeline reduces both data requirements and wall-clock time: fine-tuned models reach high predictive accuracy in under one hour with 10% of the data, compared to over two hours for scratch-trained models using 50% of the data (RQ6). Finally, the methodology generalizes to additional MLIR dialects at moderate structural distances (e.g., `scf`/`memref`/`arith`), but transfer breaks down for very distant dialects like `LLVM`, where the representation gap becomes too large (RQ7).

Beyond the experimental findings, this thesis makes three broader contributions. It establishes cross-dialect transfer learning as a viable methodology for MLIR, showing

that the structural similarities preserved by MLIR's progressive lowering pipeline can be exploited by graph neural networks. It identifies a principled decomposition of GNN parameters into transferable and dialect-specific groups, providing practical guidance for practitioners adapting models to new dialects. And it provides an open-source toolchain that extends ProGraML from LLVM-IR to MLIR, enabling other researchers to apply the same methodology to their own dialects with minimal effort.

As MLIR continues to expand as a platform for domain-specific compilation, data scarcity will likely remain a recurring challenge for new dialects. These results suggest that transfer learning over semantics-aware graph representations is a practical and effective strategy for building predictive models in such settings.

5.1 Limitations

While the results presented in this thesis are encouraging, several limitations should be noted.

The evaluation relies on a single benchmark suite (NASBench), in which all architectures are drawn from the same cell-based search space. Although the dataset contains over 400,000 unique architectures and exhibits considerable structural variation within its search space, it remains an open question whether the observed transfer benefits generalize to programs from fundamentally different application domains, such as signal processing, simulation codes, or general-purpose software.

The prediction tasks evaluated, training time and test accuracy of neural network architectures, are properties of the programs themselves rather than compilation decisions. While they serve as controlled proxies for stochastic compilation tasks, the thesis does not directly evaluate the methodology on optimization-selection tasks such as choosing tile sizes, selecting vectorization strategies, or predicting the profitability of specific compiler transformations.

All experiments use a single source dialect (`stablehlo`). The analysis therefore does not address whether combining multiple source dialects could yield better transfer, or whether the choice of source dialect matters (e.g., whether training on a mid-level dialect like `linalg` and transferring to `scf`/`memref`/`arith` would outperform transfer from the high-level `stablehlo`).

Furthermore, the evaluation considers a limited set of target dialects and dialect combinations. Additional experiments involving a broader range of MLIR dialects would be necessary to determine how broadly the observed transfer patterns generalize across the MLIR ecosystem.

The GNN models used in this work are intentionally small (approximately 62,000

parameters) to match the ProGraML architecture of Cummins et al. [2021]. While this choice provides a fair comparison with prior work, it leaves open the question of whether larger GNNs would exhibit the same transfer characteristics, or whether increased model capacity would enable more robust cross-dialect representation learning.

The failure of transfer to the `LLVM` dialect (RQ7) reveals a fundamental boundary of the approach: when the structural distance between source and target dialects is too large, with programs expanding from hundreds to thousands of operations, the shared representations learned during pre-training appear insufficient to bridge the gap. The thesis does not provide a formal characterization of this boundary, nor a method to predict in advance whether transfer to a given target dialect will be beneficial.

Finally, all experiments were conducted on a single hardware configuration. Although the conclusions concern predictive performance rather than raw execution speed, different hardware platforms may affect training times, resource utilization, and the practical scalability of the approach.

5.2 Future Work

Several directions emerge naturally from the findings and limitations of this work.

A first direction is to evaluate the methodology on downstream compilation optimization tasks. Predicting training time and test accuracy demonstrates that cross-dialect transfer captures meaningful program structure, but the ultimate goal of learned compiler models is to support optimization decisions. Applying the pipeline to tasks such as tile-size selection, operation scheduling, vectorization decisions, or hardware mapping would provide a more direct assessment of whether the transferred representations encode information that is useful for compiler optimization.

A second direction is to explore multi-source transfer, where the model is pre-trained on multiple source dialects simultaneously or sequentially. MLIR's progressive lowering pipeline produces a chain of related representations (`stablehlo` → `linalg` → `scf`/`memref`/`arith` → `LLVM`), and training on multiple levels of abstraction could produce more robust representations that transfer effectively across a wider range of target dialects.

A third direction concerns more sophisticated adaptation techniques. The selective freezing strategy evaluated in this thesis makes a binary decision in which each parameter group is either fully frozen or fully trainable. Softer regularization approaches, such as Elastic Weight Consolidation [Kirkpatrick et al., 2017], parameter-efficient fine-tuning methods such as AdapterGNN [Li et al., 2024], or low-rank adaptation

techniques could offer finer-grained control over the adaptation process and potentially improve transfer to more distant dialects where the current approach fails.

A fourth direction is to investigate the concept of *dialect distance* more formally. The experiments in RQ7 suggest that transfer effectiveness degrades as the structural gap between source and target dialects increases, but the thesis does not provide a metric for quantifying this distance. Developing such a metric, perhaps based on graph-level statistics such as operation counts, graph density, or embedding-space divergence, would enable practitioners to estimate transferability before collecting target-domain data and performing fine-tuning experiments.

Another promising direction is to study the interaction between model scale and transfer learning. The experiments in this thesis employ a relatively small GNN architecture. Larger models may learn more general representations that transfer across dialects more effectively, potentially enabling successful adaptation even in cases where transfer currently breaks down, such as the `LLVM` dialect.

Finally, a natural extension is to apply the methodology beyond the lowering pipeline to dialect pairs that are not connected by a direct lowering relationship. MLIR hosts dialects for diverse domains, including hardware description (e.g., `hw`), sparse tensor computation (e.g., `sparse_tensor`), GPU programming (e.g., `gpu`), and quantum computing. Understanding whether transfer learning remains effective across these domain boundaries would help establish whether dialect transfer is primarily a consequence of shared lowering semantics or a more general property of MLIR's representation framework.

More broadly, the long-term vision suggested by this work is the development of foundation models for compiler intermediate representations. Rather than training separate models for each dialect and task, future systems could pre-train on large and diverse collections of MLIR programs and subsequently adapt to new dialects, optimization objectives, and hardware targets with minimal additional data. The results presented in this thesis provide initial evidence that such transfer is feasible, while also highlighting the challenges that must be overcome to realize this goal.

References

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), September 2018. ISSN 0360-0300. doi: 10.1145/3197978. URL https://doi.org/10.1145/3197978.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, Red Hook, NY, USA, 2018. Curran Associates, Inc. URL https://api.semanticscholar.org/CorpusID:49314097.

Charlie Blake, Vitaly Kurin, Maximilian Igl, and Shimon Whiteson. Snowflake: Scaling GNNs to high-dimensional continuous control via parameter freezing. In *Advances in Neural Information Processing Systems*, Red Hook, NY, USA, 2021. Curran Associates, Inc. URL https://openreview.net/forum?id=REjT_c1Eejk.

John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, page 183–194, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138075. doi: 10.1145/996841.996864. URL https://doi.org/10.1145/996841.996864.

John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *SC*, page 14, USA, 2005. IEEE Computer Society. ISBN 1595930612. doi: 10.1109/SC.2005.14. URL https://doi.org/10.1109/SC.2005.14.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, Portland, OR, USA, Sept 2017. IEEE. doi: 10.1109/pact.2017.24. URL http://dx.doi.org/10.1109/PACT.2017.24.

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253, Online, 18–24 Jul 2021. PMLR. URL https://proceedings.mlr.press/v139/cummins21a.html.

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. Anghabench: a suite with one million compilable c benchmarks for code-size reduction. In *CGO*, page 378–390, Virtual Event, Republic of Korea, 2021. IEEE Press. ISBN 9781728186139. doi: 10.1109/CGO51591.2021.9370322. URL https://doi.org/10.1109/CGO51591.2021.9370322.

Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintao Pereira, Nilton Queiroz, and Otavio Oliveira Napoli. Program representations for predictive compilation: State of affairs in the early 20's. *Journal of Computer Languages*, 73(101153): 101171, apr 2022. doi: 10.1016/j.cola.2022.101171.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, Ottawa, Canada, 2008. GNU Tools Cauldron.

Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke. Towards a high-performance AI compiler with upstream MLIR. *CoRR*, abs/2404.15204, 2024. URL https://arxiv.org/abs/2404.15204.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th In-*

ternational Conference on Learning Representations (ICLR), Virtual Event, Austria, 2021. OpenReview.net. URL https://openreview.net/forum?id=jLoC4ez43PZ.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. doi: 10.1073/pnas.1611835114. URL https://www.pnas.org/doi/abs/10.1073/pnas.1611835114.

Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 147–162, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315616. doi: 10.1145/2384616.2384628. URL https://doi.org/10.1145/2384616.2384628.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Washington, DC, USA, 2021. IEEE. doi: 10.1109/CGO51591.2021.9370308.

Hugh Leather and Chris Cummins. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*, page 1–8, Kiel, Germany, 2020. IEEE. doi: 10.1109/fdl50818.2020.9232934. URL http://dx.doi.org/10.1109/FDL50818.2020.9232934.

Shengrui Li, Xueting Han, and Jing Bai. Adaptergnn: Parameter-efficient fine-tuning improves generalization in gnns. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(12):13600–13608, March 2024. ISSN 2159-5399. doi: 10.1609/aaai.v38i12.29264. URL http://dx.doi.org/10.1609/aaai.v38i12.29264.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks, 2015. URL https://arxiv.org/abs/1511.05493.

Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. The mlir transform dialect: Your compiler is more powerful than you think. In *CGO*, page 241–254, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708922. URL https://doi.org/10.1145/3696443.3708922.

Charith Mendis, Alex Renda, Saman P. Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, CA, USA, 2019a. PMLR. URL http://proceedings.mlr.press/v97/mendis19a.html.

Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, Vancouver, BC, Canada, 2019b. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf.

Antoine Monsifrot, François Bodin, and René Quiniou. *A Machine Learning Approach to Automatic Production of Compiler Heuristics*, page 41–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 9783540461487. URL http://dx.doi.org/10.1007/3-540-46148-5_5.

Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: Static profiling for binary optimization. *Proc. ACM Program. Lang.*, 5 (OOPSLA), oct 2021. doi: 10.1145/3485521. URL https://doi.org/10.1145/3485521.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010. doi: 10.1109/TKDE.2009.191.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, May 2003. ISSN 0362-1340. doi: 10.1145/780822.781141. URL https://doi.org/10.1145/780822.781141.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework, 2021. URL https://arxiv.org/abs/2101.04808.

Zheng Wang and Michael O'Boyle. Machine learning in compiler optimisation, 2018. URL https://arxiv.org/abs/1805.03441.

Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *PACT*, pages 307–318, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854313.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/ying19a.html.