THE UNIVERSITY *of* EDINBURGH

PRECISE CONTROL OF COMPILERS: A PRACTICAL APPROACH TO PRINCIPLED OPTIMIZATION

MARTIN PAUL LÜCKE

Doctor of Philosophy
School of Informatics
Institute of Computing Systems Architecture
The University of Edinburgh

– 2024 –

ABSTRACT

Production compilers remain constrained by rigid designs and inflexible optimization strategies that limit their adaptation to evolving hardware designs and application requirements. While academic research continues to produce innovative compilation techniques and intermediate representations that enable more flexibility and control over the compilation process, bridging the gap between these theoretical advances and practical implementation remains a significant challenge. The emergence of the MLIR compiler infrastructure, with its extensible dialect-based architecture, presents a unique opportunity to address two fundamental challenges in modern compiler design: how to integrate principled representations of programs and transformations into production compilers and how to provide users with meaningful control over optimization decisions.

For the *representation challenge*, we first integrate Rise, a functional pattern-based intermediate representation, as an MLIR dialect to establish the foundation for systematic rewrite-based optimizations in a production-ready compiler infrastructure. We demonstrate the practical utility of this integration by leveraging Rise's pattern-based abstractions to compile a machine learning model.

While Rise's functional foundation provides natural guarantees about program properties during transformations, these are lost when integrating with MLIR's largely unconstrained rewriting system. Transformations do not guarantee the preservation of dialect-specific invariants or even the Static Single Assignment (SSA) form beyond dynamic checks. We present an approach for statically validating that a set of MLIR rewrites maintains these critical properties.

For the *control challenge*, we introduce two distinct approaches to transformation control. The Transform dialect, now part of upstream MLIR, encodes optimization sequences as an MLIR dialect. It exposes previously hidden compiler optimizations, enabling developers to express complex transformation strategies directly in MLIR's infrastructure and moving beyond rigid pass pipelines to provide precise control over which optimizations are applied to specific parts of the program.

To provide a principled foundation for composing complex optimization strategies, we adapt Elevate, a functional language for composing transformations, to the MLIR ecosystem. We extend Elevate with MLIR-specific constructs while making MLIR's IR immutable to enable both approaches to work in unison. This adaptation enables compiler developers to express sophisticated optimization strategies simply by composing rewrite rules. We show how this approach scales to flexibly match computational structures, such as attention, in machine learning models.

By systematically addressing both representation and control challenges, we establish principled foundations for optimization while maintaining precise control, demonstrating how modern compiler infrastructures can evolve beyond rigid designs without sacrificing practical applicability.

LAY SUMMARY

Modern computer programs need to run efficiently on increasingly diverse hardware, from phones to specialized AI processors in large data centers. Software engineers write these programs as human-readable text, but each type of hardware only understands its own specific machine language. Compilers are the critical software tools that bridge this gap, automatically translating human-written code into efficient machine instructions. This translation process has become increasingly complex as the variety of hardware continues to grow.

Current compilers are too rigid in two ways. First, they understand programs only as simple, low-level instructions, missing opportunities to recognize and optimize larger computational patterns. Second, they follow a fixed set of rules when translating programs, much like following the same recipe every time, regardless of available ingredients or dietary needs. This one-size-fits-all approach means they often miss opportunities to make programs run faster or more efficiently, especially for complex calculations.

We address the rigidity by integrating a better way to understand programs into modern compilers. Our approach recognizes larger computational patterns, enabling intelligent translation of complex calculations common in scientific and machine learning applications.

Beyond improving how compilers understand programs, we also made their internal capabilities accessible to developers. This gives them direct control over powerful program improvements that existed in compilers but were previously hidden and lets them specify exactly where and how these improvements should be applied. These improvements can be combined flexibly to create translation strategies that are optimized for specific programs and hardware platforms.

We built these solutions into an existing compiler system that is already used by major technology companies. This means our research can directly improve real-world applications, particularly in areas like artificial intelligence, where program efficiency is crucial. By making compilers more flexible, our work helps programs run faster and use less energy across many different types of computer hardware.

ACKNOWLEDGMENTS

The journey of completing this doctoral thesis has been one of profound personal and academic growth, made possible by an extraordinary network of supporters, mentors, and friends.

I am deeply grateful to my supervisor, Michel Steuwer, whose exceptional mentorship extended far beyond academic guidance. What I value most is that I could always count on his support and understanding, even during periods when research progress seemed distant. His patience and encouragement have been fundamental to this journey.

I want to thank my colleagues in our research group - Bastian Köpke, Johannes Lenfers, Rongxiao Fu, Thomas Koehler, and Xueying Qin - for their collaborative spirit and intellectual discourse. I am particularly grateful to my colleagues at the University of Edinburgh for creating such a welcoming research environment. Special thanks to Mathieu Fehr, Sasha Lopoukhine, and Tobias Grosser, who made Edinburgh still feel like home during a time of transition.

My internship experiences have greatly enriched my research perspective, and I thank my mentors: Andrew Fitzgibbon while at Microsoft, Liam Fitzpatrick and Harsh Menon at AMD, and Alex Zinenko during my time at Google DeepMind. Each internship added unique perspectives that have profoundly influenced both my research and my understanding of the field.

Beyond colleagues and mentors, I am blessed with friends who have become family. Dirk, Johannes, Jonny, Julius, and Timo - you have been there through every step, providing not just support and perspective but a brotherhood that has been essential to this journey. I am equally grateful to my actual family - my parents, Michael and Cornelia, Matthias, my sister, Caren, and Jona - for their endless encouragement and unwavering belief in me throughout my life, not just during this PhD journey.

A special place in these acknowledgments belongs to Johannes (who keeps popping up in this text), who has been far more than just a colleague. From passionate discussions about our research to late-night gaming sessions with our friends, his friendship has been a cornerstone of this journey. Together with Bastian, our shared office became a place of not just work but of friendship, marked by fruitful conversations over cake, mutual

commiseration about cafeteria food, and camaraderie that made even the most challenging days brighter.

Above all, I owe an immeasurable debt of gratitude to my partner, Marie. Her unwavering support and understanding went far beyond what I could have asked for. Through the deepest valleys of this journey, her strength, patience, and unconditional love were my guiding light. She stood by me through every challenge, every doubt, and every triumph, making this achievement as much hers as it is mine.

Martin Paul Lücke
Münster, December 2024

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

This thesis is based on ideas and results that have been presented in the following publications:

[1] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Lücke, Théo Degioanni, Michel Steuwer, and Tobias Grosser. "Sidekick compilation with xDSL." In: *Accepted at the 2025 International Symposium on Code Generation and Optimization, CGO 2025*.

[2] Martin Lücke, Michel Steuwer, and Aaron Smith. "Integrating a functional pattern-based IR into MLIR." In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 12–22.

[3] Martin Lücke, Michel Steuwer, and Aaron Smith. "A functional pattern-based language in mlir." In: *2nd Workshop on Accelerated Machine Learning@ ISCA 2020*.

[4] 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: *Accepted at the 2025 International Symposium on Code Generation and Optimization, CGO 2025*.

Martin Paul Lücke

CONTENTS

1 INTRODUCTION

Advances in the field of machine learning (ML) are exposing fundamental limitations in compiler design. As ML applications push the boundaries of what we expect from compilers, they highlight challenges that extend far beyond ML-specific problems: how do we systematically represent, transform, and optimize programs for increasingly diverse computing demands and emerging hardware?

Traditional compilers, while successful for general-purpose programming, face increasing pressure to evolve. Their fixed optimization passes, and hidden heuristics create fundamental challenges: users cannot easily influence optimization decisions, fine-tune transformation strategies, or adapt the compiler to new domains or hardware. These limitations become particularly evident in domains like ML, where optimization requirements vary dramatically between applications and target hardware and where opportunities for optimization could be systematically derived but are often hidden by rigid program representations.

The emergence of MLIR (Multi-Level Intermediate Representation) as an industrial-strength compiler infrastructure creates new opportunities to address these challenges. Its extensible architecture and growing ecosystem provide a foundation for academics to reimagine how compilers represent and transform programs. However, bridging the gap between academic innovations and MLIR's practical constraints presents significant challenges.

1.1 FOUNDATIONS FOR PRINCIPLED COMPILER DESIGN IN ACADEMIC RESEARCH

Academic research has advanced our understanding of fundamental compiler design principles, particularly in program representation and transformation. For example, foundational work on algorithmic skeletons [15] first introduced systematic program structuring patterns that were later picked up by modern domain-specific compilers like Halide [79] and Lift [92], research has consistently pushed toward more principled approaches to

compilation. These research contributions have demonstrated promising approaches for evolving compilers beyond their current limitations, offering insights into more systematic and verifiable ways to represent and transform programs. However, most of these academic advances have not been integrated into practical compilers. For instance, modern production compilers like Clang [43] and GCC [88] lack systematic approaches to program transformation that have been demonstrated in academic work decades ago: where academic projects express optimizations through composable rewrite rules [101] and pattern-based program representations, production compilers continue to rely on hard-coded transformation passes with baked-in heuristics [53]. The following sections examine these differences in detail and explore how to bridge these distinct approaches.

1.1.1 *Pattern-Based Program Representation*

One key insight from academic work is that many complex computations from domains like dense linear algebra used in high-performance computing and machine learning can be systematically decomposed into fundamental patterns [15]. Consider a matrix multiplication operation: while frameworks like TensorFlow and PyTorch excel at providing intuitive high-level abstractions for machine learning development, their reliance on monolithic operators for internal representation limits optimization opportunities [6]. These operators, while convenient for frontend programming, restrict the compiler's ability to explore alternative implementations. For instance, fusing matrix multiplication with other operations requires explicit implementation of fusion patterns, even though these opportunities could be systematically derived if the operations were expressed through more basic patterns or formal algebraic principles. While algebraic approaches can derive these optimizations through formal mathematical manipulation, in practice, most current production frameworks opt instead for pattern matching on named operations, sacrificing systematic derivation for implementation simplicity [23, 38, 83].

Languages like Rise [90] and Lift [92] demonstrate the power of this approach. They represent computations using a small set of compositional patterns such as map, reduce, and zip. For example, matrix multiplication can be expressed as shown in Figure 1.1.

```
fun(A : M.K.float => fun(B : K.N.float =>
  A |> map(fun(arow =>
    B |> map(fun(bcol =>
      zip(arow, bcol) |> map(*) |> reduce(+, 0) )) ))))
```

Figure 1.1: Matrix multiplication expressed using a pattern-based representation.

This representation decomposes matrix multiplication into fundamental operations: mapping over rows and columns, zipping corresponding elements, multiplying pairs, and reducing with addition. The power of this approach lies in how it enables systematic program transformation through rewrite rules. These rules transform programs in verified ways - for example, reordering operations to improve locality or parallelizing computations - while preserving semantic meaning. Prior work has demonstrated how this combination of decomposition and principled transformation can provide both correctness guarantees and flexible optimization for different hardware targets. For instance, the Lift compiler framework has shown competitive performance across diverse hardware architectures including CPUs, GPUs, and FPGAs [29, 62, 85, 93].

1.1.2 *Principled Program Transformation*

While frameworks like TensorFlow and PyTorch rely on optimization passes that are tightly coupled to specific operators and hardware targets, academic work shows how transformations themselves can be treated as composable programming constructs. Elevate [28], for instance, represents program transformations as first-class objects that can be composed using functional programming principles. This approach enables performance engineers and domain experts to precisely control optimization strategies without modifying compiler internals. For example, tiling is a well-known optimization technique that improves locality by breaking computations into smaller blocks that better utilize the cache hierarchy. While frameworks like Halide provide tiling as a predefined compiler API, Elevate allows these experts to define and customize such transformations through composition of simple primitives:

```
def tileND(n: List[Int]): Strategy[Rise] = DFNF ; (n.size match {
  case 1 => function(split(n.head))     // loop-blocking
  case i => fmap(tileND(d-1)(n.tail)) ; // recurse
            function(split(n.head)) ;   // loop-blocking
            interchange(i) })           // loop-reorder
```

This approach shows how complex optimizations can be built from simple primitives: loop splitting, reordering, and recursion combine to create a general-purpose tiling strategy. Unlike monolithic compiler APIs that only expose predetermined transformations, this approach enables users to build their own optimization abstractions. The explicit and composable nature of these transformations enables formal reasoning about correctness while allowing complex optimizations to be built from simpler, well-understood components.

These academic approaches—whether pattern-based representations like Rise and Lift or compositional transformation systems like Elevate—demonstrate how programs can be systematically represented, transformed, and optimized using principled foundations. The challenge now lies in bringing these innovations into practical compiler infrastructure while preserving their foundational benefits.

This integration is challenging for several reasons:

- Academic projects like Rise and Lift rely on custom software stacks with fundamentally different design choices. For instance, they use purely functional transformations and dependently-typed systems to guarantee correctness features incompatible with the C++-based architecture of LLVM [44] or GCC [88].

- The systematic transformation capabilities of academic approaches like Elevate depend on immutable data structures and composition through pattern matching. This directly conflicts with production compilers' mutable, imperative nature, where transformations destructively modify the IR. For example, while Elevate can explore multiple optimization sequences by treating transformations as pure functions, LLVM's passes permanently modify the IR, making it difficult to backtrack or explore alternatives.

- Academic frameworks typically focus on optimizing isolated computational kernels. For instance, Lift excels at optimizing individual linear algebra operations but lacks integration with full program compilation pipelines like those in LLVM that handle everything from function calls to memory management. This isolation makes it difficult to consider how kernel optimizations interact with broader program properties and existing optimization passes.

1.2 THE CHALLENGES OF BRIDGING ACADEMIC INNOVATION AND PRODUCTION COMPILERS

MLIR's emergence as a production-quality compiler infrastructure presents a unique opportunity to address these challenges. Its extensible architecture and dialect ecosystem provide a foundation for integrating academic innovations with industrial compiler technology. However, realizing this potential requires addressing several fundamental challenges in both representation and transformation.

1.2.1 *The Representation Challenge*

Modern compiler design faces a fundamental challenge in how to represent both programs and their transformations in ways that enable systematic optimization while maintaining strong correctness guarantees. We identify two essential aspects:

COMPOSITIONAL PROGRAM REPRESENTATION At the heart of effective program optimization lies the question of how to represent complex computations, particularly from domains like machine learning, in a way that systematically exposes optimization opportunities. While industrial ML frameworks typically encode operators as monolithic units, these operators are inherently mathematical compositions of simpler building blocks. For example, a convolution operator in TensorFlow or PyTorch encapsulates multiple nested loops and mathematical operations that could be optimized independently if exposed. We need representations that make this compositional nature explicit and amenable to systematic transformation. This requires designing an intermediate representation that captures the high-level mathematical semantics while enabling interoperability with existing frontends and dialects.

STRUCTURAL PROPERTIES OF TRANSFORMATIONS The way we represent transformations directly impacts our ability to reason about their correctness. Academic systems demonstrate how principled representations can provide guarantees by construction through their type systems and mathematical foundations. However, practical compiler implementations frequently trade off such formal foundations against accessibility for compiler engineers and ease of maintenance. We discuss this further in Section 3.3. Thus, we need alternative mechanisms to ensure

transformation validity. This requires developing static verification techniques that can analyze transformations and enforce crucial invariants at compile time rather than discovering violations irreparable during execution.

1.2.2 *The Control Challenge*

Modern compilers contain sophisticated optimization capabilities, but their rigid design creates two fundamental control challenges: how to expose transformation capabilities at the right granularity and how to enable systematic composition of these transformations.

GRANULARITY OF CONTROL Compiler optimizations are traditionally implemented as monolithic passes with baked-in heuristics. While production compilers contain powerful optimization capabilities, they hide these behind rigid, hard-to-customize interfaces. They bundle multiple transformations into fixed passes and require users to implement entirely new passes for slight variations in optimization strategy. This rigidity becomes particularly limiting when optimizing complete programs, where different optimization needs could be met by recomposing existing transformations in new ways rather than implementing specialized passes. Access to these transformations at a finer granularity would unlock opportunities for systematic reuse, enabling more flexible and powerful optimization strategies.

COMPOSITION MECHANISMS The key is to elevate program transformations from fixed compiler capabilities to first-class programming constructs accessible to performance engineers and domain experts. While academic systems achieve this via pure functional principles that enable these specialists to freely compose and explore transformation sequences, traditional compiler frameworks apply transformations destructively, committing to optimization decisions immediately. This limits performance engineers to predetermined optimization strategies and prevents domain experts from leveraging their specialized knowledge to guide optimizations. When integrating principled transformation approaches with MLIR, we need mechanisms that preserve this ability to experiment with transformation sequences while maintaining practical efficiency.

1.3 CONTRIBUTIONS OF THIS THESIS

This thesis presents a systematic approach to addressing these challenges by demonstrating how to bring principled approaches from academic research into a production compiler infrastructure and, with them, improve the control over compiler optimizations. Our contributions directly address both challenges while maintaining their advantages: foundational guarantees combined with practical usability:

Addressing the Representation Challenge:

RISE IN MLIR - A PATTERN-BASED DIALECT: We demonstrate how to integrate a principled pattern-based intermediate representation within MLIR's framework. By implementing Rise as an MLIR dialect, we enable its use in practical compilation pipelines - demonstrated through the compilation of TensorFlow models where Rise seamlessly integrates with existing MLIR dialects. Based on a principled IR design grounded in lambda calculus, our implementation captures Rise's pattern-based semantics. We show the effectiveness of this approach through case studies, including the decomposition of a 2D convolution into simpler patterns as a compiler optimization, and demonstrate that this representation incurs no significant performance overhead.

STATIC ANALYSIS OF MLIR TRANSFORMATION PROPERTIES: We demonstrate how to verify appropriately represented MLIR transformations statically before they are applied rather than relying on runtime checks. By developing a static analysis framework for all rewrites represented in MLIR's Pattern Definition Language (PDL), we enable compile-time verification of both SSA properties and user-defined constraints. We validate our approach through systematic testing using a custom fuzzing infrastructure, which revealed actual bugs in an existing MLIR pattern. This verification framework ensures transformation correctness without restricting MLIR's flexibility, providing a foundation for safe transformation composition.

Addressing the Control Challenge:

A PRACTICAL REWRITING SYSTEM FOR MLIR: Through the introduction of the Transform dialect, we demonstrate how to encode transformations as MLIR operations and apply them to IR at different abstraction levels. We enable users to compose existing compiler optimizations that were previously hidden in internal passes and explore optimization strategies systematically. We validate our approach through five case studies, demonstrating how to build robust transformation pipelines and systematically explore optimization opportunities. This work is now part of upstream MLIR, where it has been adopted by the community as a new way to control compiler optimizations.

A PRINCIPLED REWRITING SYSTEM FOR MLIR: We demonstrate how to build a composable transformation system from first principles within MLIR's framework. Our design introduces an alternative, immutable IR structure that enables explorative optimization through backtracking and provides basic building blocks for composing transformation sequences. We validate this approach by showing how certain classes of existing MLIR transformations can be expressed as rewrites, demonstrating pattern matching capabilities for ML models, and proving that the immutable design maintains reasonable performance overhead. This demonstrates how a principled approach to transformation composition can enhance a production compiler framework without sacrificing its practical benefits.

1.4 OUTLINE OF THIS THESIS

The remainder of this thesis is structured as follows:

CHAPTER 2 introduces the fundamental concepts and technologies underlying this work. It presents MLIR's architecture and transformation framework, with particular focus on its dialect ecosystem and extension mechanisms. The chapter then introduces the academic foundations we build upon: the Rise language with its pattern-based representation and the Elevate transformation system.

CHAPTER 3 surveys compiler frameworks and their methods for program representation, transformation, and correctness. We examine systems ranging from LLVM to modern multi-level IRs,

different transformation paradigms, and verification techniques, providing context for our technical decisions in subsequent chapters.

CHAPTER 4 presents our pattern-based representation approach through the implementation of Rise as an MLIR dialect. We describe our encoding of Rise's core concepts and detail our two-phase compilation approach for integrating with industrial ML compilers.

CHAPTER 5 introduces our static analysis framework for checking the structural properties of MLIR transformations. Building on MLIR's Pattern Definition Language (PDL), we present complementary analysis mechanisms for maintaining both SSA and dialect-specific structural invariants.

CHAPTER 6 presents the Transform dialect as a solution to expose fine-grained control over compiler transformations. We demonstrate how this enables systematic exploration of optimization possibilities while maintaining verification guarantees.

CHAPTER 7 introduces our principled rewriting system that reconciles MLIR's transformation framework and Elevate's approach to strategic rewriting. We present our immutable IR design and demonstrate how it enables features like transformation rollback and strategic exploration.

CHAPTER 8 critically analyzes the contributions of this thesis. We highlight the limitations of the proposed approaches and outline promising directions for future work.

2

BACKGROUND

This chapter introduces the key systems and concepts that form the foundation for this thesis. To address the challenges outlined in the previous Chapter - specifically, the gap between industrial compiler practice and principled academic approaches - we deliberately focus on three systems: MLIR, Rise, and Elevate.

We chose MLIR because it represents the state-of-the-art in industrial compiler infrastructure. Through its extensible dialect system, MLIR offers practical solutions for handling diverse computation models. However, while MLIR provides powerful mechanisms for program transformation, it lacks formal guarantees about transformation correctness and offers limited control over optimization decisions.

To address these limitations, we selected Rise and Elevate from the academic domain. Rise's pattern-based functional design provides a principled approach to program representation, making program properties explicit and amenable to formal reasoning. Elevate complements this by offering precise control over program transformations through its strategy language, enabling systematic and reproducible optimization decisions.

While alternative approaches like polyhedral compilation offer powerful mathematical frameworks for program analysis and transformation, our work explores a different direction. Polyhedral methods excel at analyzing and optimizing affine loop nests through precise mathematical modeling of data dependencies and loop transformations, as demonstrated by their successful integration in MLIR's affine dialect. Our choice of Rise and Elevate as a starting point was motivated by their ability to express transformations through a functional programming paradigm that naturally aligns with our goal of composable program transformations and explicit reasoning about program properties. While both approaches operate within their respective restricted domains, the functional approach provides a framework that more directly supports our research goals of exploring systematic program transformation strategies. Modern machine learning approaches in compilation, particularly deep learning and large language models, have shown promising results in areas like auto-tuning and optimization selection [30,

95]. While these methods can discover effective optimizations, their decision-making process can be difficult to interpret and validate formally. In contrast, Rise and Elevate provide a foundation where transformation decisions are explicitly encoded and their effects can be reasoned about systematically.

The combination of these systems is particularly valuable as it addresses both representation and control challenges in modern compiler design. MLIR provides the industrial-strength infrastructure and practical compilation pipeline, while Rise and Elevate bring principled approaches to program representation and transformation. Understanding how these systems approach program compilation differently–and how they can complement each other–is crucial for appreciating this thesis's contribution to bridging the gap between industrial and academic compiler design.

2.1 MLIR - MULTI-LEVEL INTERMEDIATE REPRESENTATION

The increasing complexity of computing systems has fundamentally changed the requirements for compiler design. While traditional compilers focused primarily on translating high-level programming languages to machine code, modern compilers must handle diverse programming models, multiple hardware targets, and domain-specific optimizations. This evolution has exposed limitations in traditional compiler architectures, particularly in their intermediate representations (IRs).

Traditional compiler IRs were designed with a fixed abstraction level in mind. LLVM IR, for instance, provides a low-level, hardware-independent representation suitable for traditional imperative languages. However, this single level of abstraction makes it challenging to represent and optimize domain-specific constructs efficiently. High-level optimization opportunities are often lost once programs are lowered to these traditional IRs, while domain-specific knowledge crucial for optimization becomes obscured.

The MLIR [45] The Multi-Level Intermediate Representation) framework addresses these challenges using a novel architecture that supports multiple levels of abstraction within a unified infrastructure. Its design enables seamless transitions between different representation levels while maintaining the ability to perform optimizations at each level.

2.1.1 *IR Design*

At its core, MLIR's IR is built on well-known compiler foundations that enable efficient program analysis and transformation. We introduce these key concepts before discussing MLIR-specific constructs.

STATIC SINGLE ASSIGNMENT Static Single Assignment (SSA) form [81] is a program representation where each variable is assigned exactly once in the program text. Variables that would be reassigned are instead split into distinct versions, making data flow explicit. Figure 2.1 illustrates this concept:

```
1 x = 1
2 x = x + 1
3 y = x * 2
4 x = y + 3
```

(a) Program with reassignments

```
x1 = 1
x2 = x1 + 1
y1 = x2 * 2
x3 = y1 + 3
```

(b) SSA form with unique definitions

Figure 2.1: Two equivalent program representations of $(1 + 1) * 2 + 3$. The original program (a) reuses variable names, while the SSA form (b) creates distinct versions for each definition, making data flow explicit.

SSA form makes the data flow explicit: we can easily see which definition of x is used in each operation, as shown in Figure 2.1b. By making value definitions unique, it significantly simplifies compiler analysis and optimization. This property also simplifies dataflow analysis, as determining which value is used at any program point becomes trivial. Common optimizations, such as constant propagation and dead code elimination, become more straightforward to implement because the definitions and uses of each value are immediately apparent. Additionally, SSA form benefits register allocation: since each value is defined exactly once and has a limited lifetime, the compiler can more easily and effectively manage register usage, often leading to better allocation decisions.

DEF-USE CHAINS SSA form explicitly tracks def-use relationships. A def-use chain is a data structure maintained by the compiler that connects a value's definition to all its uses in the program. These chains form a directed graph where nodes are program points and edges connect definitions to their uses. The compiler explicitly maintains both forward edges (from defini-

tions to uses) and backward edges (from uses to definitions) to enable efficient traversal in both directions. For example:

```
1 x1 = 42 // definition
2 x2 = x1 + x1 // uses x1 twice
3 x3 = x2 * x1 // uses x2 and x1
```

Here, x1's def-use chain contains three use sites: two in the addition and one in the multiplication. By maintaining these chains as explicit data structures, the compiler can perform efficient queries like:

- Finding all uses of a definition without scanning the entire program
- Determining the defining operation for any use
- Updating references when transforming the program

These capabilities are essential for many compiler analyses and transformations, such as dead code elimination or constant propagation.

CONTROL FLOW GRAPHS While SSA form works straightforwardly for straight-line code, real programs contain branches and loops. A Control Flow Graph (CFG) represents these control flow relationships. In a CFG, nodes are basic blocks - sequences of instructions with a single entry point and a single exit point - and edges represent possible control flow between blocks.

Basic blocks follow three key properties:

- Control flow in a block proceeds linear through the block from top to bottom
- Each block has a single entry point at its beginning
- Each block has a single exit point at its end in form of a terminator instruction

Common terminators include unconditional branches (br), conditional branches (cond_br), and return instructions. Figure 2.2 shows two equivalent representations of the same control flow structure.

CONTROL FLOW MERGING When control flow paths merge in SSA form, we need a mechanism to select values based on which path was taken. As shown in Figure 2.2a, traditional SSA implementations use phi nodes (line 15) for this purpose.

The phi node explicitly lists all possible incoming values and selects the appropriate one based on which predecessor block was executed.

MLIR takes a different approach using block arguments, as shown in Figure 2.2b. Instead of using a special phi operation, values are passed as arguments to blocks through the control flow edges themselves. Branch instructions explicitly specify which values to pass to their target blocks, similar to how function calls pass arguments to functions.

```
entry:
  cond = compute_condition()
  cond_br cond then else

then:
  x1 = compute_x()
  br exit

else:
  x2 = compute_y()
  br exit

exit:
  // merge x1 and x2
  result = phi(x1, x2)
  return result
```

(a) SSA form using phi nodes to merge values at control flow join points.

```
entry():
  cond = compute_condition()
  cond_br cond then() else()

then():
  x1 = compute_x()
  br exit(x1)

else():
  x2 = compute_y()
  br exit(x2)

exit(result): // Exit block


  return result
```

(b) SSA form using block arguments to pass values explicitly between blocks.

Figure 2.2: Two equivalent representations of control flow merging in SSA form. The left shows the traditional approach using phi nodes, while the right shows the block argument approach used by MLIR.

Both approaches are equivalent in expressiveness, but block arguments offer several advantages. Representing control flow merges like function calls simplifies IR analysis, as the same mechanisms used for function arguments can be applied to control flow merges. A crucial practical advantage is that block arguments significantly simplify program transformations. In traditional SSA, phi nodes must always appear at the beginning of a block, requiring transformation passes to carefully maintain this invariant. With block arguments, this concern is eliminated as arguments are a property of the block itself. This allows transformations to be more local and reduces the complexity of managing control flow merging points.

DOMINANCE In a CFG, a node A dominates node B if every path from the program entry to B must pass through A. More formally, A dominates B if A appears on every path from the entry node to B.

Looking back at the CFG example in Figure 2.2, the entry block dominates all other blocks because every path must start there. However, neither the then nor else block dominates the exit block, as there are paths to it that do not include either one. The dominance property is crucial for SSA form: the definition of a value must dominate all its uses. This means that along any possible execution path, a value must be defined before it can be used. This property enables many compiler optimizations and ensures program correctness.

2.1.2 *MLIR's IR Structure*

Modern compilers must handle diverse programming models, hardware targets, and domain-specific optimizations. This requires an IR that can represent programs at multiple abstraction levels while maintaining the ability to perform optimizations at each level. MLIR addresses this challenge through an extensible operation-centric design.

MLIR's fundamental building block is the operation. Unlike traditional IRs that provide a fixed set of instructions, MLIR provides an extensible framework where new operations can be defined. The basic syntax of an MLIR operation is:

```
%result = "dialect.op"(%input) {attr=42:i32} : (!typeA) -> !typeB
```

Each operation encapsulates:

- The operation name (dialect.op) identifying the operation and its dialect namespace
- Lists of *operands* and *results* (%input and %result) with their types (!typeA and !typeB)
- Operation-specific *attributes* (attr = 42) storing constant metadata
- A number of *regions* containing nested operations (not shown in this example)
- Operation *traits* defining behavior and structural properties (not visible in syntax)

This design enables extensibility while maintaining a uniform structure for analysis. New dialects can define custom operations with their semantics, but all operations follow this consistent format. This allows MLIR to represent domain-specific abstractions while reusing common infrastructure for SSA properties, dominance analysis, and control flow.

BLOCKS AND REGIONS MLIR adopts the block argument-based CFG structure we discussed earlier, where basic blocks receive arguments through their control flow edges. Building on this foundation, MLIR introduces regions as a key organizational concept. A region contains a Control Flow Graph of basic blocks, where each block is a sequence of operations that ends with a terminator operation. Regions are attached to operations–an operation may contain zero, one, or multiple regions depending on its semantics. For example, a function operation contains a single region representing its body, a conditional operation typically contains two regions for its then and else branches, and operations like addition contain no regions at all.

Control flow in MLIR operates at two levels: within a region between blocks and across regions. While blocks within a region are connected through explicit branch operations, control flow between regions is managed by their containing operations. This two-level approach enables MLIR to represent both high-level programming constructs, such as loops and conditionals or even lambdas, as well as low-level machine code that predominantly uses branches and jumps. The same IR can thus smoothly transition from high-level abstractions to low-level implementation details through its compilation pipeline.

This structure provides powerful abstractions beyond basic control flow. For example, in accelerator offloading, regions can encapsulate code meant for different execution targets - the semantics of such regions is not immediate execution but rather scheduling computation for the accelerator. For parallel execution, regions can represent concurrent tasks or thread blocks. Some dialects use regions to express asynchronous execution, where a region's operations are scheduled for future execution while the surrounding code continues. This flexibility allows MLIR to represent complex execution models while maintaining clear semantics: the implementation details of scheduling, synchronization, and data movement are encapsulated within the region-holding operation. Let's examine these concepts with

an example from the MLIR language reference [61], shown in Figure 2.3

```
func.func @accelerator_compute(i64, i1) -> i64 {
^bb0(%a: i64, %cond: i1): // Code dominated by ^bb0 may refer to %a
  cf.cond_br %cond, ^bb1, ^bb2

^bb1:
  // This def for %value does not dominate ^bb2
  %value = "op.convert"(%a) : (i64) -> i64
  cf.br ^bb3(%a: i64)    // Branch passes %a as the argument

^bb2:
  accelerator.launch() {
    ^bb0:
      // Region of code nested under "accelerator.launch"
      // it can reference %a but not %value.
      %new_value = "accelerator.do_something"(%a) : (i64) -> ()
  }
  // %new_value cannot be referenced outside of the region

^bb3:
  ...
}
```

Figure 2.3: Example showing MLIR's hierarchical structure of regions and blocks. The function contains a region with multiple blocks forming a CFG, and the `accelerator.launch` operation contains a nested region. This demonstrates both explicit control flow through branches and control flow through nested regions

This example illustrates the hierarchical organization of blocks and regions in MLIR. The function operation contains a region implementing its body, which consists of a CFG with four blocks (^bb0 through ^bb3). The entry block ^bb0 receives the function's parameters as block arguments.

Within this function region, we see both forms of control flow: explicit branches between blocks (cf.cond_br and cf.br) and control through nested regions. The `accelerator.launch` operation contains its own region with its own blocks, showcasing how regions enable different execution semantics - in this case, code meant for accelerator execution.

The example also demonstrates MLIR's scoping rules for values:

- Values defined in a block (like %value in ^bb1) are only visible in blocks reachable through the CFG
- Values from an outer region (like %a) are visible in inner regions, allowing the accelerator code to access function parameters

- Values defined in an inner region (like %new_value) cannot be referenced outside that region, maintaining proper encapsulation of the accelerator computation

REGION ISOLATION A region's interaction with its surrounding context depends on the traits of its containing operation. In MLIR, when an operation has the *IsolatedFromAbove* trait, its regions are not allowed to access values defined outside - all values must be passed explicitly through operands. For example, in the `func.func` operation, which has this trait, nested operations can only access values through block arguments. In contrast, regions in operations without this trait, like `scf.for`, may access values from their surrounding context.

Traits like *IsolatedFromAbove* are compile-time properties that operations can declare. Other common traits include *Commutative* for operations where operand order does not matter or *Terminator* for operations that must appear at the end of a block. Traits are automatically verified and enable certain optimization opportunities.

DIALECTS Building on these fundamental IR concepts, MLIR introduces its key innovation: the dialect system. Dialects are namespaced sets of operations, types, and attributes that together form a cohesive abstraction level. Each dialect defines its own semantics and invariants while adhering to MLIR's core IR structure. For example, the `arith` dialect provides basic arithmetic operations, the `scf` dialect defines structured control flow operations, and the `func` dialect introduces function declarations and calls.

Dialects can define custom types and attributes specific to their domain. For instance, the `tensor` dialect introduces tensor types with static or dynamic shapes, while the `gpu` dialect provides attributes for specifying parallel execution configurations. These custom types and attributes allow dialects to capture domain-specific information crucial for optimization while maintaining MLIR's uniform IR structure.

This modular design allows MLIR to simultaneously represent different abstraction levels within the same IR. A single MLIR program might contain high-level machine learning operations from the `tf` dialect alongside low-level loop structures from the 'scf' dialect. This multi-level representation enables the gradual transformation of programs while preserving semantic information at each level.

Based on the paper you shared, I'll help draft a section about IRDL to add to the MLIR background chapter. Here's a proposed addition that fits with the existing content:

IR DEFINITION LANGUAGE (IRDL) While MLIR provides powerful building blocks for creating compiler IRs through dialects, types, and operations, historically these components had to be defined through verbose C++ code or generic record formats like TableGen. This made exploring and innovating in IR design costly. To address this challenge, IRDL [21] was developed as a domain-specific language for concisely defining MLIR dialects and their components.

IRDL enables developers to specify dialects through a declarative format focusing on three key aspects:

- *Structural Definition* - operations, types, and attributes are defined using a concise syntax embedded into MLIR's IR structure
- *Constraints* - invariants on types, attributes, and operations can be expressed through a constraint system
- *Verification* - constraints are automatically converted into verifiers that enforce IR validity

For cases requiring more complex verification logic beyond IRDL's declarative capabilities, IRDL-C++ allows embedding C++ code while maintaining the benefits of the declarative approach where possible. This hybrid approach enables expressing the full range of IR designs needed in practice while keeping common cases concise.

Analysis of MLIR's core dialects shows that IRDL can express the vast majority of IR components declaratively - 97% of operations can specify their local constraints purely in IRDL. By making IR definitions explicit and self-contained, IRDL significantly reduces the cost of exploring new IR designs while enabling better tooling support for compiler development.

2.1.3 *Compilation through progressive lowering*

Traditional compilers typically translate programs quickly to a single low-level intermediate representation like LLVM IR. Consider how a traditional C++ compiler handles matrix multiplication written using nested loops:

When lowered to LLVM IR, this computation becomes a sequence of low-level operations: pointer arithmetic, scalar loads and stores, and floating-point multiplications organized in nested loops. At this level, the mathematical structure of matrix multiplication is completely obscured. The compiler must perform complex analysis to recognize optimization opportunities:

- Loop dependence analysis to determine if iterations can be reordered
- Memory access pattern analysis to enable vectorization
- Alias analysis to prove memory regions do not overlap
- Pattern matching to identify idioms like reduction loops

This early lowering creates several challenges. First, even when the analysis succeeds, the compiler has limited flexibility in applying optimizations because many implementation decisions are already fixed. Second, targeting specialized hardware like GPUs or matrix acceleration units becomes difficult because their abstractions do not match the low-level representation. Third, domain-specific knowledge that could guide optimization is lost - for instance, the fact that matrix multiplication is associative and can be processed in multiple ways.

The gap between high-level program representations and executable machine code thus presents multiple interrelated challenges. A matrix multiplication operation encapsulates mathematical properties like associativity, semantic information about data layouts, and opportunities for parallel execution. Making all implementation decisions simultaneously while preserving these properties leads to complex, brittle compiler implementations. A one-step lowering must simultaneously consider loop structure, memory layout, mathematical properties, and hardware features.

MLIR addresses these challenges through *progressive lowering*, where operations from higher-level dialects are gradually transformed into operations from lower-level dialects through a series of smaller, more manageable steps. As shown in Figure 2.4, this process starts with high-level dialects like TensorFlow (tf) and progressively moves through different levels of abstraction, with each level focusing on specific implementation decisions.

At the highest level, the tf dialect represents operations like matrix multiplication as atomic operations, allowing the compiler to make high-level decisions about shape specialization

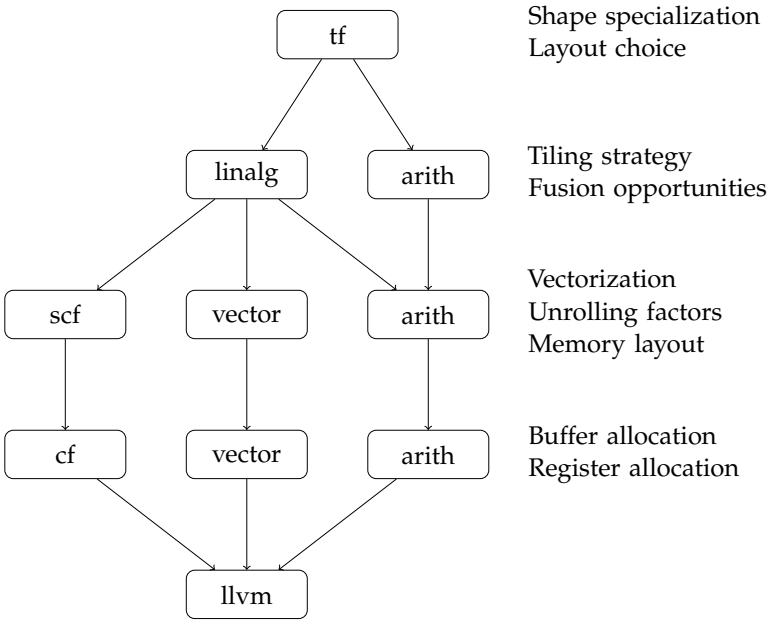


Figure 2.4: An example of the dialects used during the progressive lowering of a program through MLIR's dialect ecosystem. Each level represents a different abstraction, with decisions about implementation strategy made during lowering. Multiple dialects can coexist at the same level, enabling gradual transformation while preserving high-level information.

and data layout. The next level introduces more structured representations through dialects like linalg (Linear Algebra), where mathematical properties become explicit and enable decisions about tiling strategies and operation fusion.

As we move further down, the scf (Structured Control Flow) and vector dialects introduce explicit iteration structure and vectorization. At this level, the compiler can make informed decisions about unrolling factors and memory access patterns while still maintaining enough structure to perform meaningful optimizations. The final stages handle low-level concerns like buffer allocation and register assignment before ultimately producing LLVM IR.

Let's trace this progression through a concrete example. A matrix multiplication operation starts at the highest level in the TensorFlow dialect, represented as a single atomic operation:

```
%C = tf.MatMul(%A, %B) : (tensor<8x16xf32>, tensor<16x32xf32>) ->
    tensor<8x32xf32>
```

Following the lowering path shown in Figure 2.4, the first transformation step converts this to the Linear Algebra (linalg)

dialect. This representation maintains the mathematical structure while making the operand roles explicit:

```
%C = linalg.matmul ins(%A, %B : tensor<8x16xf32>, tensor<16x32xf32>)
                outs(%init : tensor<8x32xf32>) -> tensor<8x32xf32>
```

At this point, the compiler can make decisions about tiling and fusion strategies, as indicated in the figure. The next lowering step transforms the operation into explicit loops using the Structured Control Flow (scf) dialect, introducing opportunities for vectorization:

```
%C = scf.for %i = %c0 to %c8 step %c1 {
  scf.for %j = %c0 to %c32 step %c1 {
    %acc = vector.load %C[%i, %j] : vector<4xf32>
    scf.for %k = %c0 to %c16 step %c1 {
      %a = vector.load %A[%i, %k] : vector<4xf32>
      %b = vector.load %B[%k, %j] : vector<4xf32>
      %acc = vector.fma %a, %b, %acc : vector<4xf32>
    }
    vector.store %acc, %C[%i, %j] : vector<4xf32>
  }
}
```

Each lowering step is implemented as a transformation pass that replaces operations from one dialect with semantically equivalent operations from other dialects. As our example and Figure 2.4 show, different dialects can coexist at the same level, preserving different aspects of the program representation. Each level exposes specific optimization opportunities: tiling decisions at the linalg level where iteration spaces are explicit, vectorization when loop structure is available in scf, and memory layout optimizations once computations use explicit buffers.

Unlike the traditional approach of lowering directly to LLVM IR, this multi-level strategy preserves high-level semantic information until explicitly needed. By applying transformations at their most suitable level of abstraction, MLIR avoids the need to rediscover optimization opportunities through complex analysis. The ability to represent programs at multiple abstraction levels simultaneously provides the foundation for building sophisticated compilation pipelines that can effectively target diverse hardware platforms. The final lowering step produces operations in MLIR's LLVM dialect, providing seamless integration with LLVM's extensive optimization infrastructure.

2.1.4 *Program Transformations in MLIR*

MLIR provides several mechanisms for implementing program transformations, each with different trade-offs between expressiveness, reusability, and analyzability.

The most direct approach are compiler passes. A pass takes IR as input and produces modified IR as output, with full access to modify the program structure. While passes provide maximum flexibility, they are challenging to analyze or compose as they can perform arbitrary modifications to the IR. They are typically implemented in C++ and must be applied in a predefined order through a pass manager, making it difficult to adapt transformation strategies dynamically based on program structure.

For more targeted transformations, MLIR provides a pattern-based rewriting system. Patterns specify matching criteria and a replacement template, enabling more declarative program transformation. Patterns are applied by a rewrite driver, such as the *GreedyRewriteDriver* that applies patterns greedily based on a static benefit score until no more matches are found. While patterns are less expressive than general passes, they are easier to reason about and can be composed to achieve complex transformations.

MLIR supports two approaches to defining patterns. Patterns can be written directly in C++ using the `RewritePattern` class, specifying separate match and rewrite methods. Alternatively, patterns can be specified more concisely using *TableGen*, MLIR's domain-specific language for pattern definition. TableGen patterns are translated to C++ at compile time but provide a more declarative syntax.

Building on this pattern infrastructure, MLIR introduces the Pattern Definition Language (PDL) dialect. PDL elevates patterns to first-class operations in the IR, enabling pattern definitions to be analyzed and transformed like any other MLIR program. The key innovation of PDL is its efficient matching infrastructure: many patterns are lowered to a single matching function represented in the `PDL_interp` dialect. It provides optimized pattern matching capabilities, particularly when many patterns match for similar structures that vary only slightly [82]. However, like TableGen patterns, PDL does not support matching and rewriting regions, one of MLIR's core features. This limitation restricts PDL to simpler structural rewrites, requiring developers to fall back to C++ patterns for transformations involving regions.

These different transformation mechanisms complement each other in MLIR's ecosystem. While passes provide the flexibility needed for complex transformations involving regions and control flow, patterns significantly reduce the boilerplate code re-

quired for simple structural changes. PDL offers efficient pattern matching infrastructure but is limited to region-free transformations. PDL's ability to define patterns at runtime enables extending the compiler's transformation capabilities without rebuilding it. This variety of approaches allows developers to choose the most appropriate mechanism for each transformation task, trading off between expressiveness, implementation complexity, and runtime flexibility.

In practice, MLIR's progressive lowering infrastructure primarily relies on custom C++ passes, which provide the necessary flexibility and analysis capabilities for complex dialect conversions. TableGen patterns are mainly used for simple optimization patterns that can be expressed as direct replacements without requiring sophisticated analysis. While PDL promised performance benefits for pattern matching, real-world use cases where these benefits materialize have been limited, leading most developers to either use TableGen for simple patterns or implement more complex transformations involving regions directly in C++.

2.1.5 *Interfaces to group similar operations*

While traits are static properties declared by operations, interfaces provide a more flexible mechanism for operation behavior. Interfaces solve a fundamental problem: how can transformations work across different dialects without knowing their specific implementation details? For example, a loop tiling implementation should be reusable for any dialect's loop operation, whether `scf.for`, `affine.for`, or a custom loop operation from a domain-specific dialect.

MLIR supports several types of interfaces. Operation interfaces specify methods that operations must implement, like *LoopLikeOpInterface*, which requires operations to expose their loop bounds and step values. This enables optimizations like loop tiling to work across different loop representations. Dialect interfaces enable dialect-level functionality, such as the *InliningInterface*, which allows dialects to specify how their operations participate in function inlining by defining legal inlining rules and handling symbol resolution. Type interfaces provide methods for working with custom types consistently across dialects.

For example, any operation that represents a loop can implement the LoopLikeOpInterface:

```
%result = custom.for %i = %lb to %ub step %step
    iter_args(%acc = %init) -> (i32) {
  // loop body
  custom.yield %val : i32
}
```

Here, `custom.for` implements the LoopLikeOpInterface. A tiling transformation can query the loop bounds and step values through the interface to split the iteration space into smaller blocks, regardless of whether it's working with scf.for, affine.for, or any other dialect's loop operation.

MLIR's extensible architecture, with its dialect system, progressive lowering approach, and transformation infrastructure, provides a powerful foundation for modern compiler development. Its ability to represent programs at multiple levels of abstraction while maintaining a uniform IR structure enables systematic program transformation. Through interfaces, MLIR allows these transformations to work across different dialects without sacrificing their specific semantics. However, as we will see in the following chapters, bringing principled approaches from academic compiler research into this infrastructure presents both opportunities and challenges that need to be carefully addressed.

2.2 RISE: A FUNCTIONAL DATA-PARALLEL LANGUAGE

RISE [90] is a functional programming language designed specifically for expressing data-parallel computations over multi-dimensional arrays. It is implemented in Scala and builds upon the foundations established by Lift [92], focusing on generating high-performance parallel code for multiple hardware targets, including CPUs via OpenMP and GPUs through OpenCL.

At its core, RISE employs a pattern-based programming model in which computations are expressed through composable, high-level data-parallel patterns. These patterns clearly separate what is computed from how it is computed, enabling systematic program optimization through rewriting. The language enforces this separation through a sophisticated type system that distinguishes between data types and function types. This system prevents functions from being stored in memory while supporting dependent types for symbolic array length specifications.

2.2.1 *Core language design*

RISE is built upon a formal foundation of lambda calculus, extended with specific constructs for array programming. The language's expression syntax follows traditional λ-calculus constructs, including abstraction (written as fun(x, e)), application (using parentheses), identifiers, and literals. This foundation ensures strong theoretical properties while providing a familiar basis for functional programming.

The RISE type system comprises several fundamental data types. At its base are scalar types such as integers (int) and floating-point numbers (float). These basic types can be composed into pairs, enabling the construction of compound data structures. The central feature of RISE's type system is its treatment of arrays, which are multi-dimensional through nesting and carry their size information within their type. For instance, an array type might be expressed as `array(n, float)`, denoting a one-dimensional array of `n` floating-point numbers, or `array(n, array(m, float))`, representing an `n×m` two-dimensional array through explicit nesting. This size-aware type system leverages dependent types, allowing array dimensions to be expressed using arithmetic expressions involving natural numbers. For example, when splitting an array of size `n` into chunks of size `m`, the resulting type would be `array(n/m, array(m, T))`, where `T` is the element type. This rich type information enables static verification of array operations and supports the compiler in generating efficient code with proper memory access patterns.

2.2.2 *Pattern-Based Programming Model*

RISE organizes computations around well-defined patterns that operate on multidimensional arrays. These patterns can be categorized into algorithmic patterns, data layout patterns, and array access patterns, each serving a distinct role in program expression and optimization.

Algorithmic Patterns:

- map applies a function f to each element of an array independently:
  $map(f, [x_1, x_2, ..., x_n]) = [f(x_1), f(x_2), ..., f(x_n)]$
- reduce combines array elements using an associative binary function f and initial value z:
  $reduce(z, f, [x_1, x_2, ..., x_n]) = f(...f(f(z, x_1), x_2)..., x_n)$

Data Layout Patterns:

- split divides a 1D array into a 2D array of specified shape: $split(m)([x_1, x_2, ..., x_n])$ creates an array of arrays of length m
- join flattens a 2D array into a one-dimensional array
- transpose reshapes an n×m array into an m×n array
- zip combines two arrays into an array of pairs: $zip([x_1, x_2, ..., x_n], [y_1, ..., y_n]) = [(x_1, y_1), ..., (x_n, y_n)]$

All patterns are composable, allowing complex computations to be expressed through pattern compositions. The type system ensures that pattern composition is well-formed, with array dimensions and types matching at each composition point.

Each pattern preserves high-level semantic properties that enable systematic program optimization through rewriting. For instance, map operations can be freely reordered or fused under certain conditions, and reductions can be reassociated when their combining function is associative.

This pattern-based approach provides a clear separation between what is computed (expressed through high-level patterns) and how it is computed (specified through low-level implementations).

The composability of these patterns enables the expression of complex computations in a clear, structured manner. Consider matrix multiplication, a fundamental operation in many domains. In Rise, matrix multiplication of two matrices A and B can be expressed as:

```
1 fun(A:array(M,array(K, float)) => fun(B:array(K,array(N, float)) =>
2   A |> map(fun(arow =>
3     B |> map(fun(bcol =>
4       zip(arow, bcol) |> map(*) |> reduce(+, 0) )) ))))
```

This representation clearly expresses the algorithmic structure: For each row of A and column of B, we zip the corresponding elements, multiply them pointwise, and reduce the results with addition. The type system ensures that compositions are valid, verifying that array dimensions and types match throughout. Here, it ensures that the inner dimensions K match for valid matrix multiplication.

Each pattern preserves high-level semantic properties that enable systematic program optimization through rewriting. For instance, map operations can be freely reordered or fused under

certain conditions, and reductions can be reassociated when their combining function is associative (as with addition in this example).

2.2.3 *Compilation Approach*

The Rise compiler, Shine, follows a language-oriented design that clearly separates optimization from code generation through two distinct intermediate languages, as illustrated in Figure 2.5. This design ensures that all implementation decisions are explicit before code generation begins.

The compilation process starts with a high-level RISE program expressing what to compute. This program is systematically transformed through rewriting into a low-level RISE program that explicitly encodes implementation choices such as:

- How to parallelize computations
- Where to store intermediate results
- Which optimizations to apply

These transformations are controlled by optimization strategies written in the Elevate strategy language, which allows experts to precisely specify which optimizations should be applied and in what order. We will examine Elevate closer in Section 2.3.

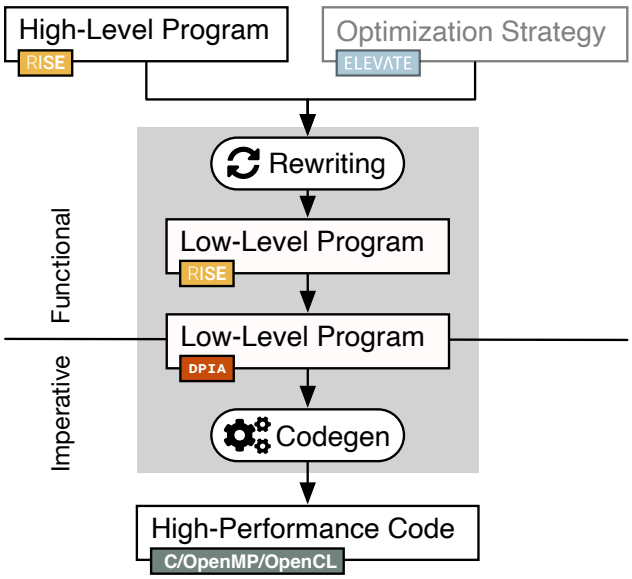


Figure 2.5: A Rise program is successively lowered and optimized in the Shine Compiler [90]

Once all implementation choices are encoded, the low-level RISE program is translated into DPIA (Data Parallel Idealized ALGOL), a hybrid functional-imperative language that serves as the final intermediate representation before code generation. DPIA's type system ensures that all necessary implementation decisions have been made explicit in the previous phase.

The final code generation step translates the imperative DPIA program into the target language (OpenMP or OpenCL). This translation is straightforward and deterministic because DPIA's imperative constructs directly correspond to the target language constructs, and all implementation decisions have already been encoded in the program. For example, DPIA's parForWorkGroup and parForLocal constructs map directly to OpenCL's work-group and local parallel loops.

This clear separation of optimization and code generation provides significant benefits for both compiler implementation and program development. The explicit nature of optimization decisions makes the compilation process more predictable and easier to reason about. Programmers can explore different optimization strategies without modifying the underlying code generator, enabling rapid experimentation with various performance tradeoffs. Furthermore, this approach facilitates verification, as the correctness of transformations can be checked independently of the final code generation step.

The effectiveness of this approach has been demonstrated in practice. Steuwer et al. [89] showed that RISE's pattern-based representation, combined with systematic rewrite rules, enables powerful program optimization. By applying algebraic identities and other transformations through rewrite rules, their automatic search process successfully optimized computations across various domains, including matrix multiplication and convolution operations. Their implementation achieved performance comparable to hand-optimized libraries such as clBLAS, validating the practical viability of this principled approach to program optimization.

Hagedorn et al. [29] demonstrate that this approach is easily extensible to new domains, in particular, the domain of stencil computations. They introduce a new pattern and corresponding rewrite rules and achieve performance comparable to hand-optimized stencils and outperform automatically generated stencils.

Mathematical operations, ranging from simple arithmetic to more elaborate computations like matrix multiplication, can

often be decomposed in multiple mathematically equivalent ways. RISE's pattern-based approach captures these different decompositions through its compositional nature. This flexibility is particularly valuable for recent GPUs, which contain specialized hardware components called tensor cores to perform matrix multiplications of a fixed size of 16x16. Similarly, recent AMD GPUs provide hardware components and instructions to perform outer product operations efficiently [1], requiring yet another approach to decomposing matrix multiplication operations. This hardware diversity highlights the importance of flexible compiler representations, which should allow complex operations to be expressed in ways that can be effectively mapped to different hardware architectures. RISE's pattern-based representation exemplifies this flexibility, allowing operations to be expressed as compositions that can be efficiently adapted and mapped to diverse hardware targets.

Siefke et al. [87] extend Rise with support for NVIDIA tensor cores without changes to the frontend language. They show that the pattern-based approach is flexible enough to target specialized hardware components using custom rewrite rules and backend abstractions.

2.3 ELEVATE: A STRATEGY LANGUAGE FOR PROGRAM OPTIMIZATION

Following RISE's strategy-preserving compilation approach, ELEVATE provides a formal language for expressing program optimization strategies. While RISE defines what can be computed through its patterns, ELEVATE controls how these patterns are transformed during optimization. This separation enables precise control over the optimization process through composable program transformations.

2.3.1 *Relationship to RISE*

The combination of RISE and ELEVATE creates a powerful synergy for high-performance code generation. RISE's pattern-based approach makes computational structures explicit - maps, reduces, and data movement patterns clearly express what is being computed. However, there are often many valid ways to implement these patterns, each with different performance characteristics. This is where ELEVATE comes in: it provides

a principled way to explore and express these implementation choices.

For example, when RISE expresses matrix multiplication as nested maps and reduces, ELEVATE enables describing how to transform this high-level representation into efficient implementations. Through strategies, we can specify exactly how to tile the computation for better cache usage, which loops to vectorize, or when to store intermediate results. The ability to compose ELEVATE strategies mirrors RISE's compositional nature - just as complex computations are built from simple patterns in RISE, sophisticated optimizations are built from simple transformations in ELEVATE.

This complementary design creates a system where optimization decisions are both flexible and precise: RISE makes program structure explicit through patterns, while ELEVATE makes transformation choices explicit through strategies.

2.3.2 *Strategies*

At its heart, ELEVATE models program transformations as strategies with the type:

```
type Strategy[P] = P => RewriteResult[P]
```

where `P` represents the type of the program being transformed (such as a RISE program), and `RewriteResult` is a type that represents either the successfully transformed program or the failure of the transformation. This foundation ensures that transformations are type-safe and their composition is well-defined. The `RewriteResult` type allows for systematic handling of transformation outcomes, including potential errors or failures.

2.3.3 *Strategy Composition*

ELEVATE provides a set of fundamental strategy combinators that enable the composition of complex optimization strategies from simpler ones:
Sequential composition (`;`): applies strategies in sequence.
Choice composition (`<+`): tries alternative strategies.
Try: attempts a strategy but falls back to identity if it fails.
Repeat: applies a strategy repeatedly until it can no longer be applied.
These combinators allow optimization strategies to be expressed

as clear, compositional specifications rather than hard-coded compiler passes.

2.3.4 *Rewrite Rules as Strategies*

ELEVATE treats rewrite rules and strategies uniformly - both are functions that transform programs. Each rewrite rule is implemented as a strategy that pattern matches on program structures and produces transformed versions. For example, a strategy might implement the fusion of two consecutive map operations in RISE as shown in Listing 2.1.

```
1 mapFusion: Strategy[Rise] = p => p match {
2   case app(app(map, f), app(app(map, g), xs)) =>
3     Success(map(fun(x, f(g(x))))(xs))
4   case _ =>
5     Failure(mapFusion)
6 }
```

Listing 2.1: Example of the map fusion rewrite rule implemented as a strategy in ELEVATE

These basic rewrite rules serve as building blocks for more complex optimization strategies. By composing rules using ELEVATE's combinators, developers can express sophisticated program transformations that would be difficult to implement as monolithic compiler passes.

2.3.5 *Traversal Strategies*

Program transformations often need to operate on specific parts of a program's structure. ELEVATE provides traversal strategies that control where and how transformations are applied within a program tree. These traversal operators act as strategy transformers: they take a strategy as input and produce a new strategy that applies the original one at specific locations in the program:

```
type Traversal[P] = Strategy[P] => Strategy[P]
```

Basic traversal operators include:

- topDown: applies a strategy while traversing the program tree from root to leaves
- bottomUp: applies a strategy while traversing from leaves to root

These traversal operators can be combined with the basic strategy combinators to create sophisticated transformation patterns. For example, a strategy might use bottomUp to ensure that innermost expressions are transformed first or topDown to handle outermost patterns before their subexpressions.

The traversal operators become particularly powerful when combined with ELEVATE's strategy combinators. For example, a bottom-up traversal combined with a try combinator ensures that a transformation is attempted at every node, working from the leaves upward:

```
bottomUp(try(mapFusion))
```

Such compositions are essential for implementing complex program transformations. The systematic application of transformations throughout a program becomes controllable and predictable, while the order of transformations can be precisely specified. This capability allows transformations to be applied to specific program fragments when needed while also preventing issues like infinite recursion in recursive transformations.

Advanced traversal patterns can be defined to handle specific program structures. When dealing with nested patterns in RISE, specialized traversals might target array dimensions, computational patterns, or data layout transformations. These specialized traversals work in concert with memory access patterns to ensure optimal program transformation.

The combination of traversal strategies with RISE's explicit optimization approach ensures that transformations are applied systematically and predictably while maintaining control over where and how optimizations take effect. This systematic approach to program transformation provides a powerful framework for expressing complex optimization strategies.

2.3.6 *Strategy Predicates*

While traversals define how to navigate program structure, strategy predicates help identify specific locations where transformations should be applied. A strategy predicate checks if a program matches certain criteria, succeeding without modifying the program if there is a match. For example, to identify specific RISE primitives:

```
def isMap: Strategy[Rise] = p => p match {
  case map => Success(p)
  case _   => Failure(isMap)}

def isReduce: Strategy[Rise] = p => p match {
  case reduce => Success(p)
  case _ => Failure(isReduce)}
```

These predicates can be composed with traversals to describe precise locations in the program. For instance, `topDown(isMap)` finds the first `map` operation in a top-down traversal. The `@` operator provides a convenient way to combine a strategy with a location description:

```
def '@'[P](s: Strategy[P], t: Traversal[P]) = t(s)
```

This enables a clear separation between what transformation to apply and where to apply it. For example: `vectorize(32) ' @' innermost(isMap)` applies vectorization to the innermost map operation.

More complex predicates can be built by combining simpler ones. For instance, to identify a map operation that has been applied to two arguments:

```
def isAppliedTwice(s:Strategy[Rise]):Strategy[Rise]=isApp(isApp(s))
```

This compositional approach to describing program locations provides a powerful and flexible way to target specific parts of a program for optimization while maintaining the clarity and reusability of transformation strategies.

2.3.7 *Comparison with Traditional Scheduling Languages*

Where traditional scheduling languages like those in Halide and TVM provide fixed sets of predefined optimization primitives, ELEVATE takes a fundamentally different approach by expressing optimizations as composable strategies. Traditional schedulers force programmers to work with black-box optimization primitives - for example, Halide's `tile` primitive is provided as a monolithic operation that cannot be customized or decomposed. In contrast, ELEVATE and similar principled approaches enable expressing such optimizations through the composition of simpler, well-understood transformations.

This difference becomes particularly apparent when implementing complex optimizations. For instance, while TVM implements tiling as a built-in primitive combining loop-blocking and loop-interchange, in ELEVATE, the same optimization is expressed as a composition of five basic rewrite rules. This compositional nature provides several advantages:

- **Extensibility:** Rather than being limited to predefined primitives, programmers can build their own optimization abstractions from simple building blocks
- **Transparency:** The implementation of each transformation is explicit and can be inspected, understood and modified
- **Correctness:** Each basic transformation can be proved correct independently, and their composition preserves correctness guarantees
- **Flexibility:** Optimizations can be customized and adapted by modifying their constituent transformations

Traditional scheduling languages also typically couple optimization decisions to specific program identifiers. For example, in TVM one might write `s[C].tile(C.op.axis[0], 32)` to tile a specific loop. This approach makes it difficult to reuse optimization strategies across different programs. ELEVATE instead uses traversal strategies to describe where optimizations should be applied, enabling the same strategy to be reused across different programs with similar structure.

Furthermore, scheduling languages often include implicit default behaviors that can lead to unexpected performance characteristics. ELEVATE takes an explicitly strategy-preserving approach where all optimization decisions must be expressed as strategies. While this requires more verbose specifications, it provides complete control over the optimization process and makes the performance impact of each decision clear.

2.3.8 *Benefits of Strategy-Based Optimization*

ELEVATE's strategy-based approach to program optimization offers several key advantages over traditional compiler optimization techniques. Where traditional compilers often rely on fixed optimization passes with limited configurability, ELEVATE enables explicit and reproducible optimization decisions. This approach is particularly valuable when targeting different hardware architectures, as optimization strategies can be tailored to specific hardware features without modifying the underlying compiler infrastructure.

The combination of RISE's pattern-based programs and ELEVATE's transformation strategies creates a powerful framework for performance portability. Domain experts can encode their

optimization knowledge as reusable strategies, making it easier to maintain and evolve optimization approaches as hardware architectures evolve. This explicit control over the optimization process also simplifies debugging and performance tuning, as each transformation step can be traced and understood in isolation.

This strategy-based approach represents a significant departure from both fixed optimization passes and purely automatic search-based methods. By making optimization decisions explicit and composable, ELEVATE provides a principled foundation for expressing complex program transformations while maintaining the flexibility needed for high-performance code generation across diverse hardware targets.

Hagedorn et al. [28] demonstrate how this approach enables the expression of complex optimizations like loop tiling through the composition of only five basic rewrite rules while achieving performance competitive with state-of-the-art DSL compilers like Halide and TVM. Their evaluation shows that despite applying over 60,000 individual rewrite steps for complex optimizations, the strategy-based approach maintains practical compilation times of under two seconds, validating the scalability of composable program transformations.

2.4 SUMMARY

This chapter has presented the key technologies and concepts that underpin this thesis. We have seen how MLIR addresses the challenges of modern compiler design through its extensible multi-level IR, while Rise and Elevate demonstrate how functional programming principles can bring precision and composability to program optimization. The contrast between these approaches - MLIR's practical but sometimes ad-hoc transformation framework versus Rise and Elevate's principled but less production-ready approach - highlights the opportunity this thesis addresses.

In the following chapters, we will show how to bring these worlds together, combining MLIR's industrial strength with the principled approaches of Rise and Elevate. This synthesis enables both systematic program transformation and practical high-performance code generation, addressing the challenges outlined in the introduction.

3

RELATED WORK

The previous chapter introduced MLIR, Rise, and Elevate as the foundational technologies for our work. These systems represent specific solutions to fundamental challenges in modern compiler design: representing programs across multiple abstraction levels, systematically transforming them while preserving semantics, and showing that they retain structural guarantees. This chapter examines how different compiler frameworks have approached these challenges, providing context for our design decisions and highlighting the novel aspects of our approach.

We begin by exploring the evolution of compiler intermediate representations (Section 3.1). Starting with LLVM's influential but single-level design, we trace the development of multi-level approaches through SUIF, Thorin, and MimIR to modern frameworks like MLIR and domain-specific solutions like Triton. This evolution reveals a fundamental tension between extensibility, semantic preservation, and implementation complexity.

Section 3.2 examines different paradigms for expressing and implementing program transformations. We analyze approaches ranging from Halide's scheduling language to polyhedral frameworks, equality saturation techniques, and extensible transformation systems like Exo. These systems demonstrate various trade-offs between transformation expressiveness, composability, and automation. Understanding these trade-offs informed our development of a transformation framework that combines the flexibility of strategy-based approaches with the rigor of formal methods.

Finally, Section 3.3 investigates how different systems approach correctness guarantees in compiler implementations. We examine approaches ranging from full compiler verification to translation validation and domain-specific verification tools, highlighting the trade-offs between comprehensive guarantees and practical applicability in production compilers.

We identify key design patterns, trade-offs, and lessons learned from existing compiler research by examining these alternative approaches. This analysis of the state of the art informs our technical decisions and highlights opportunities for innovation, particularly in combining principled academic techniques with

production compiler infrastructure. Understanding how different systems have addressed fundamental compiler challenges provides valuable insights that guide our technical solutions.

3.1 COMPILER SYSTEMS

Modern compiler systems must balance competing concerns: they must support diverse programming models, enable effective optimizations across different abstraction levels, and remain maintainable and extensible. This section examines how different compiler architectures have approached these challenges, focusing on three key areas: general-purpose compiler frameworks that provide foundational infrastructure, intermediate representations designed to expose optimization opportunities in a principled way through functional concepts, and domain-specific compilers that optimize for particular application domains.

We examine general-purpose compiler frameworks, from LLVM's influential single-level design to modern multi-level approaches like MLIR. We then explore intermediate representations specifically designed for functional programming, offering insights relevant to our work with Rise. Finally, we investigate domain-specific compiler systems, demonstrating how specialized representations and optimizations can achieve superior performance for specific domains.

LLVM [44] As a milestone in compiler infrastructure design, LLVM IR found success by providing a language-independent and target-independent abstraction that bridges high-level programming languages and machine code. Operating at a fixed abstraction level slightly higher than assembly, LLVM IR combines SSA form with a simple type system to enable powerful optimizations. However, LLVM IR's single-level design creates fundamental challenges when compiling modern applications: high-level optimization opportunities are often lost during lowering, and domain-specific information crucial for optimization becomes obscured. These limitations become particularly evident in domains like machine learning, where high-level semantic details is vital for effective optimization. Despite these limitations, LLVM remains the backbone of many modern compilers that operate at higher abstraction levels. Its mature optimization pipeline and comprehensive target support make it an ideal final lowering target - compiler frameworks like MLIR leverage

LLVM's capabilities by generating LLVM IR as their final representation, combining high-level optimizations in their own IR with LLVM's sophisticated low-level optimizations.

SUIF COMPILER [104] The Stanford University Intermediate Format (SUIF) pioneered many concepts that modern multi-level IRs have rediscovered. Initially developed in 1994, SUIF demonstrated how to represent programs at multiple abstraction levels within a single compiler framework. The system used a hierarchical IR structure where high-level constructs (like loops and arrays) coexisted with low-level details, while a flexible annotation system allowed for preserving analysis results and user-defined information throughout the compilation pipeline. This design enabled SUIF to maintain semantic information while supporting aggressive optimizations, particularly for parallel architectures. While its reliance on a fixed core IR limited flexibility compared to modern approaches like MLIR's dialect system, SUIF's successful demonstration of multi-level program representation significantly influenced subsequent compiler designs.

XDSL [22] As a modern companion framework to MLIR, xDSL demonstrates how to make multi-level compiler development more accessible through a Python-native environment. Unlike traditional MLIR implementations that require extensive C++ development, xDSL enables rapid prototyping of dialects and transformations while maintaining MLIR's core concepts like SSA form and regions. The framework's bi-directional integration with MLIR through textual interchange formats and IRDL [21] allows it to leverage MLIR's dialect ecosystem while providing a more agile development environment. While xDSL trades runtime performance for development speed, its success in both teaching and research contexts demonstrates the value of complementing production compiler frameworks with tools optimized for experimentation and prototyping.

3.1.1 *Principled Program Representation*

Several compiler systems have explored principled approaches to program representation, particularly through functional programming concepts and formal semantics, in ways that systematically expose optimization opportunities. These approaches are particularly relevant to our work, as they demonstrate dif-

ferent solutions to the fundamental challenge of maintaining high-level semantic information and enabling powerful program transformations.

THORIN [47] As an early attempt to bridge imperative and functional programming paradigms in compiler design, this graph-based higher-order intermediate representation introduced several novel concepts. Unlike LLVM IR's fixed abstraction level, Thorin supported multiple levels of abstraction via its dependency graph representation and continuation-passing style control flow. This design enabled native representation of both high-level constructs like higher-order functions and low-level features, eliminating the need for complex lowering of functional features in the frontend as required by LLVM IR.

The key contribution of Thorin was lambda mangling, a transformation that combined partial inlining and outlining to eliminate higher-order function overhead. This transformation subsumed several classic optimizations like tail-recursion elimination and loop unrolling, demonstrating that functional abstractions could be efficiently compiled away. Through its implementation in the AnyDSL [46] framework, Thorin showed that high-level functional programming constructs could achieve performance comparable to hand-written C code while maintaining the benefits of abstraction.

MIMIR [48] Building on Thorin's insights, this successor project reimagines the concepts with a more robust theoretical foundation based on the Calculus of Constructions and pure type systems. While maintaining a graph-based representation, MimIR supports both direct style and continuation-passing style and replaces Thorin's hard-coded built-ins with an extensible plugin system. Like MLIR, MimIR provides extensibility but distinguishes itself through a unified type system that supports polymorphism and dependent types, enabling type-safe composition of domain-specific abstractions. This allows MimIR to preserve and optimize high-level semantic information while maintaining the ability to lower to LLVM IR for final code generation.

A key innovation in MimIR is its normalization framework that interacts with type checking and partial evaluation. This enables novel approaches to dependent type checking and optimization while keeping the implementation significantly simpler than traditional compiler frameworks. Through its plugin ar-

chitecture, MimIR allows compiler developers to define custom operations and transformations that operate at any abstraction level, from high-level domain-specific optimizations down to low-level code generation. This multi-level capability, combined with its formal type system, provides a powerful foundation for developing domain-specific compilers that can maintain high-level semantics throughout the compilation pipeline.

3.1.2 *Domain-Specific Compilers*

The complexity of modern hardware and the increasing diversity of application domains have driven the development of specialized compiler systems. These systems demonstrate how domain-specific abstractions and optimization strategies can achieve superior performance compared to general-purpose approaches, particularly in domains like high-performance computing and machine learning.

Domain-specific compilers typically introduce specialized abstractions that directly capture domain concepts. For example, Triton [97] represents tensor computations through statically shaped multi-dimensional sub-arrays (tiles), while TACO [38] uses sparse tensor algebras as its primary abstraction. These domain-specific abstractions enable the compiler to reason about and optimize operations at a higher level than traditional scalar-oriented IRs.

Many of these systems leverage existing compiler infrastructure while adding domain-specific layers. Triton builds on MLIR for progressive lowering from tile operations to GPU code, while TVM [12] generates specialized implementations of tensor operations through its own optimization framework before targeting LLVM. This demonstrates how domain-specific abstractions can be effectively integrated into existing compilation frameworks while maintaining semantic information crucial for optimization.

The success of these approaches in achieving performance comparable to hand-optimized vendor libraries while maintaining programming flexibility has established domain specialization as a key strategy in modern compiler design. Systems like TVM and Halide (discussed in detail in Section 3.2) have further shown how domain-specific knowledge can be leveraged to separate algorithmic specifications from hardware-specific implementation decisions, enabling portable performance across different architectures.

3.1.3 *Conclusion*

Several key patterns emerge from examining these compiler systems. First, there's a clear evolution from single-level to multi-level representations, driven by the need to preserve high-level semantic information while supporting low-level optimizations. This evolution is visible across general-purpose frameworks like MLIR, principled approaches like MimIR, and domain-specific systems. Second, extensibility mechanisms have become increasingly sophisticated, from SUIF's annotation system through MLIR's dialects to MimIR's type-based approach, enabling better management of domain-specific abstractions. Third, domain specialization has emerged as a powerful strategy for managing complexity, whether through dedicated domain-specific compilers or extensible frameworks supporting domain-specific abstractions.

These systems reveal a crucial lesson: effective modern compilers must balance four competing concerns - preservation of semantic information, practical extensibility, implementation complexity, and domain-specific optimization capabilities. MLIR's dialect system represents a particularly attractive solution to this challenge, allowing domain-specific abstractions and optimizations to be cleanly integrated into a general-purpose compilation framework. The continued relevance of LLVM IR as a lowering target demonstrates that this balance can be achieved through careful separation of concerns, with different systems handling different parts of the compilation pipeline.

3.2 COMPILER TRANSFORMATIONS

While the previous section explored different approaches to program representation, how these representations can be transformed and optimized is equally essential. This section examines various frameworks for expressing and implementing program transformations, from scheduling languages that separate algorithms from their implementation strategies to rewriting systems that enable systematic transformation composition to approaches based on formal methods that provide strong correctness guarantees.

3.2.1 *Scheduling Languages*

The introduction of scheduling languages marked a significant shift in how compilers express and implement program transformations. By separating the specification of what to compute from how to compute it, these systems enable the exploration of different implementation strategies without modifying the original algorithm. This separation has proven particularly valuable in domains like image processing and machine learning, where the same algorithmic patterns often require different optimizations across various hardware targets.

HALIDE [76, 78–80] Halide revolutionized domain-specific compiler design by introducing a fundamental separation between algorithmic specification and optimization decisions. Users write pure functional descriptions of their algorithms, then separately apply scheduling primitives like split, tile, and vectorize to specify implementation details. This separation enables rapid exploration of different optimization strategies without modifying the algorithm's specification, treating schedules as first-class programming constructs.

However, Halide's approach has limitations. Schedules are applied eagerly and destructively, making it difficult to experiment with different optimization sequences or roll back unsuccessful transformations. Additionally, scheduling primitives are provided as a fixed set of built-in operations, limiting users' ability to compose or extend the scheduling language.

The success of Halide's separation principle influenced many subsequent systems, including Elevate, which builds upon this idea by representing transformations as composable, first-class entities.

TVM [12] TVM introduced a comprehensive approach to optimizing deep learning workloads across diverse hardware platforms. Unlike previous frameworks that relied on vendor-specific operator libraries, TVM provides an end-to-end compilation stack that automatically generates optimized code for different hardware targets, including CPUs, GPUs, and specialized accelerators. The system separates computation descriptions from schedules, extending Halide's compute/schedule separation with primitives for deep learning-specific optimizations like tensorization. Like Halide, it initially relied on manual scheduling through a domain-specific language for specifying

program transformations. Subsequent work extended TVM with auto-tuning capabilities [108] to automatically explore the optimization space and discover efficient schedules for specific hardware targets.

However, like Halide, TVM's transformation system remains constrained by a fixed set of scheduling primitives. While effective for standard deep learning operations, this limits the framework's ability to express novel optimization strategies or compose existing transformations in unexpected ways. These limitations have motivated subsequent work on more flexible transformation systems.

TACO [38] The Tensor Algebra Compiler (TACO) extends scheduling concepts to sparse tensor computations by introducing a format abstraction language that separates computation specification from data layout decisions. Where dense tensor compilers like Halide and TVM focus on loop transformations, TACO's key innovation is its format algebra that allows users to specify tensor storage formats using format primitives. By treating data layout as a first-class scheduling decision, TACO generates format-specific code paths that handle complex tasks like the merging of sparse tensor indices. While achieving performance comparable to hand-optimized sparse BLAS libraries, TACO shares the limitation of other scheduling systems in relying on built-in primitives rather than composable transformations.

FIREIRON [27] As a scheduling language designed explicitly for GPU optimization, Fireiron introduced the concept of treating data movements as first-class citizens in compiler frameworks. Unlike previous scheduling languages that implicitly handled data movement through computation schedules, Fireiron provides explicit abstractions for specifying computations and data movements through decomposable specifications. This approach enables precise control over how data flows through the GPU memory hierarchy, which is crucial for optimizing tensor operations on specialized hardware like NVIDIA's Tensor Cores. The system demonstrated that making data movement explicit could achieve performance improvements over vendor-optimized libraries while requiring significantly less code than hand-tuned implementations. Fireiron's approach highlighted a fundamental limitation in existing scheduling languages: by treating data movement as a second-class concern, they miss

crucial optimization opportunities that arise from coordinating computation and data movement strategies.

EXOCOMPILATION Exo [35] and its successor Exo 2 [36] introduce exocompilation - an approach that externalizes accelerator-specific code generation and optimization policies to the user level through a plugin architecture. Unlike MLIR's dialect system, Exo allows custom hardware instructions, specialized memories, and accelerator configuration state to be defined in user libraries without modifying the core compiler. This enables hardware vendors to support new accelerators without maintaining compiler forks or exposing proprietary details.

While building on user scheduling concepts from Halide, Exo advances the approach by implementing scheduling operators as composable program rewrites rather than monolithic transformations. Exo 2 significantly improves upon the original Exo by enabling users to define new scheduling operations by composing primitive transforms, allowing scheduling libraries to be built up from reusable components. This is achieved through three key mechanisms: actions (ways of modifying code), inspection (ways of interrogating code), and references through a novel Cursor system. Where the original Exo relied on pattern matching for code references, Exo 2's Cursor system provides stable, relative references that can be maintained across transformations. This enables much more sophisticated scheduling libraries that can inspect and transform code while maintaining safety guarantees. The evaluation shows both versions can match or exceed the performance of heavily optimized libraries like OpenBLAS and Intel MKL, with Exo 2 requiring significantly less implementation effort through its library abstractions.

3.2.2 *Strategy Languages*

Strategy languages provide formal frameworks for expressing and reasoning about program transformations through composable rewriting rules. While scheduling languages focus on orchestrating predefined transformations, strategy languages offer richer abstractions for defining new transformations and controlling their application. These languages tackle fundamental challenges in transformation systems: how to compose basic rewrites into complex transformations, how to maintain correctness guarantees across compositions, and how to reason systematically about transformation properties.

STRATEGO[101, 102] Stratego pioneered many concepts in modern transformation systems by introducing programmable rewriting strategies. Unlike traditional rewrite systems that apply rules in a fixed order, Stratego allows developers to compose basic transformations using strategy combinators like sequential composition, choice, and recursion. The system's key innovation was treating transformation strategies as first-class entities that can be passed as arguments and returned as results, enabling powerful abstractions like generic traversal strategies that separate tree traversal logic from the actual transformations being applied. Its ideas about composable transformation strategies have been reimagined in modern systems like Elevate, which applies similar principles to optimizing high-performance code.

FORMAL FOUNDATIONS FOR STRATEGIC REWRITING Shoggoth [73] provides formal foundations for strategic rewriting that enable systematic reasoning about transformation correctness. Unlike previous formalizations that focused on operational semantics without divergence, Shoggoth introduces a comprehensive semantic model that accounts for non-determinism, errors, and divergence through a denotational semantics. Its key innovation is a location-based weakest precondition calculus that allows precise reasoning about transformation properties like termination and correctness. The calculus extends traditional weakest preconditions by making locations in the AST explicit parameters, enabling verification of traversal-based transformations. Through its mechanized proofs in Isabelle/HOL, Shoggoth demonstrates how formal methods can verify complex properties of strategic rewriting systems, such as ensuring that normalizing transformations preserve semantic guarantees.

3.2.3 *Mathematical Models for Program Transformation*

The approaches discussed so far rely primarily on operational specifications of program transformations - explicit sequences of steps that modify programs. In contrast, mathematical models provide alternative foundations for reasoning about and implementing transformations. These approaches trade practical simplicity for theoretical power, enabling systematic exploration of transformation spaces and automated reasoning about program equivalence.

POLYHEDRAL FRAMEWORKS The polyhedral model represents a fundamentally different approach to program transformation by modeling loop nests and array accesses as geometric objects in a mathematical framework. URUK [26] pioneered a transformation-centric view of polyhedral compilation by introducing a unified framework for composing complex loop transformations. Unlike scheduling languages like Halide, URUK represents transformations as sequences of affine functions that can be analyzed, composed, and even reversed. This mathematical foundation enables formal reasoning about transformation sequences while supporting loop optimizations like fusion, distribution, and tiling.

CHiLL [11] built upon URUK's concepts by providing a more practical, script-based interface to polyhedral transformations. Where URUK focused on mathematical elegance, CHiLL emphasized usability by introducing a high-level transformation specification language. These transformations can be automatically adapted to different loop nests, allowing compiler developers to express complex optimization sequences. Later extensions established a unified framework combining polyhedral and AST transformations [106]. This extended framework also allowed CHiLL to handle non-affine constructs like sparse matrix operations, where array accesses involve indirect indexing (e.g., accessing $A[B[i]]$ where B is an index array) or non-affine loop bounds. CHiLL's approach influenced subsequent work by demonstrating how to preserve the composability advantages of polyhedral frameworks while broadening their practical applicability.

These frameworks exemplify the evolution of polyhedral compilation: from URUK's purely mathematical foundations to CHiLL's more practical combination of polyhedral and AST transformations. Recent empirical studies have shown that polyhedral optimizations can significantly improve data locality through tiling, though they may sometimes hinder auto-vectorization opportunities [96]. This has motivated more comprehensive approaches, such as Vasilache et al.'s work [99] on jointly optimizing parallelism, locality, vectorization, and data layout transformations in a single formulation. These developments, along with parallel efforts to integrate polyhedral and AST-based transformations [86], demonstrate the field's progression toward more flexible and comprehensive program optimization approaches.

EQUALITY SATURATION Equality saturation offers a fundamentally different approach to program transformation than traditional rewrite systems. Instead of destructively applying transformations in sequence, it maintains an e-graph data structure [67, 68] that efficiently represents many equivalent program versions simultaneously. This allows the system to discover optimizations that would be difficult to find through sequential rewriting, as it can combine the results of different transformation paths without committing to specific choices prematurely.

The approach gained significant traction with the release of egg [103], a high-performance equality saturation engine implemented in Rust. egg's efficient implementation and ergonomic API have made equality saturation practical for a wider range of applications, leading to its adoption in various domains, from super-optimization to program synthesis. In production compilers, Cranelift demonstrates the practical viability of equality saturation through ægraphs [19], a variant designed for efficient optimization in a production setting. The approach has also shown promise in deep learning, where systems like TenSAT [105] apply equality saturation to optimize tensor computation graphs.

Koehler et al. [40] demonstrated equality saturation's potential for optimizing functional programs by applying it to Rise. By encoding Rise programs and their transformations in an e-graph, this approach can automatically discover complex optimization sequences that would be difficult to express through traditional rewriting strategies. This shows how equality saturation can complement strategy-based approaches to program transformation, potentially offering a way to automate the discovery of effective transformation sequences.

However, equality saturation still faces challenges in practical compiler implementations. The approach can be computationally expensive and memory-intensive, especially when working with large programs or many rewrite rules. While systems like SPORES have made progress in addressing these limitations through selective saturation strategies, the approach currently remains most effective for specialized optimization tasks rather than full compiler pipelines.

MAUDE [14] Term rewriting logic provides formal foundations for program transformation through systems like Maude, which specifies transformations as mathematical rewrite rules rather than imperative procedures. This approach enables for-

mal verification of transformation properties through built-in model checking and equational reasoning. While systems like Roșu and Șerbănuță [84]'s K framework demonstrate Maude's practical application in programming language semantics, the approach trades performance for mathematical rigor. Nevertheless, Maude's principled foundation for reasoning about transformations has influenced modern verification approaches in production compilers.

3.2.4 *Conclusion*

The examination of transformation systems reveals several fundamental design trade-offs. The separation of algorithmic specification from optimization strategy, first demonstrated by Halide and refined in subsequent systems, provides clear benefits for optimization exploration and maintainability. Systems like Exo show how composable transformation primitives can enable extensibility while maintaining semantic guarantees. Meanwhile, approaches like equality saturation demonstrate the power of exploring multiple transformation paths simultaneously, though at increased computational cost. The polyhedral model illustrates how mathematical foundations can enable powerful analyses within a restricted domain. These different approaches highlight a central tension in transformation system design: balancing expressiveness, composability, and practical applicability. Particularly noteworthy is the trend toward more programmable transformation frameworks that allow users to build complex optimizations from simpler, verified components.

3.3 COMPILER CORRECTNESS

Compiler correctness is crucial: a single compiler bug can propagate to millions of deployed applications, potentially causing critical failures or security vulnerabilities. As the previous section showed, modern compilers employ increasingly sophisticated transformations, making correctness verification more challenging. This raises a fundamental question: how can we ensure these transformations preserve program semantics?

This section examines different approaches to establishing compiler correctness, ranging from complete formal verification of entire compilers to practical techniques for validating specific transformations. While full verification provides the strongest guarantees, its significant engineering cost has moti-

vated the development of lighter-weight approaches that can be integrated into production compilers. These approaches often build upon each other, combining formal foundations with practical validation techniques to balance verification strength with implementation feasibility.

3.3.1 *Full Compiler Verification*

The most comprehensive approach to ensuring compiler correctness involves mathematically proving that the entire compilation pipeline preserves program semantics. This ambitious strategy provides the strongest possible guarantees but requires significant engineering effort and careful compiler design.

CompCert represents a milestone in this approach as the first realistic C compiler with complete mathematical proof of correctness [49]. Designed from the ground up with verification in mind, its architecture carefully balances complexity and verifiability through a series of intermediate languages with formally specified semantics. Its semantic preservation theorem guarantees that the compiler introduces no bugs during compilation: every behavior of the compiled program matches a possible behavior of the source program. This formal verification covers the entire compilation chain from C source to assembly code, including complex optimizations like constant propagation, common sub-expression elimination, and register allocation.

While CompCert's optimizations are more conservative than those of production compilers like GCC or LLVM, empirical studies have shown that CompCert-compiled code typically achieves 80% of the performance of GCC -O2 [50]. The project demonstrated that formal verification of a production-quality compiler is feasible, albeit requiring significant effort. CompCert's success influenced several subsequent verified compiler projects, notably CakeML [41], a verified compiler for a subset of MLor the VST-Floyd project [10] which builds on CompCert's verification approach to reason about C programs, demonstrating how verified compilation can serve as a foundation for broader program verification efforts..

3.3.2 *Practical Verification Approaches*

While full compiler verification provides the strongest guarantees, its substantial engineering costs have motivated the development of more practical approaches. These methods typically

focus on verifying specific aspects of compilation or individual compilation runs, trading completeness of verification for practicality, and integration with existing compilers.

TRANSLATION VALIDATION This pragmatic approach to compiler correctness verifies individual compilation runs rather than proving the correctness of the entire compiler. Instead of formally verifying the compiler implementation itself, translation validators check that each specific input program and its compiled output maintain semantic equivalence. By focusing validation effort on actual compilation instances, this approach can detect miscompilations even in complex, non-verified optimizing compilers without requiring changes to the compiler implementation.

Notable implementations include [66]'s validator for the GNU C compiler and [98]'s translation validator for LLVM's interprocedural optimizations, and Bang et al. [5]'s recent SMT-based framework for MLIR. The latter work is particularly relevant for modern compiler infrastructures, as it successfully validates complex transformations in machine learning compilers through novel techniques for handling floating-point arithmetic and tensor operations while also uncovering several specification-implementation mismatches in MLIR.

However, translation validation faces several challenges. First, establishing semantic equivalence becomes difficult for complex optimizations and loop transformations. Second, handling floating-point arithmetic requires careful trade-offs between precision and tractability, as shown by Bang et al. [5]'s progressive approximation techniques for floating-point operations. These limitations have motivated alternative approaches that provide different foundations for reasoning about compiler correctness.

FORMAL SEMANTICS AND CREDIBLE COMPILATION A different approach to verification builds on formal semantics to enable rigorous reasoning about compiler transformations. VeLLVM [107] provides a formal semantics for LLVM IR that captures nuances of undefined behavior and nondeterminism inherent in LLVM's design. Its key contribution is reconciling LLVM's intentional looseness around undefined behavior with formal verification techniques through multiple successively refined operational semantics. VeLLVM implements a nondeterministic semantics that models undef values as representing sets of possible values and includes deterministic refinements that

enable testing and extraction of executable verified implementations. Using Coq, it proves preservation and progress theorems that establish the soundness of the LLVM IR type system and formalization. The framework demonstrates its utility by extracting a verified implementation of the SoftBound transformation pass [65] that enforces spatial memory safety. While VeLLVM does not verify the full LLVM compiler, it provides a foundation for reasoning about LLVM program transformations and for extracting verified implementations of specific optimization passes.

Building on this formal foundation, Crellvm applies credible compilation to LLVM optimization validation, building on VeLLVM's formal semantics of LLVM IR. Where translation validation attempts to automatically verify each optimization instance without knowledge of the compiler's reasoning, credible compilation requires the compiler to generate explicit evidence of correctness in the form of proofs that capture the compiler's decision process. These proofs are then checked by a verified proof checker. The framework provides a variant of relational Hoare logic called ERHL that is specialized for LLVM IR and handles complexities like undefined behavior, memory operations, and SSA form. Building on VeLLVM's Coq formalization but significantly upgrading its memory model and adding support for additional instructions like switch, Crellvm provides formal guarantees about optimization correctness. Applied to major LLVM optimization passes, including mem2reg and global value numbering, Crellvm found several long-standing miscompilation bugs that had evaded testing approaches. Unlike translation validation, which can produce false positives when it fails to automatically discover a validity proof, Crellvm's explicit proof generation provides precise feedback about optimization correctness. While not providing CompCert's comprehensive guarantees, Crellvm demonstrates that selective formal verification can provide strong correctness assurances for production compiler optimizations. For optimizations validated with verified inference rules, it achieves the same level of guarantee as CompCert's verification.

TARGETED VERIFICATION OF PEEPHOLE OPTIMIZATIONS

The Alive project [52] demonstrates how formal verification can be effectively targeted at specific compiler components. Alive introduced a domain-specific language for specifying LLVM peephole optimizations, automatically proving their correctness

using SMT solvers. Despite being limited to peephole optimizations without cyclic control flows, the tool's effectiveness at finding bugs in LLVM's InstCombine pass led to a significant change in LLVM's development process: all new InstCombine optimizations must now be verified using Alive before acceptance.

Alive2 [54] significantly extends this approach by combining translation validation with direct LLVM IR verification. Unlike its predecessor's domain-specific language, Alive2 validates transformations by comparing the semantics of LLVM IR before and after optimization. This enables verification of more complex transformations, including floating-point arithmetic and memory operations.

3.3.3 *Conclusion*

These various approaches to compiler verification represent different trade-offs between comprehensiveness, practicality, and integration with production compilers. CompCert demonstrates the feasibility of full verification through a complete compiler architecture built from the ground up with verification in mind. Translation validation offers pragmatic checking of individual compilations but struggles with complex transformations. VeLLVM provides formal foundations for reasoning about LLVM semantics, while Crellvm builds on this to enable verified optimization validation with explicit correctness proofs. Alive shows how targeted formal methods can effectively verify specific classes of optimizations. Together, these approaches demonstrate a rich spectrum of verification techniques, each offering different balances of verification strength, implementation effort, and practical applicability. The fundamental challenge of verifying modern multi-level IR compilers like MLIR, while maintaining their extensibility and performance, remains an important open problem, with no clear single solution emerging from existing approaches.

4

A COMPOSITIONAL PROGRAM REPRESENTATION FOR MLIR

This chapter is largely based on the publication: "Integrating a functional pattern-based IR into MLIR" by Lücke, Steuwer, and Smith published at CC'21

In this chapter, we address the challenges and solutions related to the continued specialization in hardware and software development due to the end of Moore's law. This paradigm shift requires reconsidering the fundamental design choices in compilers, particularly for domain-specific languages (DSLs). The era where a single universal compiler intermediate representation (IR) sufficed for all essential optimizations is behind us. There is a growing need for novel high-level IRs leveraging established compiler infrastructures.

We introduce a functional pattern-based intermediate representation implemented within the SSA-based MLIR framework. The IR captures program semantics as compositions of common computational patterns, which facilitates rewrite-based optimizations. Through the integration with other IRs, we demonstrate the compilation process of a neural network represented as a TensorFlow graph down to optimized LLVM code via our functional pattern-based IR.

This implementation represents the first practical integration of a functional pattern-based IR with other IRs, enabling the construction of sophisticated code generators for domain-specific languages. We examine how our approach interacts with existing IRs, addressing the direct correspondence between functional programming and SSA, and evaluate whether this method incurs any performance penalties when compiling to imperative loop-based code.

By exploring these aspects, we aim to showcase the practicality and efficiency of incorporating a functional pattern-based IR into modern compiler toolchains, highlighting its potential in optimizing computations for specialized hardware architectures used in domains such as deep learning and physical simulations.

4.1 INTRODUCTION

Software and hardware are becoming increasingly specialized. Only a few years ago, the focus was on optimizing single-

threaded performance represented by SPEC benchmarks for general-purpose CPUs. Now, interest has shifted away from general-purpose computing into application-specific domains – none more prominent than deep learning. Deep learning hardware today is represented by CPUs, GPUs, FPGAs and a zoo of specialized hardware devices such as Google's TPU.

Traditional compiler designs evolved around the idea of a *"one size fits all"* single intermediate representation (IR) that unifies the representation of multiple high-level programming languages and allows optimizations to use a single representation for generating optimized code for different hardware targets. The highly successful LLVM compiler infrastructure [44] is built around this idea.

But LLVM IR provides a single level of abstraction that is often too low-level and difficult for optimizations to exploit the rich high-level semantics of domain-specific languages. For this reason many higher level IRs have been developed for specific domains such as machine learning (e.g, the graph-based IRs of TensorFlow [57], PyTorch [71], and TVM [12]) or image processing (e.g., Halide [77]), but also more generic higher level IRs such as INSPIRE [37] and Thorin [47].

LIFT [92] is an unconventional higher level IR that represents computations as compositions of functional patterns. This design is easily extended to new application domains, for example, by adding new patterns to support stencil computations [29]. Optimizations are expressed as semantics-preserving rewrite rules that are either applied automatically as part of a stochastic search process [91] or their application can be controlled by a developer precisely [28]. Using this approach LIFT has demonstrated high performance on linear algebra and stencil codes used in machine learning and physical simulations for hardware architectures ranging from multi-core CPUs to mobile- and desktop-class GPUs.

Employing a novel IR, such as LIFT, in practical end-to-end toolchains is a challenging task. The recently proposed MLIR project [45] seems a promising solution as it aims to provide a common framework to enable the integration of multiple IRs. Individual IRs are implemented as *dialects* following a basic SSA-based design. Each dialect can define a custom type system and operations as well as optimization passes. For interaction among dialects MLIR provides common infrastructure to facilitate the translation from one dialect to another.

But how do we encode a functional pattern-based IR such as LIFT in the SSA-based MLIR framework? Thanks to Appel [2], we know for a long time about a direct correspondence between functional programming and SSA, but what does this look like in practice? How does a functional IR integrate with other IRs in MLIR? While our functional IR is convenient for expressing domain-specific computations at a high level, will we pay a performance penalty when compiling to imperative loop-based code?

In this chapter, we are answering these questions. We present a practical implementation of the functional pattern-based IR RISE [28] as an MLIR dialect. We discuss its implementation (Section 4.3) and its integration with other MLIR dialects (Section 4.4) to build a practical end-to-end code generation solution for machine learning that progressively lowers the representations from a domain-specific TensorFlow graph to our generic high-level functional pattern-based representation before lowering it to a lower level polyhedral loop-based representation and eventually to LLVM IR. Our evaluation (Section 4.5) shows that our implementation of the functional pattern-based IR introduces negligible compile time overhead and generates code with no runtime overhead.

4.2 MOTIVATION AND BACKGROUND

4.2.1 *What's Wrong with Existing Compiler IRs?*

Specialized domain-specific compilers have become an integral component of achieving high performance in many crucial application domains, such as machine learning. But according to Paul Barham and Michael Isard, two of the original authors of TensorFlow, machine learning systems are stuck in a rut [6]. They argue that while TensorFlow and similar frameworks enabled great advances in machine learning, their current design and implementations focus on a fixed set of monolithic and inflexible kernels (such as matrix-multiplication) that are expressed as fixed nodes in the IR. They continue to say that the *"reliance on high performance but inflexible kernels reinforces the dominant style of programming model"* and argue that *"these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress"* in machine learning.

To overcome these problems, we need new intermediate representations that break up the monolithic and inflexible ker-

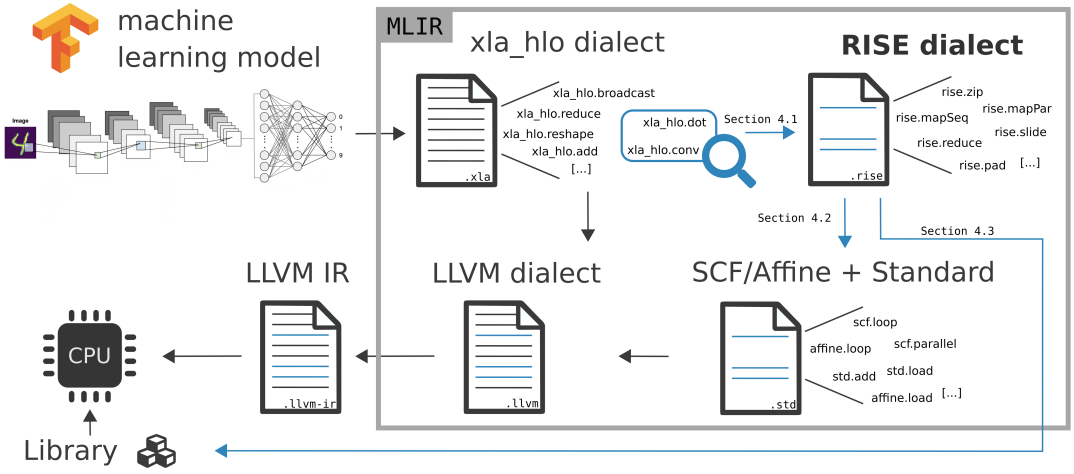


Figure 4.1: End-to-end code generation using the pattern-based Rise intermediate representation implemented as an MLIR dialect. A TensorFlow machine learning model (top left) is represented in MLIR with the XLA HLO dialect. Supported operators are lowered into compositions of computational patterns in Rise (top right) before being lowered to a loop-based representation (bottom right) that is compiled to LLVM IR (bottom left). Or alternatively, Rise patterns are lowered directly into library code.

nels and represent computations using more flexible and finer-grained building blocks. Pattern-based IRs are built around this idea and are an interesting sweet spot between specialized high-level IRs with monolithic domain-specific abstractions and low-level loop-code or three-address style IRs similar to LLVM. In pattern-based IRs, computations are represented at a high level as compositions of generic computational patterns common across many domains. This allows to easily perform algorithmic optimizations at the right abstraction level as demonstrated by Lift [89, 92] that encodes optimizations as rewrites of pattern-based programs.

4.2.2 *Rise: A Functional Pattern-Based IR*

Rise [28] is a functional pattern-based IR in the style of Lift. Rise provides a set of data-parallel high-level patterns that are composed to describe computations over higher dimensional arrays (i.e. tensors) such as matrix multiplication. For that, the `map` pattern is used twice (in lines 2 and 3) to apply the dot product computation to each combination of a row of matrix A and a column of matrix B:

```
1 fun( (A: N.K.f32, B: K.M.f32),
2  A |> map(fun(arow,
3   B |> transpose |> map(fun(bcol,
4    zip(arow,bcol) |> map(mult) |> reduce(add,0)))))))
```

Rise uses the lambda-calculus to compose patterns. A lambda expression (i.e., anonymous function) takes the form `fun(x, e)` in our notation. Function application can be expressed either using the forward application operator `|>` as in `x |> f`, or using traditional notation `f(x)`. Type annotations are required for parameters of top-level lambda expressions. For example, matrix `A` of type `N.K.f32` represents an $N \times K$ matrix of 32-bit floating-point values (line 1).

High-level programs are gradually transformed into low-level programs by applying semantics preserving rewrite rules that encode optimization and implementation decisions. For example, fusing **`map`** and **`reduce`** to ensure they are computed in a single loop is expressed with this rule:

$$\texttt{map}(f) \texttt{|>} \texttt{reduce}(\oplus, 0) \mapsto \texttt{reduceSeq}(\texttt{fun((a,x), a} \oplus f\texttt{(x)),0)}$$

Prior work on Rise [28] has shown that compiler optimizations such as tiling and vectorization are expressible as compositions of rewrites. The closely related Lift project has demonstrated high performance by exploring optimization choices encoded as rewrite rules for tensor-algebra [91], stencil computations and kernel convolutions [29].

4.2.3 *End-to-End Compilers by Integration of IRs*

Rise (and Lift) are implemented in the Scala programming language, making the development of end-to-end compiler solutions e.g. for machine learning challenging. Even for IRs implemented in easier-to-integrate languages such as C++, e.g., INSPIRE [37] or Thorin [47], integration is hard in practice due to the code required to convert between IRs.

We are interested in developing end-to-end compiler solutions for machine learning to allow the development of novel optimizations and analyses by leveraging the best of what is available from other domain experts.

This chapter presents a C++-based implementation of Rise in MLIR and shows how to build integrated compiler solutions using this novel implementation. Figure 4.1 shows a prototype machine learning compiler that we built. Starting from an unmodified TensorFlow machine learning model (top left), e.g.,

for handwriting detection using the popular MNIST dataset, the model is directly encoded in the existing MLIR XLA dialect (top middle). Computational intensive operations such as matrix multiplication encoded as `xla_hlo.dot` operations are lowered into the MLIR implementation of RISE (top right). RISE expressions are then transformed into lower level IRs such as the polyhedral affine dialect and the standard dialect (bottom right), from which eventually LLMV IR is generated (bottom left). Crucially, we can easily leverage existing optimizations – at the domain-specific level as well as perform polyhedral optimizations and ultimately perform classical compiler optimizations before code generation – while building a foundation to drop in rewrite-based optimizations at the pattern-based level.

The remainder of the chapter discusses the technical details of how we achieved this integration by starting with a discussion on how to implement the functional RISE IR in the SSA-based MLIR framework in the following section.

4.3 RISE AS AN MLIR DIALECT

Building on MLIR's foundational concepts discussed in Section 2.1, we implement RISE as a custom MLIR dialect. This section presents how we map RISE's functional concepts to MLIR's infrastructure, starting with the type system and followed by the implementation of lambda calculus operations and computational patterns.

4.3.1 *Types*

RISE is a functional IR where computations are represented by compositions of pattern applications. Patterns are represented as built-in functions and have a corresponding function type. The grammar of RISE types is shown in Figure 4.2. Function types are separate from data types, only allowing data types to be stored in memory. RISE data types include array, tuple, and scalar types. For arrays, their length is tracked in the type. Different from many tensor representations, higher-dimensional tensors are represented by nesting array types such as the matrix type $N.M.\texttt{f32}$ from the prior example.

The grammar directly corresponds to the class hierarchy shown in Figure 4.3. The RISE dialect follows the MLIR notation to prefix types by `!rise`. For example, a function type with one argument and return type is written: `!rise.fun<scalar<i32> ->`

$$
\begin{aligned}
T &::= T \to T \mid D && \textit{Function and Data Types} \\
D &::= N.D \mid D \times D \mid S && \textit{Array, Tuple, and Scalar Types} \\
S &::= \texttt{i32} \mid \texttt{f32} \mid \ldots && \textit{standard Scalar Types} \\
N &::= 0 \mid \ldots \mid N + N \mid N * N \mid \ldots && \textit{Array Length}
\end{aligned}
$$

Figure 4.2: Grammar of RISE types

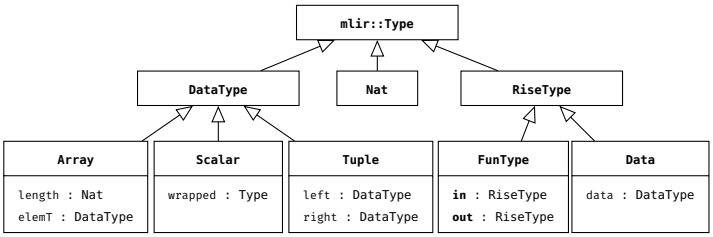


Figure 4.3: RISE dialect types

`scalar<i32>>`. Note, `i32` is an existing MLIR standard type and it is reused by wrapping it in a `!rise.scalar` type.

4.3.2 *Lambda Calculus as MLIR Operations*

RISE is a functional IR based on lambda calculus with two core operations: function abstraction and application. We represent these with the **`lambda`** and **`apply`** operations.

Figure 4.4 shows their implementations as subclasses of the `mlir::Op` superclass. The **`lambda`** operation together with its associated `FunType` models a lambda expression by wrapping an MLIR region containing a single block whose arguments become the arguments of the lambda expression. The **`apply`** operation models a function application (or function call). The function can be any value with a `FunType`, such as **`lambda`**s or patterns. The argument must have a matching RISE type.

The listing below shows how **`lambda`** and **`apply`** are used to represent a call to the identity function: `y |> fun(x => x)`

```
1 %id  = rise.lambda (%x)   : !rise.fun<scalar<i32> ->
2                                    scalar<i32>> {
3     rise.return %x        : !rise.scalar<i32>
4 }
5 %res = rise.apply %id, %y : !rise.scalar<i32>
```

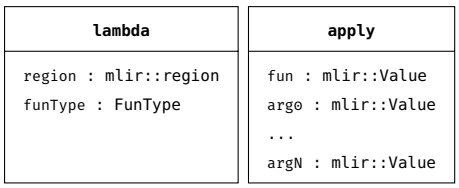


Figure 4.4: Implementation of **lambda** and **apply** classes

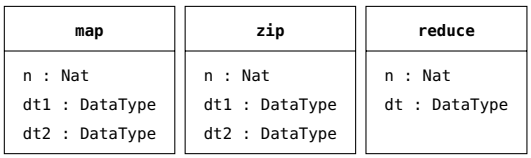


Figure 4.5: Implementation sketches of pattern classes

4.3.3 RISE *Patterns as MLIR Operations*

Computations in RISE are represented by computational patterns such as **map**, **reduce** and **zip**. Each pattern is represented by an operation and implemented as a subclass of the `mlir::Op` class. Implementation sketches are shown in Figure 4.5.

Each pattern in RISE has a function type. For example, **map**:

```
map : (n: Nat) ↦ (dt1: DataType) ↦ (dt2: DataType) ↦
        (dt1 → dt2) → n.dt1 → n.dt2
```

The ↦ arrow represents a special function type for representing polymorphism, i.e. for introducing type variables. The MLIR RISE dialect does not support polymorphic function types and instead provides the type arguments when a pattern is created and stores them in the pattern classes. This design results in patterns having monomorphic function types that are straightforward to handle. Applying **map** to a function `%f` and an array of 1024 `int`s is represented as:

```
1 %m   = rise.map #nat<1024> #scalar<i32> #scalar<i32>
2    : !rise.fun< fun< scalar<i32> -> scalar<i32> > ->
3           fun< array<1024, scalar<i32>>          ->
4                array<1024, scalar<i32>> > >
5 %res = rise.apply %m, %f, %array
6    : !rise.array<1024, scalar<i32>>
```

4.3.4 *Matrix Multiplication in the Rise MLIR Dialect*

Figure 4.6 shows an example matrix multiplication in the RISE dialect. This corresponds to the functional RISE expression seen

earlier after the **map-reduce** fusion rule has been applied. Some type annotations have been removed for readability.

```
A |> map(fun(arow, B |> transpose |> map(fun(bcol,
     zip(arow,bcol) |> reduce(fun((ab,acc), (ab1×ab2)+acc),0)))))
```

```
func @mm_fused(%outArg, %inA, %inB) {
 %A = in %inA
 %B = in %inB
 %t = rise.transpose #rise.nat<2048>
                     #rise.nat<2048> #rise.scalar<f32>
 %B_t = rise.apply %t, %B
%m1fun = lambda (%arow) -> array<2048, scalar<f32>> {
 %m2fun = lambda (%bcol) -> scalar<f32> {
  %zipFun = zip #nat<2048> #scalar<f32> #scalar<f32>
  %zippedArrays = rise.apply %zipFun, %arow, %bcol
  %reduceLambda = lambda(%tuple, %acc)->scalar<f32> {
   %fstFun = rise.fst #scalar<f32> #scalar<f32>
   %sndFun = rise.snd #scalar<f32> #scalar<f32>
   %first  = rise.apply %fstFun, %tuple
   %second = rise.apply %sndFun, %tuple
   %result = rise.embed(%first, %second, %acc) {
    %product = mulf %first, %second :f32
    %result = addf %product, %acc : f32
    return %result : f32
   }
   return %result : scalar<f32>
  }
  %init = rise.literal #lit<0.0>
  %reduceFun = reduceSeq #nat<2048> #tuple
  %result = rise.apply %reduceFun, %reduceLambda,
                       %init, %zippedArrays
  return %result : scalar<f32>
 }
 %m2 = mapSeq #nat<2048> #array<2048, scalar<f32>>
                         #scalar<f32>
 %result = rise.apply %m2, %m2fun, %B_t
 return %result : array<2048, array<2048, scalar<f32>>>
}
%m1 = mapSeq  #nat<2048> #array<2048, scalar<f32>>
                           #array<2048, scalar<f32>>
%result = rise.apply %m1, %m1fun, %A
 out %outArg <- %result
 return
}
```

Figure 4.6: 2048x2048 matrix multiplication in the RISE dialect

The dot product computation is defined in the innermost (green) boxes with the **zip** and **reduceSeq** patterns. This computation is nested inside two **lambda**s (`%m1fun` and `%m2fun`) that are used as arguments when calling the **map** patterns and applying them to matrix `%A` and the transposed matrix `%B_t`.

The multiplication and addition performed on the scalar values is represented using the standard MLIR operations **mulf** and **addf**. These are nested inside of an **rise.embed** operation that makes

the named MLIR values with RISE `ScalarType`s (here: `%first`, `%second`, and `%acc`) available inside the nested block with their types unwrapped (here with type `f32`). The **`rise.in`** and **`rise.out`** operations allow the integration with external values that have non RISE types.

4.3.5 *Building Rise IR in C++*

When using our RISE dialect as a tool, it is tedious to construct the RISE IR from scratch, supplying type information for all operations explicitly. To simplify building RISE expressions, we have developed an easy-to-use C++ API using MLIR builders. Listing 4.1 shows how this API is used for building the RISE IR code shown in Figure 4.6. The `makeRiseProgram` function accepts the output value (`C`) and the input values (`A`, `B`) for the computation as arguments and handles generation of the **`rise.in`** and **`rise.out`** operations. Each highlighted C++ function builds the corresponding operations without the need to specify types explicitly as they are inferred. We use C++ lambda expressions to build the **`rise.lambda`** operations. The **`rise.apply`** operations are inserted automatically.

```
1  makeRiseProgram(C, A, B)([&](Value A, Value B) {
2    return mapSeq([&](Value arow) {
3      return mapSeq([&](Value bcol) {
4        return reduceSeq([&](Value tuple, Value accum){
5          return embed({fst(tuple), snd(tuple), accum},
6                [&](Value a, Value b, Value accum) {
7                  return accum + (a * b);  });
8        },literal(scalarF32(), "0.0"),zip(arow, bcol));
9      }, transpose(B));
10   }, A);  });
```

Listing 4.1: C++ API to build a composition of RISE patterns modelling matrix multiplication as shown in Figure 4.6

4.4 INTEGRATION WITH OTHER MLIR DIALECTS

The MLIR infrastructure allows us to easily integrate RISE with other dialects. In this section, we are going to discuss the integration that allows the building of a full machine-learning compiler by compiling a TensorFlow machine-learning model via our functional pattern-based dialect into low-level loop-based code. First, We will discuss how to lower domain-specific dialects into RISE. Then, we will discuss how a program in the RISE dialect

is lowered into a loop-based representation. As an alternative, we discuss the lowering from RISE directly to optimized library implementations.

4.4.1 *Lowering Domain Specific Dialects to Rise*

RISE's functional, pattern-based nature makes it an attractive target for domain-specific dialects. Its mathematical foundations, based on well-understood patterns like map and reduce, provide a natural mapping from domain-specific operations. For instance, recent work suggests that tensor operations expressed in Einstein notation can be systematically translated into compositions of these functional patterns [7, 39].

Crucially, RISE avoids committing to implementation details like memory allocation or execution models prematurely. Instead, it maintains program semantics at a level that exposes optimization opportunities through pattern rewriting. This allows domain-specific optimizations to be expressed as pattern compositions while deferring hardware-specific decisions to later compilation stages.

Let us consider machine learning as a popular domain: The MLIR XLA_HLO dialect encodes TensorFlow graphs representing the computation of a neural network. Operations in this graph represent computations processing tensors, such as the `xla_hlo.dot` operation that represents a tensor dot product. Depending on the tensors' dimensionality, this represents a vector dot product, a matrix multiplication, or their higher dimensional equivalent.

To lower XLA operations to RISE, the MLIR infrastructure allows for defining declarative rewrites that automatically match specific operations and sequences of operations as a starting point to modifying the IR. For lowering the `xla_hlo.dot` operation, we provide an MLIR rewrite that checks for the types of the tensors to determine the equivalent RISE expression that is emitted to replace the operation. For the two-dimensional matrix multiplication, we use the C++ builder API to generate the RISE IR (Listing 4.1). Currently, our implementation supports lowering `xla_hlo.dot` and convolutions, which are two of the most computationally intensive operations in machine learning. Our C++ builder API and the MLIR declarative rewriting make it straightforward to target RISE from other domain-specific MLIR dialects.

4.4.2 *Lowering Rise to Loop-Based Dialects*

The functional patterns in Rise provide a convenient abstraction to express computations at a high level. Before execution, these functional patterns have to be lowered into imperative code and eventual LLVM instructions that can easily be compiled into executable code. For this process, we gradually lower the MLIR Rise dialect into a loop-based MLIR dialect from which LLVM IR is generated.

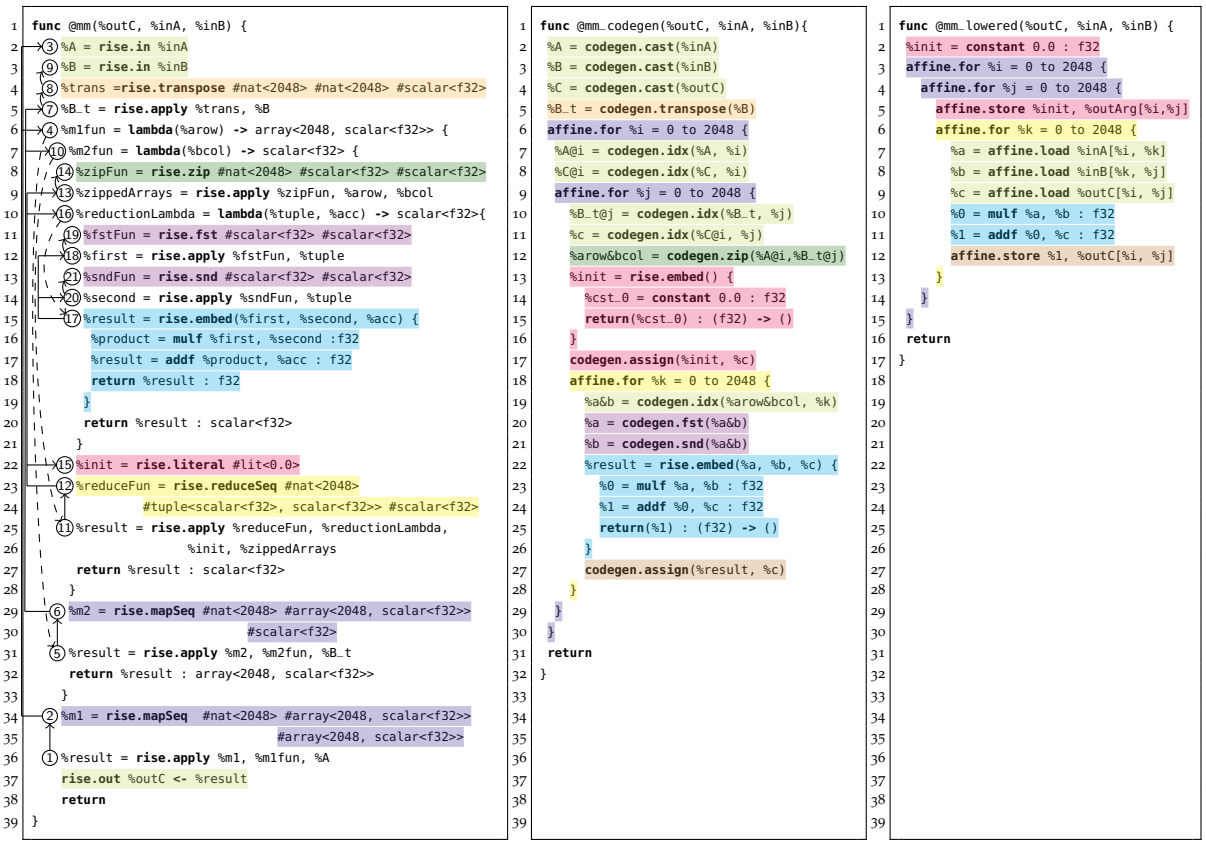
While the high-level Rise representation may appear more verbose due to its explicit representation of computational patterns and their composition, this verbosity directly encodes crucial algorithmic properties. For example, a map pattern represents potential data parallelism, and function composition directly shows a data flow relationship. During lowering, these high-level computational patterns are transformed into more compact but lower-level control flow and memory access patterns, effectively trading algorithmic expressiveness for concrete implementation decisions and one set of optimization opportunities for another. While the lowered representation uses fewer instructions, this apparent simplicity comes at the cost of high-level program properties and optimization flexibility. While techniques exist to recover high-level patterns from lower-level representations [9, 16, 64], progressive lowering provides a more natural path without requiring heroic efforts to recover semantic information. Rise serves as a key abstraction level in this process, positioned between domain-specific dialects and lower-level implementations where pattern-based optimizations can be expressed directly.

We adopt a formal compilation method for translating a functional pattern-based IR to imperative code originally presented in [3]. This lowering process is complete and capable of lowering all possible Rise expressions into loop code. It is split in two phases:

1. **Functional → Intermediate IR**:
   The functional Rise IR is lowered into an intermediate imperative representation without the functional lambda calculus representations and the functional patterns. This intermediate representation still uses Rise specific types and has not yet resolved indices for accessing memory.

2. **Intermediate IR → Target Representation**: The indexing into multi-dimensional arrays is resolved.

Splitting the lowering into these two phases greatly improves the composability of the implementation. While the process is fully generic, we explain it by example to be more approachable without requiring a background in functional programming. Figure 4.7 shows the lowering of matrix multiplication in the RISE dialect (4.7a) to a combination of the loop-based Affine dialect and the standard MLIR dialect (4.7c) via the intermediate imperative representation (4.7b). The colors show which parts of the left-hand side are translated into which parts in the middle and on the right-hand side.



(a) Functional RISE IR (b) Imperative RISE IR + Affine IR (c) Affine IR

Figure 4.7: Lowering of matrix multiplication from RISE (left) via the intermediate imperative representation (middle) to the affine loop-based dialect (right). Colors indicate which part from the left is lowered into which part in the middle and right.

4.4.2.1 *Phase 1: Functional → Intermediate IR*

In this first lowering phase, we eliminate all operations modeling the functional lambda calculus and patterns from the program. Patterns such as **`mapSeq`** and **`reduceSeq`** are transformed into loops, while patterns such as **`zip`** and **`fst`** are rearranged to model the multidimensional indexing of memory. For each pattern there exist translation rules that explain how the pattern is lowered. These rules are detailed in [3]. Lambda expressions and function application nodes that make the control flow in RISE explicit are removed as the control flow is now expressed as an imperative program performing a sequence of loop-based computations.

For multi-dimensional memory accesses of RISE values at this intermediate stage, we introduce a number of intermediate operations that are not exposed to the user:

- **`rise.codegen.idx`**
- **`rise.codegen.assign`**
- **`rise.codegen.fst`**
- **`rise.codegen.snd`**
- **`rise.codegen.zip`**
- **`rise.codegen.transpose`**
- **`rise.codegen.slide`**
- **`rise.codegen.pad`**

Lowering by example: Matrix Multiplication

We explain the lowering process by example following the matrix multiplication. Figure 4.7 shows on the left the matrix multiplication represented in the RISE dialect. To lower this to the intermediate imperative representation in the middle, we start at the end of the RISE program with the **`rise.out`** operation that specifies the value representing the computed result. From here the lowering process chases all referred SSA values (as indicated by arrows in Figure 4.7 (a)) and visits the defining operation in the highlighted order form ① to ㉑. As RISE programs are compositions of pattern and function applications, the referenced operations are mostly **`rise.apply`** nodes whose lowering depends on the specific pattern or lambda which is applied, as well as its arguments.

For lowering the entire matrix multiplication example we first lower the ① **`rise.apply`** node in line 36 that was referred to by **`rise.out`** operation and represents the call to the outermost **`rise.mapSeq`** ② in the functional expression.

Lowering ***mapSeq***

Listing 4.2 shows the pseudo code for lowering an application of the **mapSeq** pattern. These steps are performed (The numbering matches the line numbers in Listing 4.2):

1 First, the input array is lowered. Here this is ③ `%A` represented as a **rise.in** operation that is translated into the **codegen.cast** operation in line 2 Figure 4.7 (b).

2 A for loop is generated by building an **affine.for** operation (line 6 Figure 4.7 (b)) with the `generate_affine_for` helper function using the array length captured in the RISE type as the iteration bound of the loop.

3&4 The loop index is used to generate **codegen.idx** operations (lines 7 and 8 Figure 4.7 (b)) representing indexing into the already translated input and output values.

5 Finally, a temporary **rise.apply** is created representing the application of the indexed input value (`%A@i`) to the lambda (④ `%m1fun`). Then the lowering of this temporary **apply** (described next) is invoked using the indexed output value (`%C@i`) as the output location to write to.

Lowering ***lambda***

To lower the application of a **lambda** operation, we first substitute the block argument of the lambda expression with the values that are applied to it. In the example, we substitute `%arow`, the parameter of the **lambda** (line 6 Figure 4.7 (a)) with `%A@i` the argument created in the prior step. After the substitution, we start the lowering from the **return** operation traversing the nested block backward.

In this example, we encounter the ⑤ **rise.apply** operation in line 31 Figure 4.7 (a) that represents another application of the

```
   // op = rise.apply(mapSeq n s t, fun, input)
   void lowerAndStore(ApplyOp op, Value out) {
1   Value in = lower(op.input);
2   generate_affine_for(0, op.n, inc, [&](index i){
3    Value inAtI = idx_gen(in, i);
4    Value outAtI = idx_gen(out, i);
5    lowerAndStore(rise.apply(op.fun,inAtI),outAtI);});}
```

Listing 4.2: Pseudo code for translating the **mapSeq** pattern

mapSeq pattern ⑥. To lower this operation we just recursively invoke the `lowerAndStore` function as discussed before that visits nodes ⑦ – ⑩. The only noticeable difference is that we pass `%C@i` as the output value to write to, ultimately resulting in a multi-dimensional indexing expression that is resolved in the second phase of our lowering. Inside of the second **lambda** we encounter ⑪ the application of the **reduceSeq** pattern ⑫ in line 25 Figure 4.7 (a) for which we describe the lowering next.

Lowering ***reduceSeq***

Listing 4.3 shows the pseudo-code for lowering application of the **reduceSeq** pattern in which the following steps are processed line-by-line:

1 First, the input array is lowered. In the example, this is ⑬ the result of the application of **zip** ⑭ with arguments `%arow` and `%bcol` (line 9 Figure 4.7 (a)). By applying the lambdas to the indexed matrices, these have been substituted by `%A@i` and `%B_t@j`. We emit a **codegen.zip** with these arguments in line 12 Figure 4.7 (b).

2 Next, the initialization of the reduction is lowered. In the example, this is ⑮ a **rise.literal** operation with the float value `0.0` specified as an attribute. Literals are simply lowered to the corresponding **constant** operation of the standard MLIR dialect (line 14 Figure 4.7 (b)). The type of the **constant** operation is the MLIR standard `f32` type. To make this accessible as a RISE type, we generate an enclosing **rise.embed** operation (line 13 Figure 4.7 (b)).

3 To initialize the reduction accumulator, we assign the initialization value to the output value that we use as storage for the accumulator value. This is done using **rise.codegen.assign**. In the example, the output value at this

```
// op = rise.apply(reduceSeq n s t, fun, init, input)
void lowerAndStore(ApplyOp op, Value out) {
1  Value in = lower(op.input);
2  Value init = lower(op.init);
3  assign_gen(init, out);
4  generate_affine_for(0, op.n, inc, [&](index i) {
5   Value inAtI = idx_gen(in, i);
6   lowerAndStore(rise.apply(op.fun,inAtI,out),out);
7  }); }
```

Listing 4.3: Pseudo code for translating the **reduceSeq** pattern

stage in the lowering is `%c` (line 11, Figure 4.7 (b)), which represents an individual element of the output matrix C.

4 Then the reduction loop is generated (line 18 Figure 4.7 (b)) using the input array length as the upper bound.

5 The loop index is then used to generate **`codegen.idx`** operation (line 19 Figure 4.7 (b)) indexing into the already translated input that in the example represents the zipped row and column (`%arow&bcol`).

6 Finally, a temporary **`rise.apply`** is created representing the application of the indexed input value and the accumulator to the reduction lambda ⑯. Then the lowering of this temporary **`apply`** is invoked using the accumulator as the output location to write to.

Lowering ***`embed`***

Nested inside the `%reductionLambda` is an **`embed`** operation ⑰ that is lowered next. The lowering is performed in two steps:

1 **`embed`**'s arguments (line 15 Figure 4.7 (a)) are lowered:
   - The accumulator `%acc` is already lowered at this point.
   - `%first` and `%second` are applications (⑱ and ⑳) of the **`rise.fst`** ⑲ and **`rise.snd`** ㉑ operations. Their lowering produces the intermediate **`rise.codegen.fst`** and **`rise.codegen.snd`** operations (lines 20 & 21 Figure 4.7 (b)).

2 **`embed`** remains unchanged, and a **`codegen.assign`** is generated (line 27 Figure 4.7 (b)), representing writing the computed result to the indicated output.

Now all **`rise.apply`** operations with patterns such as **`mapSeq`** and **`reduceSeq`**, as well as all **`lambda`** and **`embed`** operations have been lowered. Only `rise.codegen` operations remain, as can be seen in Figure 4.7 (b). These are resolved to indices next.

4.4.2.2 *Phase 2: Intermediate IR → Target IR*

In this second phase, the multi-dimensional indexing is resolved, generating standard MLIR `load` and `store` operations.

Figure 4.9 visualizes the process of generating the load of a value of matrix B (line 8 Figure 4.7 (c)). Starting from the `%b`

```
resolveIndex(cast(%val, type), path)                 {  generateLoad(%val, path);                                  }
resolveIndex(embed(%args, region), path)             {  resolveIndex(%args, path); inline(region);                 }
resolveIndex(idx(%array,%iv), path)                  {  resolveIndex(%array, %iv :: path);                         }
resolveIndex(zip(%lhs, %rhs), %iv::(fst|snd)::path)  {  resolveIndex((%lhs|%rhs), %iv :: path);                    }
resolveIndex(fst(%tuple), path)                      {  resolveIndex(%tuple, fst :: path);                         }
resolveIndex(snd(%tuple), path)                      {  resolveIndex(%tuple, snd :: path);                         }
resolveIndex(split(%array, n), i :: j :: path)       {  resolveIndex(%array, i∗n+j :: path);                       }
resolveIndex(join(%array, n), i :: path)             {  resolveIndex(%array, i/n :: i%n :: path);                  }
resolveIndex(transpose(%array), i :: j :: path)      {  resolveIndex(%array, j :: i :: path);                      }
resolveIndex(slide(%array, stride), i :: j :: path)  {  resolveIndex(%array, i∗stride+j :: path);                  }
resolveIndex(pad(%array, n, l, r), i :: path)        {  resolveIndex(%array,
                                                          ((i < l) ? 0 : (i < l+n) ? index : n−1) :: path);        }
```

(a) Resolve load indexing of `rise.codegen` operations

```
resolveStoreIndex(assign(%val, %storeTo))            {  resolveIndex(%val, {});
                                                        resolveStoreIndex(%storeTo, {});                          }
resolveStoreIndex(cast(%val, type), path)            {  generateStore(%val, path);                                 }
resolveStoreIndex(idx(%array, %iv), path)            {  resolveStoreIndex(%array, %iv :: path);                    }
resolveStoreIndex(split(%array, n), i :: path)       {  resolveStoreIndex(%array, i/n :: i%n ::path);              }
resolveStoreIndex(join(%array, m), i :: j :: path)   {  resolveStoreIndex(%array, i∗m+j :: path);                  }
resolveStoreIndex(transpose(%array), i :: j ::path)  {  resolveStoreIndex(%array, j :: i :: path);                 }
```

(b) Resolve store indexing of `rise.codegen` operations

Figure 4.8: Resolving indices and generating `load` and `store` operations by consuming the access path.

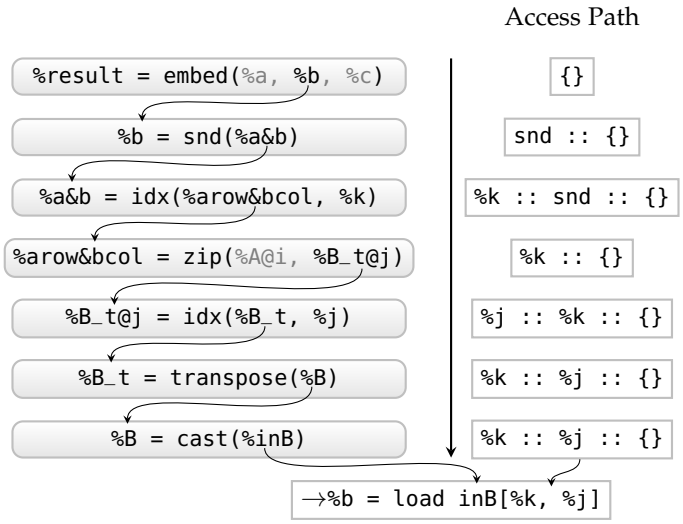


Figure 4.9: Resolving indices for reading from matrix B

argument of the **`rise.embed`** operation in line 22 Figure 4.7 (b) we chase the def-use chains of the SSA-values and construct an *access path* shown on the right of Figure 4.9.

For every operation encountered on the def-use chain traversal, we modify the access path as indicated in Figure 4.8. Fig-

ure 4.8a shows the path changes when resolving a load and Figure 4.8b for a store. There are a few RISE patterns (such as **`zip`**) that only influence the reading and, therefore, only appear when generating loads.

In Figure 4.9 when encountering the **`snd`** operation we add **`snd`** to the path. For `idx`, we add the index to the path. For **`zip`**, we look at the path to decide if we follow the def-use of the first or second argument: we observe **`snd`** on the path and visit the second argument. After adding another index onto the path, **`transpose`** simply flips the order of the first two indices on the path. When we hit `cast` we are leaving the RISE dialect and have reached the end of our index computation. We generate the `load` operation shown in line 8 Figure 4.7 (c) using the remaining information on the access path.

We perform a similar process for generating `store` operations using the recursive functions described in Figure 4.8b. `store` operations are generated when encountering an `assign` operation such as in line 27 Figure 4.7 (b) that generates the `store` operation in line 12 Figure 4.7 (c).

The generated `load` and `store` operations have replaced all remaining RISE operations leaving us with the final imperative target representation as shown in Figure 4.7 (c).

4.4.3 *Lowering Rise to Library Code*

One of the strengths of the pattern-based RISE IR is the ability to easily detect larger computations represented as compositions of patterns - such as matrix multiplication - without the need to perform analysis e.g. of index arithmetic. For many common computations, there exists high-performance library code that we want to leverage directly.

As an alternative to the lowering process discussed before, we might also start from our RISE MLIR dialect by detecting a composition of patterns that corresponds to a computation provided by a high-performance library. Similar to the process of lowering the `xla_hlo.dot` operation into RISE, we search for a matching composition of patterns and replace them with a MLIR `std.call` operation invoking the library interface directly. As a demonstration, we implemented a lowering of the matrix multiplication directly to the BLAS MKL library targeting Intel CPUs.

4.4.4 *Summary*

In this section, we have discussed the integration of Rise with other MLIR dialects. We have discussed how to build a practical machine learning compiler by lowering a TensorFlow graph represented in the XLA dialect to Rise and then lowering further either to loop-based representations, such as the affine dialect, or directly to library calls. Next, we are experimentally evaluating what runtime and compile time overheads we are introducing by compiling via Rise. Furthermore, we are exploring the potential of performing optimizations at the pattern-based level.

4.5 EVALUATION

4.5.1 *Experimental Setup*

We performed an experimental evaluation on an Intel Haswell quad-core i7-4790K@4.0 GHz with 64KiB L1 and 256 KiB L2 cache per core, as well as 8MiB shared L3 cache. The processor supports AVX-2 (256-bit) vector operations and has a turbo boost of 4.4 GHz. For all experiments, the frequency governor was set to "performance". Experiments were run 100 time,s and we report the median run times.

4.5.2 *Overhead of* Rise

RUNTIME OVERHEAD Figure 4.10 shows the runtime of matrix multiplication in milliseconds compiled via the Rise MLIR dialect compared to equivalently optimized versions without using Rise. We show two different sizes: a 1024x1024 matrix and a slightly more unusual size of 1x784 $\times$ 784x128, which is taken from the handwritten neural network application. The SCF version is a textbook version with three nested loops represented in the Structured Control Flow (SCF) MLIR dialect that is lowered to LLVM IR before -O3 optimizations are applied. This baseline shows a negligible runtime difference compared to the RISE naive version from Figure 4.6 lowered to the SCF dialect and then to LLVM IR as before.

Similarly, we observe no runtime overhead for the RISE opt version compared to the Affine version. This Rise version is lowered to the affine dialect as described in Section 4.4. Then both versions are optimized with polyhedral loop tiling and

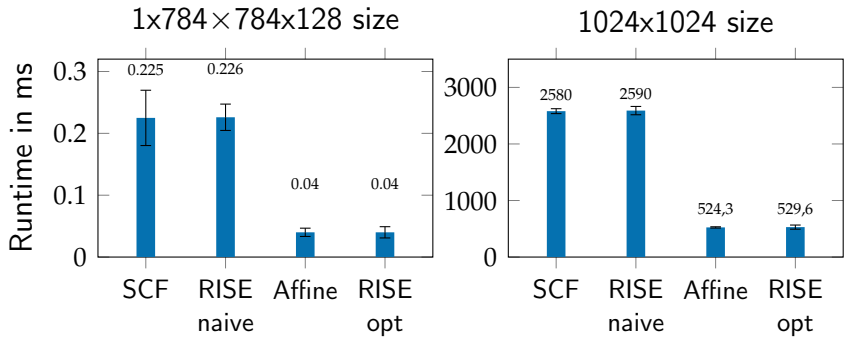


Figure 4.10: Single precision matrix multiplication runtime comparison. Rise introduces no overhead compared to equivalently optimized versions without using Rise.

vectorization passes as described in [8] before lowering to LLVM and applying `-O3` optimizations.

COMPILE TIME OVERHEAD Figure 4.11 shows compilation times of MLIR passes when compiling matrix multiplication from Rise via affine to the LLVM dialect. We report the compilation time breakdown for the median total time from 10 runs. The passes are shown in order of their execution, with all verification passes summed up at the bottom. A moderate 10% of the MLIR compilation time is spent in the lowering pass from Rise to affine described in Section 4.4.

```
          ... Compilation time report ...
===-------------------------------------------------===
--- Wall Time (s) ---  --- Name ---
   0.0002 ( 10.4%)   ConvertRiseToAffine
   0.0002 ( 10.0%)   AffineLoopTiling
   0.0003 ( 13.8%)   AffineScalarReplacement
   0.0000 (  2.4%)   AffineVectorize
   0.0001 (  7.0%)   ConvertLinalgToLoops
   0.0001 (  3.2%)   ConvertAffineToStandard
   0.0000 (  2.3%)   ConvertSCFToStandard
   0.0004 ( 21.0%)   ConvertStandardToLLVM
   0.0006 ( 29,9%)   VerificationPasses
  ------------------------
   0.0019 (100.0%)   Total
```

Figure 4.11: Breakdown of pass compilation time. Lowering of Rise only consumes about 10% of the overall time.

These results show that no runtime overhead and only a moderate compile time overhead is introduced by the functional pattern-based representation and that our lowering process produces efficient code. Furthermore, these results demonstrate the strength of integration within MLIR: starting from a functional

representation, we easily take advantage of the polyhedral optimizations, resulting in a significant performance boost. Next we explore performance gains when optimizing at the functional level.

4.5.3 *Optimizing Separate Convolutions via Rewrites*

One advantage of RISE's pattern-based representation is that optimizations are easily expressed as rewrite rules [28]. This is specifically true for algorithmic optimizations that are hard to perform at a lower loop-based level.

To demonstrate this, we consider a 2D convolution, which is an operator often used in machine learning workloads such as CNNs. A 2D convolution computes a weighted sum of a 2D neighborhood of values of an input matrix, producing an output matrix of the same size. This computational pattern is elegantly expressible in a pattern-based IR using the three patterns `pad`, `slide`, and **`map`** as discussed in [29]. Listing 4.4 shows the RISE C++ builder producing the RISE MLIR code for representing a 2D convolution: the `slide2D` pattern in line 10 creates the 2D neighbourhood that are processed by the `map2D` pattern in line 3. Each value in the neighborhood is multiplied by a weight and then summed up. This computation is expressed similarly to the dot product with the `zip2D` and **`reduceSeq`** patterns in lines 9 and 4.

CONVOLUTION SEPARABILITY A well-known optimization for 2D convolutions is to perform two 1D convolutions instead. This is based on the observation that sometimes the 2D weight matrix is decomposable, as for the Sobel filter:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

As this optimization is not universally applicable, it is not implemented in compilers, but it is recognized as beneficial in the image processing community and applied manually.

But when building specialized compiler solutions operating on domain-specific and high-level representations, we are interested in enabling the application of this optimization.

By exploiting the semantics of the high-level patterns of RISE we can define a sequence of program transformations as semantic preserving rewrite rules, as shown in [28], that separate

```
1  makeRiseProgram(out, image, weights)(
2  [&](Value image, Value weights) {
3    return mapSeq2D([&](Value slidingWindow) {
4      return reduceSeq([&](Value tuple, Value acc) {
5        return embed({fst(tuple), snd(tuple), acc},
6          [&](Value fst, Value snd, Value acc) {
7            return acc + fst * snd; });
8        }, literal(scalarF32Type(), "0.0"),
9             join(zip2D(slidingWindow, weights)));
10     }, slide2D(3, 1, pad2D(1, 1, image)));    });
```

Listing 4.4: C++ builders to generate a composition of Rise patterns modelling a 2D convolution

the convolution computation. The crucial rewrite step describes how the weighted sum, expressed as a dot product of the 2D weights (w2d) and the neighborhood, is transformed into two weighted sums of the horizontal (wH) and vertical (wV) weights:

```
rule separateConv(w2d, wV, wH) = dot(join(w2d), join(nbh)) ↦
  nbh |> transpose |> map(dot(wV)) |> dot(wH)
```

Listing 4.5 shows the resulting Rise expression for the separated convolution using the C++ builder API showing the separate computations of the vertical convolution (lines 3–11) and the horizontal convolution (lines 13–21).

```
1  makeRiseProgram(out, image, weightsH, weightsV)(
2  [&](Value image, Value weightsH, Value weightsV) {
3   Value vertical = mapSeq([&](Value arr) {
4    return mapSeq([&](Value nbh) {
5     return reduceSeq([&](Value t, Value acc) {
6      return embed(scalarF32(),{fst(t), snd(t),acc},
7       [&](Value a, Value b, Value acc) {
8        return acc + a * b; });
9     },literal(scalarF32(), "0.0"),zip(nbh, weightsV));
10   }, transpose(arr));
11  }, slide(3, 1, pad2D(1, 1, image)));
12
13  return mapSeq([&](Value arr) {
14   return mapSeq([&](Value nbh) {
15    return reduceSeq([&](Value t, Value acc) {
16     return embed(scalarF32(),{fst(t), snd(t),acc},
17      [&](Value a, Value b, Value acc) {
18       return acc + a * b; });
19    }, literal(scalarF32(), "0.0"),zip(nbh,weightsH));
20   }, slide(3, 1, arr));
21  }, vertical);                                    });
```

Listing 4.5: C++ builders to generate a composition of Rise patterns modelling a spatially separated 2D convolution

Figure 4.12 shows the performance gain of the separated over the non-separated convolution for different input sizes. We can observe a 30% performance gain of this optimization

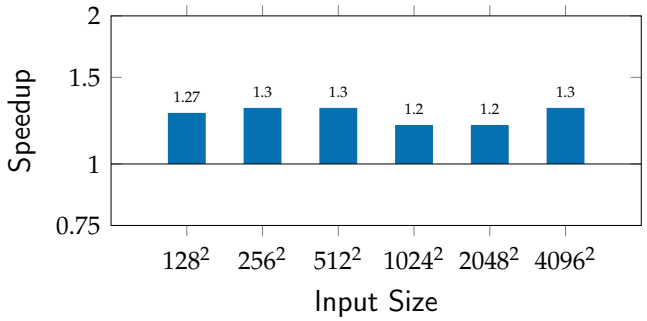


Figure 4.12: Runtime speedup of 2D spatially separated convolution over a naive 2D convolution

showing one possible simple performance gain by encoding and applying a domain-specific optimization as a sequence of rewrite rules transforming the RISE IR. We are keen to explore more opportunities such as this and contribute to improving the support for implementing such compositions of rewrites in the MLIR framework.

4.6 CONCLUSION

This chapter presented RISE – a functional pattern-based intermediate representation and its implementation in the MLIR framework. We demonstrated that integration with other intermediate representations is not only feasible but particularly effective, facilitated by MLIR's infrastructure for custom types, operators and passes.

Our evaluation showed that implementing a functional pattern-based IR within an SSA-based framework is practical and efficient when compiling to imperative loop-based code. Our principled IR design, based on lambda calculus, rewrite rules, and MLIR, provides an attractive solution for building machine learning compilers, as demonstrated through our case study of compiling TensorFlow graphs.

This work establishes pattern-based IRs as practical tools for modern compiler infrastructures, providing the first ready-to-use implementation that bridges the gap between high-level semantic representations and low-level optimizations. Our implementation is open source, and an artifact accompanying this work is publicly available.

This work directly addresses the representation challenge by providing a foundation that sets the stage for the development of principled rewriting systems within modern compiler

frameworks. In the following chapters, we will explore how to enhance MLIR with formal verification of rewrites and develop systematic approaches to compose and orchestrate transformations, building upon the pattern-based foundation established here.

5

SANE REWRITING OF HIERARCHICAL SSA

In this chapter, we address the challenge of maintaining IR integrity during compiler transformations in MLIR. We propose a practical static verification method for ensuring that transformations written in the Pattern Definition Language (PDL) maintain MLIR's hierarchical SSA properties - a fundamental requirement for compiler correctness. Our work introduces a systematic method for validating transformation patterns before they are applied, shifting verification from runtime checks to compile time. With the development of a static analysis framework and complementary fuzzing infrastructure, we demonstrate how to systematically ensure that PDL-based rewrites maintain the compiler's structural invariants, thereby preventing malformed IR and ensuring the robustness of the compilation pipeline. The development of the fuzzing infrastructure and IRDL-based verification was conducted in collaboration with Mathieu Fehr and Alexandre Lopoukhine.

5.1 INTRODUCTION

The robustness of modern compilers depends on their ability to maintain crucial invariants throughout the compilation pipeline. Static Single Assignment (SSA) form represents one of the most fundamental of these invariants, providing a foundation for reliable program analysis and transformation. In the MLIR compiler infrastructure, this challenge is particularly complex due to its hierarchical SSA form and extensible nature through dialects.

While MLIR provides runtime verification of SSA properties via dynamic checks, this approach means that structural violations are only discovered during execution, leading to bugs that might escape detection during development and testing. This delayed discovery of invariant violations makes them more difficult and costly to diagnose and fix. The Pattern Definition Language (PDL), MLIR's domain-specific language for expressing rewrite patterns, offers an opportunity for more systematic verification. However, currently, there are no guarantees

that PDL-specified transformations will preserve the compiler's structural invariants.

This chapter presents a practical approach to statically verify SSA properties in PDL-based transformations before they are executed. We develop a static analysis framework that validates critical properties such as block terminator integrity and dominance relationships. Our approach extends beyond basic SSA verification to support user-defined invariants specified through the IR Definition Language (IRDL), enabling rewrite designers to catch violations of custom structural properties early.

To demonstrate the importance of such verification, we develop a fuzzing infrastructure that generates PDL rewrites. This reveals how easily transformations can inadvertently violate SSA invariants. Our evaluation shows that static verification can catch these violations early in the development process, improving the robustness of MLIR-based compilers while reducing the reliance on runtime checks.

The techniques presented in this chapter provide compiler developers with practical tools to ensure the correctness of their transformations, contributing to the overall reliability of the compilation process. While our approach focuses on essential structural properties rather than full semantic verification, it represents an important step toward more robust compiler development in the MLIR ecosystem.

The main contributions of this chapter are:

- A static analysis framework for PDL rewrites that verifies the preservation of MLIR's hierarchical SSA properties
- An extension mechanism that enables the verification of dialect-specific invariants through IRDL specifications
- A fuzzing infrastructure for PDL rewrites that demonstrates the prevalence of invariant violations and the effectiveness of our verification approach

5.2 MOTIVATION AND BACKGROUND

Compiler transformations must maintain various structural invariants to ensure the IR remains well-formed throughout the compilation pipeline. In MLIR, these invariants become challenging to maintain due to its extensible dialect system, where custom operations need to respect fundamental properties like

SSA form. Before diving into the technical details of our verification approach, we first examine why these properties are crucial and what makes their preservation challenging.

Figure 5.1 illustrates how MLIR's unique features can create subtle pitfalls for transformation writers. In this example, we encounter an unusual MLIR capability: terminators can produce SSA values that are visible in successor blocks. The original code in Figure 5.1a uses this feature through the `my.compute_and_jump` operation, which both terminates the block and produces a value used in the successor block. While MLIR's design allows operations to serve both as value producers and block terminators, this dual role creates potential pitfalls for transformation writers. For example, a naive implementation of Dead Code Elimination (DCE) could theoretically treat a value-producing terminator like any other operation. In such a hypothetical case, when determining that `%x` is unused after eliminating the multiplication, the transformation might attempt to remove the `compute_and_jump` operation, not recognizing its critical role as a terminator. Although modern DCE implementations in MLIR prevent such cases through careful design, this example illustrates why transformation verification is essential: the result shown in Figure 5.1b would violate a fundamental MLIR invariant: every block must be properly terminated.

```
1  func.func @example(%cond: i1) -> i32 {
2    cf.cond_br %cond, ^bb1, ^bb2
3
4  ^bb1:
5    %val = func.call @get_data()
6    %x = my.compute_and_jump ^bb2(%val)
7
8  ^bb2:
9    %unused = arith.muli %x, %x
10   %y = arith.constant 42
11   return %y
12 }
```

(a) Original code with an unusual value-producing terminator, combining control and data flow

```
func.func @example(%cond: i1) {
  cf.cond_br %cond, ^bb1, ^bb2

^bb1:
  %val = func.call @get_data()
  // missing terminator!

^bb2:

  %y = arith.constant 42
  return %y
}
```

(b) After incorrect DCE transformation, violating block termination requirement

Figure 5.1: Example of how a transformation can violate MLIR's structural requirements

Current approaches rely on runtime verification, where an IR verifier checks these properties after each transformation. This means structural violations are only discovered when the specific transformation is executed on a particular input. Consequently, violations might remain undiscovered during develop-

ment and testing, only to surface later when the transformation is applied to previously unseen code patterns. Moreover, when violations are detected, developers must work backward from the broken IR to determine which aspect of their transformation pattern caused the issue. These challenges motivate our work on static verification of transformation patterns, where violations can be caught during pattern development before the transformation is even executed.

5.2.1 *Hierarchical SSA in MLIR*

Building on MLIR's hierarchical SSA structure introduced in Section 2.1.2, we focus on how this architecture impacts transformation patterns. While the basic SSA properties must be maintained, the hierarchical nature introduces additional verification challenges during pattern rewriting. Of particular importance for pattern rewriting is how operations containing regions interact with transformations. Listing 5.1 demonstrates this interaction with nested regions in a conditional operation.

```
1  func.func @nested_example(%arg0: i32) -> i32 {
2    // Region #1 (function body)
3    %0 = scf.if %cond -> i32 {
4      // Region #2 (if-true)
5      %1 = arith.addi %arg0, %arg0 : i32
6      scf.yield %1 : i32
7    } else {
8      // Region #3 (if-false)
9      %2 = arith.constant 0 : i32
10     scf.yield %2 : i32
11   }
12   return %0 : i32
13 }
```

Listing 5.1: MLIR code demonstrating hierarchical regions: a function containing an if-operation, which in turn contains two nested regions

5.2.2 *Pattern Definition Language*

MLIR's Pattern Definition Language (PDL) provides a declarative way to specify transformation patterns. Instead of implementing transformations directly in C++ or using the TableGen DSL, PDL allows patterns to be expressed as MLIR operations themselves. A PDL pattern consists of two main parts: a matcher that describes the IR structure to be transformed and a rewrite

specification that describes how to transform the matched code. Listing 5.2 shows a simple pattern demonstrating this structure.

```
pdl.pattern : benefit(1) {
  // Matcher section
  %type = pdl.type
  %input = pdl.operand
  %operation = pdl.operation "some.op"(%input) -> %type

  // Rewrite section
  pdl.rewrite %operation {
    %new_op = pdl.operation "other.op"(%input) -> %type
    pdl.replace %operation with %new_op
  }
}
```

Listing 5.2: A simple PDL pattern showing the matcher and rewriter sections, demonstrating operation replacement

PDL patterns are first lowered to the PDL interpreter dialect and then interpreted at compiler runtime. Alternatively, they can be compiled as part of the compiler. This approach offers several advantages over traditional C++ patterns or TableGen-based approaches. The interpretation-based execution enables users to add new patterns without recompiling the compiler, and the bytecode representation is significantly more compact than compiled C++ code, reducing binary size. Additionally, the interpreter can optimize pattern matching across multiple patterns by combining common match phases, making operations like attribute lookups more efficient.

The language provides operations for matching and creating operations, attributes, and types, as well as for applying constraints and performing replacements. While PDL patterns can be written directly, they are often generated from higher-level pattern descriptions or domain-specific languages, making it easier for compiler developers to express common transformation patterns.

While PDL provides a structured way to express transformations, it currently offers no guarantees about the preservation of MLIR's structural properties. Patterns that violate SSA form, break block terminator requirements, or ignore region isolation rules can be expressed and will only be caught during runtime verification. This gap between PDL's declarative pattern specification and MLIR's structural requirements motivates our work on static verification of transformation patterns.

5.2.3 *MLIR Properties*

Before we can develop effective static verification for PDL patterns, we must first understand the key properties that need to be preserved and how they can be violated. This section examines three fundamental properties of MLIR's hierarchical SSA form: block terminator integrity, dominance relationships, and region scoping rules. While PDL provides certain safety guarantees through syntactic restrictions - for instance, it lacks primitives to directly manipulate regions or block labels - some properties can still be violated via legal PDL patterns. Understanding how these properties can be violated is crucial for developing our static verification approach. We now examine each property in detail, showing how seemingly valid PDL patterns can lead to violations.

BLOCK TERMINATOR INTEGRITY Each block in MLIR must contain exactly one terminator operation, and it must be the last operation in the block. Additionally, terminators can only branch to blocks within the same region. Consider the pattern shown in Listing 5.3 that could violate this property.

```
1  pdl.pattern : benefit(1) {
2    %type = pdl.type
3    %val = pdl.operand
4    %term = pdl.operation "my.compute_and_jump"(%val) -> (%type)
5
6    pdl.rewrite %term {
7      pdl.operation "my.new_operation"()
8      // Wrong: Operation after the terminator added!
9    }
10 }
```

Listing 5.3: PDL pattern demonstrating potential terminator violation by adding operations after a terminator.

DOMINANCE As established in Section 2.1.1, MLIR enforces strict dominance requirements for value definitions over their uses. Pattern rewrites must carefully preserve these dominance relationships, particularly when transforming operations whose values are used across different blocks. Consider the PDL pattern shown in Listing 5.4.

Figure 5.2 shows both the original IR that this pattern might match (left) and the incorrectly transformed IR (right). The pattern attempts to replace both the multiplication and the addition operations but fails to recognize that `%a` is used by

```
pdl.pattern : benefit(1) {
  %type = pdl.type
  %val = pdl.operation "arith.addi" -> (%type)
  %use = pdl.operation "arith.muli"(%val) -> (%type)

  pdl.rewrite %use {
    %new_val = pdl.operation "arith.addi" -> (%type)
    %new_use = pdl.operation "arith.muli" -> (%type)
    pdl.replace %use with %new_use
    pdl.replace %val with %new_val
  }
}
```

Listing 5.4: PDL pattern that violates dominance.

the terminator `cond_br`. The rewrite violates the dominance property: in the transformed IR, `%a'` is used by the branch operation before it is defined. This example demonstrates how PDL patterns must carefully consider all uses of a value, possibly in different blocks, to maintain proper dominance relationships.

```
^bb0(%x: i1, %y: i1, %c):
  %a = andi %x, %y : i1
  cond_br %a, ^bb1, ^bb2

^bb1:

  %b = muli %a, %c : i1
  br ^bb3

^bb2:
  ...
```

(a) Original code with proper SSA dominance.

```
^bb0(%x: i1, %y: i1):

  cond_br %a', ^bb1, ^bb2

^bb1:
  %a' = addi %x, %y : i1
  %b' = muli %a', %c : i32
  br ^bb3

^bb2:
  ...
```

(b) After pattern application, showing the dominance violation.

Figure 5.2: Demonstration of a dominance violation introduced by pattern the pattern in Listing 5.4, where `cond_br` uses `%a'` before its definition.

REGION SCOPING MLIR regions establish strict value visibility boundaries. Values defined in a region are only visible within that region, and regions marked as isolated_from_above cannot access values from surrounding regions unless explicitly passed as arguments. Operations like func.func and gpu.launch use isolated regions to enforce clean separation of value scope.

PDL's design prevents violations of region isolation properties by construction through several key limitations:

- PDL cannot create new operations that contain regions. This means transformations cannot incorrectly structure

nested regions or accidentally create invalid region hierarchies.

- PDL cannot modify existing region structures. Operations containing regions can only be matched and replaced as a whole, preventing transformations from inadvertently breaking region boundaries or modifying region contents in ways that would violate scoping rules.
- Value visibility across region boundaries is handled implicitly by PDL's matching semantics. When a pattern matches operations, it can only access values that are legally visible according to MLIR's scoping rules. There is no mechanism in PDL to circumvent these visibility restrictions during pattern application.

These design choices make PDL patterns inherently "region-safe" - transformations that would violate MLIR's region scoping rules cannot be expressed. While this guarantee of region safety is valuable, it comes at the cost of expressiveness: PDL cannot represent transformations that require manipulation of region structures.

Similarly, PDL's design restricts how transformations can modify control flow by not providing primitives to match or emit block labels. Terminators cannot be constructed with references to specific blocks, which means PDL patterns cannot arbitrarily restructure the control flow graph. This limitation prevents certain classes of dominance violations, as patterns cannot modify block reachability in ways that would break the dominance relationship between blocks.

These structural restrictions in PDL represent a deliberate trade-off: while they limit the scope of possible transformations, they make the permitted transformations less error-prone and easier to verify. However, as we saw in our discussion of dominance violations, these built-in safeguards are not comprehensive.

5.3 STATIC VERIFICATION OF SSA PROPERTIES

Algorithm 1 presents our static verification algorithm, which employs abstract interpretation of PDL patterns to ensure the preservation of MLIR's SSA properties.

Algorithm 1 PDL Static Analysis

Require: PDL Pattern P

```
function AnalyzePDLPattern(P)
    state ← InitializeAnalysisState()
    // Phase 1: Initialization
    InitializePattern(P, state)
    // Phase 2: Pattern Matching
    AnalyzeMatching(P, state)
    // Phase 3: Rewrite Simulation with Invariant Checking
    SimulateRewrite(P, state)
    return P                    // Now annotated with any detected violations
function AnalyzeMatching(P, state)
    visited ← ∅
    VisitMatchedOp(P.root_op, visited)
    if visited ≠ P.matching_ops then
        raise "Pattern not connected"
function VisitMatchedOp(op, visited)
    if op ∈ visited then return
    visited ← visited ∪ {op}
    for operand ∈ op.operands do
        // operand is payload abstraction of operand, attribute or type
        VisitMatchedOp(operand.definingOp, visited)
function SimulateRewrite(P, state)
    for op ∈ TextualOrder(P.rewrite_body) do
        InterpretOperation(op, state)
        violations ← ∅
        violations ← violations ∪ CheckTerminators(op, state)
        violations ← violations ∪ CheckUseDef(op, state)
        annotate(op, violations)
function InterpretOperation(op, state)
    if op is EraseOp then
        if IsTerminator(op.operand) then
            raise "Cannot erase terminator"
        RemoveOp(op.operand, state)
    else if op is ReplaceOp then
        ValidateTerminatorReplacement(op.replaced,
        op.replacement)
        ValidateResultCount(op.replaced, op.replacement)
        UpdateUsesAndScope(op.replaced, op.replacement, state)
    else if op is OperationOp then
        ValidateRootUsage(op, state)
        ValidateInsertionPoint(state)
        AddNewOp(op, state)
```

The algorithm operates in three distinct phases through the functions InitializePattern, AnalyzeMatching, and SimulateRewrite, each building upon a common abstract state.

During Phase 1, InitializePattern constructs payload abstractions for all PDL operations in the matching section. These abstractions capture essential properties such as use-def relationships and whether operations might match terminators. The initialization phase builds the foundation for subsequent analysis by creating a complete mapping between PDL operations and their abstract representations.

In Phase 2, AnalyzeMatching (line 10) verifies pattern connectivity through the VisitMatchedOp function. Starting from the root operation, the algorithm traverses the pattern following def-use relationships, marking each visited operation. As shown in Algorithm 1, the traversal must reach every operation in the pattern's matching section - if any operation remains unvisited, the pattern is invalid as it contains disconnected components. This connectivity check ensures that all operations in the pattern are reachable and participate in the transformation.

During Phase 3, SimulateRewrite (line 21) processes rewrite operations from top to bottom while continuously validating SSA properties. As shown in Algorithm 1, this phase implements verification through specialized interpretation rules for each PDL operation type in the InterpretOperation function. The algorithm shows the three operations that directly modify IR structure—EraseOp, ReplaceOp, and OperationOp—as these require careful verification of SSA properties through CheckTerminators and CheckUseDef.

For each operation, InterpretOperation (line 28) simultaneously models the transformation's effects and verifies the preservation of SSA properties. For example, when processing an EraseOp, it first validates that the operation is not a terminator before modeling its erasure in the state. Similarly, ReplaceOp verification includes checking terminator replacement validity and result count consistency through ValidateTerminatorReplacement and ValidateResultCount. This integrated approach allows us to detect violations at the specific PDL operation that would introduce them, providing precise feedback about which operation would introduce the violation.

The analysis maintains a state that enables continuous verification of structural properties and precise error reporting, consisting of:

- The current transformation phase
- A mapping from PDL operations to payload abstractions (abstract representations of the operations they match in the payload IR)
- A graph of use-def relationships between payload abstractions
- A graph approximating the order of payload operations

The mapping between PDL operations and their payload abstractions is crucial for the analysis as it enables us to track the properties of the operations that will be transformed eventually. For example, when a PDL operation might match a terminator in the payload IR, its payload abstraction reflects this property, and we must ensure that any transformation involving this operation preserves terminator semantics. Similarly, tracking use-def relationships between payload abstractions allows us to verify dominance properties based on the actual structure of the IR being rewritten rather than just the relationships expressed in the PDL pattern.

PDL's matching style specifies def-use relationships between operations without any notion of ordering, making the exact sequence of matched operations in the payload IR unknown during analysis. We approximate their order through def-use relationships yielding a partial ordering. In contrast, operations created during the rewrite phase have a deterministic ordering: they are inserted after the root payload operation in the order of their appearance in the PDL pattern. This approach allows for detecting invalid transformations due to an invalid insertion point. For example, if a pattern erases its root operation before generating new operations, our analysis detects that no valid insertion point exists for the subsequent operations and flags the pattern as invalid.

5.3.1 *Analysis Example*

To illustrate how our verification algorithm detects SSA violations, consider the PDL pattern in Listing 5.5 that attempts to transform $(x * y) * z \rightarrow z * (y * x)$. The pattern models this transformation as two separate commutations: first commutating z with $(x * y)$ in the outer multiplication, then commutating x and y in the inner multiplication. This transformation preserves the

mathematical semantics due to multiplication's commutativity. However, this incorrect pattern's implementation leads to an SSA violation.

```
1  pdl.pattern : benefit(1) {
2    %type = pdl.type
3    %x = pdl.operand
4    %y = pdl.operand
5    %z = pdl.operand
6    %val = pdl.operation "mul"(%x, %y) -> (%type)
7    %res = pdl.operation "mul"(%val, %z) -> (%type)
8    pdl.rewrite %res {
9      %new_res = pdl.operation "mul"(%z, %val) -> (%type)
10     %new_val = pdl.operation "mul"(%y, %x) -> (%type)
11     pdl.replace %val with %new_val
12     pdl.replace %res with %new_res
13   }
14 }
```

Listing 5.5: PDL pattern attempting to commute a nested multiplication. The pattern incorrectly tries to use a value after replacing it, violating SSA properties.

To understand how our algorithm detects this, let's follow it step-by-step.

During the initialization phase, our analysis creates payload abstractions for both multiplication operations and their operands. The def-use relations between these operations are recorded, and both matched `mul` operations are marked as potentially having external uses. The algorithm identifies both multiplication operations as non-terminators based on the operation definitions.

In the pattern matching phase, our analysis starts at the operation `pdl.rewrite` (line 8), which identifies the second `pdl.operation` matching for `mul`, the producer of `%res`, as the root operation for the transformation. Starting from this root operation, it follows def-use relationships to traverse the pattern, first reaching its type `%type` and its operands `%val` and `%z`. Following `%val`, it reaches the match operation for the first `mul` and its operands `%x` and `%y`. During this traversal, each visited operation is marked as participating in the match. The analysis then confirms that all operations, including `pdl.type` and the three `pdl.operand` operations, have been marked, verifying that the pattern forms a connected component with no unused operations.

In the rewrite simulation phase, the `pdl.rewrite` is interpreted from top (line 9) to bottom (line 12), knowing that new operations are inserted before the root operation.

LINE 9 Create new `mul` operation reusing the original `%val` and modeling the first commutation: $(x * y) * z \rightarrow z * (x * y)$

LINE 10 Create new `mul` operation modeling the second commutation: $z * (x * y) \rightarrow z * (y * x)$

LINE 11 Replace all uses of the original computation with the rewritten one. This is safe because `%new_res` was emitted before the root operation and thus dominates all possible uses.

LINE 12 Attempt to replace the old inner multiplication with the commuted version. Our analysis detects that this is unsafe as `%val` might have other possible uses before the root operation that are dominated by `%new_val`. In this case, the rewrite will always fail because the first newly emitted `mul` operation is not dominated.

The analysis correctly identifies a violation of SSA's dominance requirement. The issue arises because the pattern first creates a new multiplication operation that uses `%val` but then attempts to replace `%val` with `%new_val`. Since `%new_val` is created after the first multiplication, it cannot dominate its use, violating SSA's dominance property.

This problem cannot be safely fixed by reordering operations as long as we try to replace `%val`. The original $(x * y)$ might have uses outside the pattern that appear before the root operation, and any attempt to replace it could violate dominance. The correct solution is to avoid replacing `%val` entirely, as shown in Listing 5.6.

```
1 pdl.pattern : benefit(1) {
2   // match section unchanged
3   pdl.rewrite %res {
4     %new_val = pdl.operation "mul"(%y, %x) -> (%type)
5     %new_res = pdl.operation "mul"(%z, %new_val) -> (%type)
6     pdl.replace %res with %new_res
7   }
8 }
```

Listing 5.6: Corrected PDL pattern for multiplication commutation. Instead of attempting to replace the inner multiplication, which could violate dominance, the pattern creates a new computation chain and relies on subsequent cleanup.

This version creates the new operations in the correct order and only replaces the root operation. If `%val` becomes unused after this transformation, it can be cleaned up by a subsequent

dead code elimination pass. This example highlights fundamental limitations in PDL's rewriting capabilities: while PDL provides a safe, declarative way to specify simple transformations, it lacks the flexibility needed for complex SSA rewrites. Operations can only be replaced as a whole, and there's no mechanism to reason about or enforce dominance relationships across pattern boundaries. As a result, transformation authors must often split complex rewrites into multiple simpler patterns or rely on subsequent cleanup passes, trading transformation efficiency for safety and verifiability.

5.4 BEYOND SSA PROPERTIES: VERIFYING DIALECT-SPECIFIC CONSTRAINTS

While the previously discussed SSA properties form the fundamental structure of all MLIR programs, each dialect typically defines additional constraints specific to its domain. These dialect-specific constraints operate alongside SSA properties and do not replace them. For instance, while SSA ensures proper value definition and use, a tensor dialect might additionally require specific rank relationships between operands, or a hardware dialect might enforce specific alignment requirements.

Traditionally, dialect authors implement these constraints in C++ verification functions. Recently, MLIR introduced the IR Definition Language (IRDL), described in Section 2.1.2, which provides a declarative way to specify common types of constraints. This makes some constraints more maintainable and, crucially, lays the foundation for automated analysis of constraint satisfaction during program transformation.

Dialect constraints can be categorized into two distinct types:

Structured constraints represent commonly occurring constraints that can be expressed declaratively through IRDL. These typically include:

- Constraints on operand and result types
- Relationships between types of different operands or results
- Requirements on attribute values and their types
- Basic structural requirements (e.g., number of regions or blocks)

Custom constraints encompass verification logic that cannot be expressed through IRDL's declarative syntax. These are implemented as arbitrary C++ verification functions using IRDL-C++ and may include:

- Complex structural analysis (e.g., graph connectivity checks)
- Cross-operation invariants
- Context-dependent validation

The distinction between structured and custom constraints is crucial: while structured constraints can easily be analyzed statically and verified automatically during pattern matching and rewriting, custom constraints require runtime checking and cannot be easily reasoned about during transformation. Previous work on IRDL has shown that 97% of MLIR's local operation constraints can be expressed using structured constraints, with only 30% of operations requiring additional C++ verification for non-local constraints [21]. This high coverage demonstrates that an automated verification system for structured constraints would have a significant impact, as it would enable compile-time verification for the majority of MLIR's operations.

5.4.1 *Breaking Constraints During Pattern Rewriting*

Pattern-based rewriting systems like PDL are powerful tools for implementing program transformations. However, they currently provide no compile-time guarantees that the generated code will satisfy all dialect constraints. Even when the input program is valid, and the pattern seems correct, the rewritten program might violate structural invariants of the target dialect.

To illustrate this problem, let's examine a bug that existed in MLIR's core infrastructure. Figure 5.3a shows a transformation in the canonicalization pass that attempts to simplify extended multiplication when one operand is the constant 1. The resulting transformation is shown in Figure 5.3b.

This transformation appears reasonable at first glance: when multiplying by one, the high bits should only depend on the sign of the input. The transformation is actually correct for fixed-width integer types like `i32`, where both operations are well-defined, and the sign extension preserves the mathematical properties of the multiplication. However, the rewrite violates a structural invariant: while `arith.mulsi_extended` supports the index type, the newly generated `arith.extsi` operation does

```
%one        = arith.constant 1 : index
%low, %high = arith.mulsi_extended %x, %one
```

(a) Valid input program

```
%zero   = arith.constant 0 : index
%is_neg = arith.cmpi slt %x, %zero : index
%high   = arith.extsi %is_neg : i1 to index
// %low is replaced by %x
```

(b) Invalid transformation result

Figure 5.3: Example of an invalid transformation that violates type constraints

not. This type incompatibility renders the rewrite invalid for index-typed inputs.

This real-world example demonstrates why compile-time verification of dialect constraints is essential: even experienced compiler developers can inadvertently write patterns that violate basic type constraints, and these violations might only be discovered during runtime verification or, worse, lead to invalid programs if verification is disabled.

5.4.2 *Structured invariant analysis using IRDL*

While this example exposes a serious problem in pattern verification, it also points to a solution: the violation stems from type constraints that are already formally specified in IRDL. This observation leads to our key insight: we can leverage existing IRDL specifications of dialects to verify pattern rewrites at compile time.

Our approach consists of three main steps:

1. Abstract interpretation of the pattern to derive the program structure after pattern application
2. Transform both programs into IRDL constraints using IRDL specifications of the participating operations
3. Verify via SMT solving that the rewritten program cannot violate any structural constraints

This verification happens entirely at compile time, when the pattern is defined, rather than during pattern application. A successful verification guarantees that the pattern will never generate operations that violate their IRDL-specified constraints. Conversely, if verification fails, the analysis provides a concrete example that triggers the problematic behavior, enabling pattern authors to fix issues before deployment.

For our analysis, we need precise specifications of the operations' structural constraints. Figure 5.4 shows the IRDL specifications for both operations involved in our example transformation. IRDL uses an SSA-based representation to specify dialects: type variables like `%t` represent constraints, and IRDL operations define relationships between these variables. As shown in Figure 5.4a, the specification for `arith.mulsi_extended` captures that it accepts both index and integer types for its operands and results, requiring all of them to be of the same type. This is expressed through the `any_of` constraint, which allows either an index or integer type, with the type variable `%t` ensuring all operands and results share the same type choice. In contrast, Figure 5.4b shows that `arith.extsi` is more restrictive: it only accepts integer types, not index types.

```
1 irdl.operation @mulsi_extended {
2     %id = irdl.base @index
3     %int = irdl.base
      @integer_type
4     %t = irdl.any_of(%idx, %int)
5     irdl.operands(%t, %t)
6     irdl.results(%t, %t)
7 }
```

(a) `arith.mulsi_extended` supports both index and integer types

```
irdl.operation @extsi {
    %int = irdl.base
    @integer_type
    irdl.operands(%int)
    irdl.results(%int)
}
```

(b) `arith.extsi` is restricted to integer types

Figure 5.4: IRDL specifications showing the type constraint mismatch between operations

This fundamental difference in type constraints, clearly visible in their IRDL specifications, explains why our example transformation fails: while `arith.mulsi_extended` can operate on index types, arith.extsi cannot, rendering the rewrite invalid for index-typed inputs.

With these IRDL specifications, we can verify whether a pattern rewrite will preserve all structural constraints. Our analysis works directly on PDL patterns, which express both the matching and rewriting steps.

The PDL pattern, shown in Listing 5.7, models the complete transformation: matching a multiplication by one and replacing it with a sign test and extension operation.

PDL patterns express both matching criteria and rewrite steps. To verify that a rewrite preserves all structural constraints, we transform this pattern into a constraint satisfaction problem. Our analysis determines whether the rewrite might break structured

invariants for any valid input program and, if so, provides a concrete example that demonstrates the violation.

The verification process consists of two main phases:

1. Transform the PDL pattern into an IRDL constraint query
2. Solve this query using SMT to verify constraint preservation

```
%t = pdl.type
%x = pdl.operand : %t
%one = pdl.attribute : %t
pdl.apply_native_constraint "is_one"(%one)
%one_op = pdl.operation "arith.constant" {"value" = %one} -> (%t)
%one_val = pdl.result 0 of %one_op

%root = pdl.operation "arith.mulsi_extended" (%x,%one_val) -> (%t,%t)
pdl.rewrite %root {
  %zero = pdl.apply_native_rewrite "get_zero"(%t) : !pdl.attribute
  %zero_op = pdl.operation "arith.constant" {"value" = %zero} -> (%t)
  %zero_val = pdl.result 0 of %zero_op

  %two = pdl.attribute = 2 : i64
  %i1 = pdl.type : i1
  %cmpi_op = pdl.operation "arith.cmpi" (%x, %zero_val)
                                        {"predicate" = %two} -> (%i1)
  %cmpi_val = pdl.result 0 of %cmpi_op

  %extsi_op = pdl.operation "arith.extsi"(%cmpi_val) -> (%t)
  %extsi_val = pdl.result 0 of %extsi_op

  pdl.replace %root with (%x, %extsi_val)
}
```

Listing 5.7: PDL pattern for transforming multiplication by one into a sign test and extension. This pattern demonstrates the type constraint violation between `arith.mulsi_extended` and `arith.extsi`.

5.4.3 *From PDL to IRDL Verification*

The first phase decomposes the PDL pattern into two parts that capture the complete rewrite:

- A matching part that describes all possible input programs that could match the pattern
- A rewritten part that represents the final state of our abstract interpretation, describing all possible output programs

To create the rewritten part, we extend the abstract interpretation infrastructure from our SSA verification with a component that emits the final state as a PDL matcher. This representation of the rewritten program structure serves as input for our constraint verification.

We express this verification problem by introducing the IRDL `check_subset` operation. This operation takes two regions containing the constraints from the matching and rewritten programs, respectively. The `irdl.yield` operations in both regions make explicit which values must maintain consistency between the original and rewritten programs. If all programs produced by the rewrite satisfy their structural constraints, then the constraints from the second region must be satisfied whenever the constraints from the first region are satisfied. Listing 5.8 shows the pattern lowered to this representation:

```
irdl.check_subset {
  %t = pdl.type
  %x = pdl.operand : %t
  %one = pdl.operation "arith.constant" {value = 1} -> (%t)
  %root = pdl.operation "arith.mulsi_extended"(%x,%one[0]) -> (%t,%t)
  irdl.yield %x, %one[0], root[0], root[1]
} of {
  %t = pdl.type
  %x = pdl.operand : %t
  %one = pdl.operation "arith.constant" {value = 1} -> (%t)
  %zero = pdl.operation "arith.constant" {value = 0} -> (%t)

  %cmpi = pdl.operation "arith.cmpi"(%x, %zero[0])
                                    {predicate = 2} -> (i1)
  %extsi = pdl.operation "arith.extsi"(%cmpi[0]) -> (%t)
  irdl.yield %x, %one[0], %x, %extsi[0]
}
```

Listing 5.8: Constraint satisfaction problem derived from the PDL pattern. The `irdl.check_subset` operation compares type constraints before and after the transformation, enabling verification of type consistency.

Let's examine this lowered representation of our example in detail. The first region captures all constraints from the matching part: it declares a type variable `%t` and establishes the `arith.mulsi_extended` operation with its requirements that both operands and results must be of type `%t`.

The second region reflects the program structure after the rewrite has been applied. It contains all operations created during the rewrite: the zero constant, the comparison operation that produces an `i1` result, and the `arith.extsi` operation. The

`irdl.yield` operations at the end of each region define the replacement relationship between the original and rewritten programs. In our case, it shows that:

- The first result of `arith.mulsi_extended` (`%root[0]`) is replaced with the original `%x`
- The second result (`%root[0]`) is replaced with the result of `arith.extsi`
- The operand `%x` and the constant one value remain unchanged

This representation makes the type violation explicit: while `%t` in the first region can be index, the second region attempts to use `arith.extsi` with this same type `%t`, which we know from its IRDL specification is only valid for integer types.

The next step transforms all PDL operations into pure IRDL constraints. While most PDL constraints map directly to IRDL equivalents, operations require special handling. When translating a `pdl.operation`, we incorporate the constraints from its corresponding IRDL specification to capture all relationships between operands and results. For example, when we encounter the `pdl.operation arith.mulsi_extended`, we include all constraints from its IRDL definition shown in Figure 5.4.

We apply this same principle to attributes and types. Type constraints are translated to an `irdl.parametric` operation, capturing not only the type constructor but also constraints on its parameters. This is crucial because MLIR types are parametric: for instance, an integer type has both a width parameter and a signedness parameter that must be verified.

After this translation, we apply custom simplification rewrites to reduce the query's complexity. For our example, this results in the compact representation shown in Listing 5.9:

In this simplified form, we can see how the analysis has distilled our verification problem to its essence. The left side of the `check_subset` shows four identical type variables that must be either index or integer types - these correspond to the two operands and two results of `arith.mulsi_extended`. The right side requires all four corresponding types to be integer types, reflecting the constraints from our generated operations, particularly `arith.extsi`, which only accepts integers.

This simplified query precisely captures our original type violation: can we find types that satisfy the left side (allowing index) but violate the right side (requiring integer)? Indeed,

```
irdl.check_subset {
  %index = irdl.base @index
  %bitwidth = irdl.base #int
  %signedness = irdl.base #signedness
  %integer = irdl.parametric @integer_type<%bitwidth, %signedness>
  %t = irdl.any_of %index, %integer
  irdl.yield %t, %t, %t, %t
} of {
  %bitwidth = irdl.base #int
  %signedness = irdl.base #signedness
  %integer = irdl.parametric @integer_type<%bitwidth, %signedness>
  irdl.yield %integer, %integer, %integer, %integer
}
```

Listing 5.9: Simplified IRDL representation of the constraint checking problem. The left side allows for index or integer types, while the right side requires all types to be integers.

when all types are index, the original program is valid, but the rewritten program violates the constraints of `arith.extsi`. This reduction of our problem makes it amenable to automated verification through SMT solving.

5.4.4 *SMT-Based Verification*

To automatically verify our IRDL constraints, we utilize xDSL's existing SMT translation infrastructure [70] for MLIR types and attributes. This translation represents types and attributes as an SMT datatype with constructors for built-in types and their parameters.

The translation maps each IRDL constraint to SMT formulas. For instance, irdl.base creates a constraint requiring a specific constructor, while irdl.parametric constrains the parameters of that constructor.

Our goal is to verify that any types satisfying the constraints of the matching pattern will also satisfy the constraints after the rewrite. For our check_subset query, we translate both regions into sets of variables and constraints (V_{lhs}, V_{rhs}, C_{lhs}, C_{rhs}) with additional equality constraints Eq from the irdl.yield operations. We then query the SMT solver to find any counterexample where the matching constraints are satisfied, but the rewrite constraints are not:

$$\forall V_{lhs}, (C_{lhs}(V_{lhs}) \rightarrow \neg \exists V_{rhs}, (Eq(V_{lhs}, V_{rhs}) \rightarrow C_{rhs}(V_{rhs})))$$

An unsat result proves no such counterexample exists, meaning the rewrite preserves all structural constraints. A sat result

provides a concrete case where the rewrite would violate constraints. In our example, the solver returns sat with `%t` assigned to the `index` type, confirming our rewrite incorrectly attempts to create a `arith.extsi` operation with `index` type.

Our verification approach enables static detection of structural constraint violations in PDL pattern rewrites. While we demonstrated this through a type compatibility bug in MLIR's canonicalization patterns, the approach applies to any structural constraint that can be expressed in IRDL. This includes verifying proper operand counts, result type relationships, and attribute value constraints across dialect operations.

By performing this verification when patterns are compiled, we catch constraint violations before patterns are deployed in a compiler pipeline. This shifts the detection of structural constraint violations from runtime verification to compile time, providing pattern authors with immediate feedback about potential violations.

5.5 EVALUATION

Evaluating our verification approach for PDL patterns requires a careful consideration of available test cases. While PDL is a domain-specific language designed for specifying MLIR rewrites, most production compilers using MLIR still define their rewrites directly in C++ or using TableGen. To provide a comprehensive evaluation despite this limited availability of real-world patterns, we follow a two-pronged approach: First, we use systematic fuzzing to generate and verify a large corpus of patterns, providing broad coverage of potential pattern structures and transformation types. Then, we analyze existing PDL patterns from production compilers to validate our approach on real-world transformations and demonstrate its practical relevance.

5.6 FUZZING PDL REWRITES

To evaluate our static pattern verification and MLIR's runtime verification behavior systematically, we aim to quantify how often valid patterns lead to dynamically detected failures. Our framework consists of two complementary fuzzers working in tandem. The first fuzzer generates syntactically valid PDL rewrite patterns that pass MLIR's static verification. The second fuzzer then generates valid MLIR input programs to be matched by these patterns, allowing us to verify the pattern's behavior.

PATTERN GENERATION The pattern fuzzer generates PDL rewrites by constructing valid operation graphs to match and corresponding transformations to operate on them. To ensure coverage of potential error cases, we focus on generating patterns that exercise different aspects of MLIR's rewriting capabilities.

The fuzzer constructs patterns in two phases: match-side generation and rewrite-side generation. For the match side, the fuzzer randomly generates a directed acyclic graph of matchers using `pdl.operation`, where each matched operation has a randomly chosen number of operands and results within configurable bounds. Operation matchers are predominantly generated to match regular named operations, with a small probability of generating wildcard matches or terminators.

The rewrite side of the pattern is generated via a sequence of randomly selected transformations. For each transformation, we choose one of four types and select appropriate values and operations from the match side:

1. One-to-one replacement - substituting one operation with another
2. Many-to-one replacement - replacing an operation with multiple values
3. Operation erasure - removing an operation from the IR
4. Operation creation - introducing new operations with connections to existing values

The number of transformations in each pattern is randomly determined within configured bounds, allowing us to test both simple rewrites and more complex chains of transformations.

Building on our fuzzer's pattern generation approach, let's examine a concrete example of the patterns our fuzzer generates. The following listing shows a pattern that combines multiple transformation types:

While the pattern is syntactically valid PDL and passes MLIR's verification, it contains a critical flaw that manifests only during pattern application. The rewrite first replaces operation `%op2 with a new operation %op2`, effectively erasing `%op2` from the IR. However, it then attempts to use the result `%op2` of the erased operation as an operand in the creation of `%new`. This invalid use of an erased value will cause a runtime failure, despite passing the existing PDL verification during parsing.

```
pdl.pattern : benefit(1) {
  // Match-side generation
  %type = pdl.type
  %arg = pdl.operand
  %op1 = pdl.operation "test.op"(%arg) -> %type
  %op2 = pdl.operation "test.op"(%op1) -> %type

  // Rewrite-side generation (replacement case)
  pdl.rewrite %op2 {
    %op2' = pdl.operation "test.op"(%op1) -> %type
    pdl.replace %op2 with %op2'
    %new = pdl.operation "test.op"(%op2) -> %type
  }
}
```

Listing 5.10: Generated PDL pattern demonstrating a rewrite that passes static verification but contains an invalid use of an erased value. The pattern matches two chained operations and attempts to use %op2 after replacing it.

Such patterns highlight the gap between static and dynamic verification, which our fuzzing approach aims to uncover. The fuzzer frequently generates patterns that appear valid at first glance but contain subtle violations of SSA properties that only become apparent during pattern application. While this particular pattern will fail for any matched input due to its inherent invalid value use, other generated patterns may only fail for specific program structures. For instance, some patterns might only violate IR properties when terminator operations are involved in the match, or specific dominance relationships between the matched operations are present. Simply generating a single matching program is insufficient to expose such conditional failures. This observation motivates the second phase of our approach: the systematic generation of matching input programs that explore different valid arrangements of operations matching the pattern's structure.

5.6.0.1 *Generating inputs to a rewrite*

Generating suitable input programs for a PDL pattern is non-trivial - they must not only be valid MLIR but also match the structural constraints specified in the pattern's match section. Our input generation fuzzer analyzes a given PDL pattern and creates valid MLIR programs through a three-phase process.

First, the fuzzer converts the PDL pattern's match section into a set of concrete MLIR operations. During this translation phase, it maintains a synthesis context that tracks types, values, and operations. It processes each PDL operation sequentially,

creating corresponding MLIR constructs. For example, `pdl.type` operations are materialized as concrete types (defaulting to `i32` in our implementation), while `pdl.operand` operations either reuse existing values or create new block arguments.

After translation, the generator analyzes the operation graph to identify connected components and dependency relationships. It uses a Union-Find data structure to efficiently partition operations into connected components based on their use-def chains. This analysis enables the final phase, where the fuzzer generates different variations of matching programs by exploring valid operation orderings while maintaining MLIR's well-formedness properties.

To generate these variations systematically, the fuzzer processes operations differently based on their connectivity. When handling multiple connected components, the generator creates separate blocks for each component and adds appropriate terminators to maintain control flow. For single components, it identifies and places root operations (those not used by any other operation in the component) before recursively processing the remaining operations. By processing operations in dependency order while exploring different valid placements, this approach ensures that all generated programs maintain proper SSA form.

The result is a systematic exploration of possible matching programs that can help identify potential issues in the PDL rewrite pattern. For example, given a pattern that matches multiple operations, the generator produces programs with different operation orderings and block placements, which is crucial for catching rewrites that only fail under specific structural conditions.

For earlier pattern example of Listing 5.10, the following two programs represent different valid inputs for matching of two `test.op` operations with a *use* relationship.

The combination of pattern and input fuzzing revealed numerous cases where PDL rewrites pass static verification but fail during application.

5.6.1 *Empirical Validation of Analysis Accuracy*

Using our fuzzing infrastructure, we generated a corpus of 10,000 syntactically valid PDL patterns to evaluate the accuracy of our static analysis approach. For each pattern, we compared the predictions of our static analysis against actual runtime

```
// Input 1: Operations in same block
func.func @test(i32) -> i32 {
  ^bb0(%arg0: i32):
  %0 = test.op(%arg0) : (i32) -> i32
  %1 = test.op(%0) : (i32) -> i32
  test.terminator()
}

// Input 2: Operations in separate blocks
func.func @test(i32) -> i32 {
  ^bb0(%arg0: i32):
  %0 = test.op(%arg0) : (i32) -> i32
  test.br ^bb1

  ^bb1:
  %1 = test.op(%0) : (i32) -> i32
  test.terminator()
}
```

behavior observed with generated test inputs. This systematic comparison allows us to measure both false positives (patterns flagged as problematic that execute successfully) and false negatives (patterns that pass analysis but fail during execution).

Of the 10,000 generated patterns, 87.8% (8,782 patterns) represented cases where MLIR's existing verification would accept the pattern, but our enhanced static analysis identified potential issues. This high percentage underscores the significant gap in MLIR's current verification capabilities. Table 5.1 shows the classification of results based on the correspondence between static analysis predictions and observed runtime behavior.

		Runtime Check	
		Pass	Fail
Static Analysis	Pass	1049	5[1]
	Fail	164	8782

Table 5.1: Classification of Static Analysis Results on Generated PDL Patterns

The results show that our analysis correctly identified 8,782 problematic patterns (true positives) where generated inputs triggered runtime failures and verified 1,049 patterns as safe (true negatives) across extensive input testing. In 164 cases, our static analysis conservatively flagged patterns as problematic that executed successfully in practice. This conservative ap-

1 After fixing the discovered MLIR bugs, this number dropped to 0, eliminating all false negatives.

proach is by design - our analysis deliberately overestimates potential issues in cases where a precise static determination is impossible, preferring false positives over potentially missing critical violations.

```
pdl.pattern : benefit(1) {
  %0 = pdl.type : i32
  %1 = pdl.operand : %0
  %2 = pdl.type : i32
  %3 = pdl.operand : %2
  %4 = pdl.type : i32
  %5 = pdl.operation "pdltest.matchop" (%1, %3 : !pdl.value, !
    pdl.value) -> (%4 : !pdl.type)
  %6 = pdl.result 0 of %5
  pdl.rewrite %5 {
    %7 = pdl.type : i32
    %8 = pdl.operation "pdltest.rewriteop" -> (%7 : !pdl.type)
    %9 = pdl.result 0 of %8
    pdl.replace %5 with (%9 : !pdl.value)
    %10 = pdl.operation "pdltest.rewriteop"
    // ^error: expected operation name in quotes
               %10 = pdl.operation "pdltest.rewriteop"!
  }
}
```

Listing 5.11: PDL pattern demonstrating the MLIR parsing bug in presence of pdl.replace

The discovery of 5 false negatives was particularly significant, as it revealed fundamental bugs in MLIR's infrastructure rather than limitations in our analysis. These cases, where patterns passed our static verification but failed at runtime, led to the identification and subsequent fixing of two critical bugs in MLIR's core infrastructure. The first bug manifested in the PDL parser when handling value replacements in pdl.replace operations: if a value was replaced with another value, the parser incorrectly processed the operation following the replacement. Listing 5.11 demonstrates this issue, where after the pdl.replace operation, the parser fails to correctly process the subsequent pdl.operation, resulting in a syntax error despite the pattern being semantically valid.

The second bug involved the creation of new operations under specific conditions, where the infrastructure failed to properly validate all necessary preconditions, potentially resulting in malformed IR. Both bugs were particularly subtle as they only manifested under specific combinations of pattern structures that our fuzzing framework was able to generate. After addressing the discovered MLIR bugs, our analysis maintains a conservative false positive rate of only 1.64% and achieves a zero false negative rate, which is crucial for compiler developers to

confidently rely on our analysis when verifying pattern-based transformations. These findings demonstrate the effectiveness of our combined static-dynamic approach in uncovering deep implementation issues that affect the correctness of pattern-based transformations.

5.6.2 *Case Study: Evaluating Existing PDL Patterns*

To evaluate our verification approach across different compiler domains, we analyzed three sets of PDL patterns targeting different application areas. The first two sets, comprising 69 patterns total, were previously ported from C++ to PDL as part of the same work [70]. Of these, 31 patterns come from MLIR's `arith` dialect, performing scalar optimizations common in general-purpose compilers. These patterns implement various arithmetic simplifications and canonicalizations, such as constant folding and algebraic simplification. Notably, our SMT-based verification identified a previously unknown bug involving type constraints in one of these patterns, as discussed in Section 5.4.1.

The second set contains 38 patterns from CIRCT's `comb` dialect, focusing on hardware-specific optimizations of combinational logic operations. These patterns capture bit-level transformations essential for hardware synthesis. A representative example is shown in Listing 5.12, which optimizes three-input `and` operations with constant inputs: `and(x, cst1, cst2)` $\rightarrow$ `and(x, cst1 & cst2)`.

The pattern uses `pdl.apply_native_rewrite` to call into native C++ code to compute the value for the new constant. This highlights a fundamental limitation of PDL: even a simple logical AND operation requires implementation through a native C++ function call. This not only limits static verification but also makes patterns more complex to write and understand, as basic operations must be expressed through external function calls rather than directly in the pattern language. This limitation is particularly striking in hardware transformations, where such bit operations are fundamental building blocks.

The third set consists of 6 patterns from the IREE compiler, targeting machine learning workloads. While IREE has not yet adopted PDL extensively, these patterns demonstrate complex transformations in the machine learning domain. Each pattern matches equivalent multi-layer perceptron structures across different frontend dialects (Tosa, Torch, and Linalg), outlining the

```
pdl.pattern @AndMinusOne : benefit(0) {
  %t = pdl.type : !transfer.integer
  %x = pdl.operand : %t

  %cst1_attr = pdl.attribute : %t
  %cst1_op = pdl.operation "hw.constant"{"val" = %cst1_attr} -> (%t)
  %cst1 = pdl.result 0 of %cst1_op

  %cst2_attr = pdl.attribute : %t
  %cst2_op = pdl.operation "hw.constant"{"val" = %cst2_attr} -> (%t)
  %cst2 = pdl.result 0 of %cst2_op

  %and_op = pdl.operation "comb.and"(%x, %cst1, %cst2) -> (%t)

  pdl.rewrite %and_op {
    %merged_cst = pdl.apply_native_rewrite
                    "andi"(%cst1_attr, %cst2_attr) : !pdl.attribute
    %cst_op = pdl.operation "hw.constant"{"val"=%merged_cst} -> (%t)
    %cst = pdl.result 0 of %cst_op
    %new_op = pdl.operation "comb.and"(%x, %cst) -> (%t)
    pdl.replace %and_op with %new_op
  }
}
```

Listing 5.12: PDL pattern modeling constant propagation in AND operations from the CIRCT comb dialect.

matched computation into calls using IREE's custom `flow` dialect.

IMPLEMENTATION CHALLENGES This reliance on native C++ calls affects patterns across all domains. Both the CIRCT and IREE patterns in our study made extensive use of such custom C++ calls. To make our analysis applicable, we modified these patterns to replace C++ calls with equivalent dummy PDL operations. While this allows us to analyze the patterns' structure and constraints, we are effectively verifying PDL approximations rather than the production patterns.

RESULTS After addressing the identified arithmetic pattern issue, we applied both our SSA property analysis and SMT-based constraint verification to all 75 patterns, including the pattern in Listing 5.12. The results confirmed that all patterns maintain SSA structural integrity and satisfy their specified operation constraints. In total, we verified approximately 2,400 lines of PDL code, spanning from simple arithmetic rewrites to complex outlining transformations.

Our evaluation demonstrates the verification approach's applicability across different compiler domains: scalar optimizations in general-purpose compilers, bit-level transformations in hardware synthesis, and neural network computations in machine

learning. However, our evaluation is limited by PDL's current adoption in production compilers. While we analyzed patterns from real projects, most compiler transformations are still written in C++ or TableGen. As PDL adoption grows, future work should validate our approach on a broader range of patterns from production environments.

5.6.3 *Limitations and Future Extensions*

While our static analysis algorithm successfully prevents many classes of SSA violations, certain aspects of PDL patterns require conservative approximations that might reject valid patterns.

1. HANDLING OF WILDCARD MATCHES A significant challenge arises from PDL's wildcard operations. These occur when `pdl.operation` does not specify an operation name, as shown in Listing 5.13:

```
pdl.pattern : benefit(1) {
  %type = pdl.type
  %any_op = pdl.operation -> (%type)  // Wildcard match
  pdl.rewrite %any_op {
    %new_op = pdl.operation "my.dummy" -> (%type)
    pdl.replace %any_op with %new_op
  }
}
```

Listing 5.13: Example of a PDL pattern using a wildcard operation match, which poses challenges for static verification due to its potential to match terminator operations

Such wildcard operations match any operation with compatible operands, types, and attributes. Currently, the analysis must conservatively assume these operations could be terminators, leading to potentially unnecessary restrictions in pattern verification.

To address these limitations, we propose modifying PDL's handling of wildcard operations. By default, wildcard operations would not match terminators, making this the safe default behavior. An explicit annotation would be required to enable matching of terminators:

```
pdl.operation {matches_terminator}
```

This change would make pattern verification more precise while maintaining safety, as pattern authors must explicitly opt-in to matching terminator operations.

2. ERASURE AND REPLACEMENT MODELING The current PDL modeling of `pdl.erase` and `pdl.replace` closely mirrors the internal MLIR rewriting API, which introduces harsh limitations:

A critical concern with `pdl.erase` is the lack of a mechanism to check for external uses of an operation, rendering any use of `pdl.erase` inherently unsafe. External uses here refer to any operation that uses a value produced by an operation but is not part of the pattern's match. There is always the possibility that such uses exist outside the pattern, forcing our analysis to conservatively reject any pattern containing `pdl.erase`, except when erasing operations created within the pattern itself. Although `pdl.replace` appears to offer a safer approach by replacing uses of an operation, it also deletes the original operation. This dual action leads to a surprising behavior: attempting to replace an operation with itself—seemingly a noop—actually degenerates into an unsafe erasure of the operation.

To mitigate these issues, we propose introducing a new PDL operation, `pdl.check_unused`, to explicitly verify that an operation is unused and can be safely deleted. Additionally, revising `pdl.replace` to solely handle replacements without erasing the operation would enhance its precision and safety, providing developers with greater control in managing IR transformations.

5.6.3.1 *Tool Integration*

To maximize the utility of our approach, we propose integration with MLIR's development tools:

IDE SUPPORT Implementation as an LSP feature would provide real-time verification feedback during pattern development, with inline diagnostics highlighting potential violations.

HYBRID VERIFICATION For patterns where static analysis must be conservative, the framework could generate targeted runtime checks, maintaining safety while allowing more expressive patterns when needed.

5.7 CONCLUSION

This chapter presented a systematic approach to verifying PDL-based transformations in MLIR. While MLIR enforces hierarchical SSA properties through runtime checks, it lacks static

verification of whether PDL patterns preserve these properties. Our static analysis framework addresses this gap by verifying fundamental SSA properties and operation-specific constraints specified through IRDL.

The effectiveness of our approach was demonstrated both through existing patterns, where it uncovered a bug in MLIR's arithmetic patterns, and through systematic fuzzing. Despite the limited availability of real-world PDL patterns, our fuzzing framework enabled comprehensive testing by generating both patterns and matching inputs. This revealed that 87.8% of generated patterns that pass MLIR's current verification could produce invalid IR.

While this work ensures the correctness of individual transformation patterns, the development of sophisticated optimization strategies depends on our ability to compose and orchestrate multiple transformations. Building upon these verification guarantees, the following chapters develop principled approaches to scheduling verified transformations, enabling the construction of powerful optimizations while maintaining the safety properties established here.

6

A PRACTICAL REWRITING SYSTEM FOR MLIR

This chapter is largely based on the publication: "The MLIR Transform Dialect - Your compiler is more powerful than you think" by Lücke, Zinenko, Moses, Steuwer, and Cohen published at CGO'25

In this chapter, we present the Transform dialect, a controllable IR-based transformation system implemented in MLIR that provides fine-grained control over compiler transformations. Building upon the verification guarantees established in the previous chapter, this system enables the safe composition of transformations while maintaining MLIR's extensible nature.

The core Transform dialect, implemented primarily by Alex Zinenko in the MLIR framework, provides the fundamental infrastructure for transformation composition. Building upon this foundation, our work envisions and develops the Transform dialect as a comprehensive replacement for traditional pass managers in compilers, offering complete control over the transformation process. We demonstrate how this system empowers performance engineers to optimize their compute workloads by composing and reusing existing compiler features without implementing new passes or rebuilding the compiler.

Through five case studies and the development of a specialized frontend to the Transform dialect, we demonstrate that this approach enables precise, safe composition of compiler transformations and allows for straightforward integration with state-of-the-art search methods. These case studies validate the practical utility of our approach to transformation orchestration.

6.1 INTRODUCTION

Compilers are typically assembled as a sequence of passes, or *pass pipeline*, augmented with flags to configure this pipeline and influence heuristics. These passes are subsequently run on intermediate representations (IR) of a program until the compilation process concludes. This approach to controlling compilers enables users to order the passes of the compiler and to perform specific optimizations parameterized by their corresponding flags— e.g. apply *loop invariant code motion* on all loops. However, this coarse level of control is increasingly insufficient to optimize programs for today's heterogeneous hardware that

requires precise optimization decisions. Pragmas, or compiler annotations in the source code, provide finer-grained control—e.g. vectorization or unrolling hints. These are effective but their implementation requires in-depth and non-modular changes to the compiler, hence their restriction to specific cases anticipated by compiler engineers.

Often, specific parts of a program dominate the overall runtime and are worth careful and precise optimization, for example, by offloading to an accelerator. In this case, the user, an expert in their application domain, needs the ability to communicate their knowledge on how the program is to be optimized to the compiler. Traditionally, this is performed manually on the program itself by intertwining the algorithm and its optimization using a low-level programming language like C. This style severely limits the portability of a program, often requiring writing and optimizing it again for different target hardware. Additionally, many specialized domains with complex software stacks, such as machine learning, only offer high-level programming models that do not allow expressing optimizations on the required lower level of abstraction.

Recent domain-specific scheduling approaches address these concerns in specific domains: Halide [79] and TVM [12] focus on image processing and machine learning, respectively. They enable expressing a *schedule* separately from the program to specify its optimization. As a result, supporting a new hardware target only requires a new schedule, not a complete redesign of the program. This separation of concerns facilitates reuse and enables the integration of automatic parameter search methods and schedule synthesis [108].

These existing domain-specific approaches introduce their own software stacks specific to their domain and do not integrate well with existing compiler infrastructures. For instance, reusing an existing compiler optimization requires implementing it again in the respective framework. While modern compiler construction infrastructure such as MLIR contains similar transformations as implemented in the Halide and TVM compilers, these do not get exposed to users. For instance, MLIR contains functions that implement tiling, loop interchange, or loop unrolling but only exposes them bundled as a pass which has to be applied to all loops. These passes typically follow a specific heuristic that yields good results on certain benchmarks but leaves users without further control. Realizing a specific composition of transformations, akin to a schedule, using the

existing functions in MLIR requires users to write new passes in the compiler source language and rebuild the compiler. A process that requires expert knowledge beyond the domain of the program the user aims to express and optimize.

With *Transform dialect*, we present an approach to exposing existing but currently hidden compiler features and precisely composing them to control the optimization of programs. We represent our transformation language for controlling compiler optimizations directly as compiler IR.

With our approach, the input to compilers consists of two parts: (1) the computational IR describing the computation being optimized, referred to as the *payload program*, and (2) the Transform script for controlling the compiler. The compilation process is driven by interpreting the Transform IR that describes how to gradually transform and optimize the payload program.

The Transform dialect is implemented in MLIR and contains a base set of operations to model compositions of transformations. It follows an extensible design to expose new transformations. We extend MLIR with an interface to make existing helper functions used in passes accessible to Transform scripts. Thus, exposing fine-grained transformation steps without compromising on their usability from other native code. By introducing the Transform dialect, we open up the hidden features of general-purpose compilers to non-compiler experts in order to enable precisely injecting domain knowledge into the compilation process.

Our contributions are:

- The Transform dialect, an extensible approach for fine-grained control of compiler transformations. (Section 6.3)
- A system of pre- and post-conditions to statically detect problems in compiler pipelines (Section 6.3.3)
- A specialized frontend that simplifies the composition of transformations while maintaining the Transform dialect's safety guarantees, with particular focus on integration with machine learning frameworks (Section 6.4)
- An evaluation of the Transform dialect with five case studies (Section 6.5) highlighting its low computational overhead and robustness, as well as its usefulness for detecting performance problems, generating high-performance code, and exploring optimization spaces.

6.2 MOTIVATION AND BACKGROUND

In the past, compilers were mostly designed to optimize for common cases. This made sense in a world where applications were written in one of a few general-purpose programming languages, such as C, C++, and Fortran. Compiler engineers could also focus on a relatively limited class of heuristics, as code was generated only for one of a few popular computer architectures, such as x86 and ARM.

This picture has already become one of the distant past. Nowadays, software is increasingly written in domain-specific languages and software stacks.

This has sparked the development of domain-specific compilers, such as Halide [79], TVM [12], TACO [38], Lift [92] & RISE [90], and more, but also of new compiler frameworks to facilitate their development, most prominently MLIR [45]. Our hardware landscape has also evolved, resulting in a growing diversity of accelerator architectures, including GPUs, Google's TPU, Groq's LPU, and Cerebras' wafer-scale engine.

Optimizing programs in this new landscape is more challenging. It requires tweaking and adjusting optimizations for the target hardware architecture and the specific computation performed by the input program. But, we know from the domain of machine learning that performing optimizations specific to individual computational kernels can lead to significant performance gains that can not be ignored.

6.2.1 *Controlling MLIR Transformations*

While MLIR's pass infrastructure and pattern-based transformations (introduced in Section 2.1.4) provide a foundation for program optimization, they present several practical challenges. Passes in modern compilers are designed to be reusable in different pipeline configurations, enabling performance engineers to explore which configurations lead to the best performance for their program and target scenario. However, this approach has two major limitations:

1. The control via passes is coarse-grained, with no universal way to restrict passes to targeted parts of the program
2. Pass pipeline composition is restricted by implicit dependencies between passes that are not explicitly expressed beyond textual documentation

A significant driver of the implementation of individual optimizations in MLIR is the domain of linear algebra computations found in ML workloads, particularly those expressed using *structured operations* in the Linalg and related dialects [100]. Initially, high-impact transformations, such as tiling and fusion, were orchestrated using a system designed for peephole optimization [58] of static single assignment IRs, borrowing ideas from term rewriting. In this style, each transformation is expressed as a *rewrite pattern*, implemented in an imperative style using C++ code that matches relevant operations and replaces them with a transformed equivalent. As more fine-grained controls are lacking, such patterns are applied greedily to the entire program until a fixpoint. This approach has led to: *1)* rewrite patterns that progressively include target-specific hard-coded heuristics, e.g., only to unroll loops with less than 8 iterations; *2)* conflicts between different rewrite patterns that are applicable to the same inputs, resulting in situations where it is unclear, e.g., whether a loop should be first tiled or fused with another loop.

As a remedy, such patterns started relying on IR metadata to externalize heuristic decisions (e.g., this loop should be tiled with size 32), establish order (e.g., this operation has been tiled and should not be tiled again) or otherwise communicate with each other. However, this approach is brittle as metadata is not guaranteed to be preserved by transformations. Furthermore, metadata may not reference IR constructs to which it is not directly attached, requiring delicate string-based matching and additional IR traversals to link together several IR pieces.

We need a more principled approach to facilitate the fine-grained control of compiler optimizations in MLIR.

6.2.2 *Scheduling Languages for Separating Computations and Optimization Decisions*

Halide [79] pioneered the idea of separating computation from a *schedule* that specifies the sequence of optimizations to be performed. This approach has the advantage of portability: while the computation remains unchanged, different schedules describe different optimizations for different hardware targets or even inputs. This model has since been popularised by other domain-specific compilers, including TVM and TACO. Unfortunately, all these solutions are domain-specific and not accessible to a generic compiler framework.

ELEVATE [90] demonstrates how to build a generic scheduling language from first principles using formal programming language foundations. However, this formal rewrite-based approach requires the computational program to be side-effect-free — a significant restriction that is unrealistic to assume or to ensure in many practical settings.

Therefore, we see a need for a scheduling language for a generic compiler framework that enables fine-grained control of compiler optimizations for a wide variety of settings. In the following, we describe such a practical scheduling language for MLIR implemented as the *Transform dialect*. We will explore an alternate design based on the foundations of Elevate [28] in the next Chapter.

6.3 TRANSFORM DIALECT: REPRESENTING AND CONTROLLING TRANSFORMATIONS USING IR

We introduce the Transform dialect by example, designing an optimization as a composition of existing transformations and applying it to a program as shown in Figure 6.1.

The Transform script shown in Figure 6.1a first hoists code out of a specific loop (line 3) before splitting a nested loop into two loops (line 6), then tiles the first resulting loop (line 8) and unrolls the second (line 10). The script contains a deliberate error (line 11) where we attempt to unroll the same loop a second time to highlight that such errors are detected statically.

This script is executed sequentially from top to bottom and consists of MLIR operations that we call *transforms*. A transform may directly model a transformation, such as in lines 3, 6, 8, 10, and 11, where part of the payload program is rewritten, or a transform may assist in targeting or composing other transforms, for instance, by matching a nested operation such as in lines 2 and 4. Transforms define values called *handles*, each of which refers to a list of operations in the payload IR. As Transform scripts are represented in MLIR itself, handles are regular MLIR values that obey the usual Static Single Assignment (SSA) rules. Other transforms may use handles as operands in order to transform the associated payload operations or extract information from them. The relation of transforms and their associated payload operations is indicated in the example through similar coloring. Additional operations with regions may be used to organize the Transform script into conditionals, loops, or even functions.

```
named_sequence @split_then_tile_and_unroll(%func) {
  %outer     = match.op "scf.for" {first} in %func         // type: outer:    {scf.for}
  %hoisted   = loop.hoist from %outer to %func             // type: hoisted:  {*.*}
  %inner     = match.op "scf.for" {first} in %outer        // type: inner:    {scf.for}
  %param     = param.constant 8
  %part:2    = loop.split %inner ub_div_by=%param          // type: part#1:   {scf.for[ub%8=0]}
                                                           //       part#2:   {scf.for[ub<8]}
  %tiled:2   = loop.tile %part#1 tile_sizes=[%param].      // type: tiled#1:  {scf.for[part#1.ub/8]}
                                                           //       tiled#2:  {scf.for[ub=8]}
  %unrolled  = loop.unroll %part#2 {full}                  // type: unrolled: {*.*}
  %unrolled2 = loop.unroll %part#2 {full}                  // This statically reports an error!
}
```

(a) Operations of the Transform dialect are used to express a tiling optimization of an uneven inner loop. Inputs and outputs to transforms are represented explicitly to enable precise chaining. Metadata such as tile sizes are used to further configure individual transforms. Static reasoning about the possible structure of the payload IR is shown in comments on the right-hand side.

```
func @myFunc(%values: memref) {


  scf.for %i = 0 to 4096 {
    %c1 = arith.constant 1
    scf.for %j = 0 to 2042 {
      %c2 = arith.constant 2

      %val = memref.load %values[%c1, %i, %j]
      func.call @use(%val, %c2)
    }









  } }
```

(b) Initial payload IR

```
func @myFunc(%values: memref) {
  %c1 = arith.constant 1
  %c2 = arith.constant 2
  scf.for %j = 0 to 4096 {

    scf.for %i_0 to 255 {
      scf.for %i_1 = 0 to 8 {
        %i   = arith.muli %i_0, %i_1
        %val = memref.load %values[%c1, %i, %j]
        func.call @use(%val, %c2)
      }
    }
    %val2020 = memref.load %values[%c, 2040, %j]
    func.call @use(%val)
    %val2021 = memref.load %values[%c, 2041, %j]
    func.call @use(%val)
  } }
```

(c) Transformed payload IR

Figure 6.1: A Transform script performing code hoisting, loop splitting, tiling, and unrolling is defined (a) and used to transform the initial payload IR (b) to the transformed IR (c). The operations associated with specific handles are colored similarly.

In addition to handles, the Transform script may use *parameters*. These values may be unknown when the Transform script is created but known when it is executed. Since they also obey SSA, parameters are effectively constant during execution. Together with attributes on the transform operations, parameters specialize the operation by providing, e.g., sizes for loop tiling or the preferred vector width. Figure 6.1a uses a constant parameter

in line 5 to specify split and tile sizes. This approach makes the Transform dialect a medium for *externalizing compiler heuristics*: instead of parameterizing a transform in the compiler code, one can now generate Transform scripts with function-like abstractions that accept parameters as operands and use them inside the transformation. Parameters can also be derived from the payload IR. For example, an operation accepting a loop handle and producing a parameter with desired tile sizes is perfectly suitable for the Transform dialect.

Transform scripts are executed by the compiler when compiling the program via an interpreter that maintains the association table between handles and payload operations and dispatches execution to transformation logic implemented in C++ using MLIR interfaces.[1] While our current interface-based approach is designed for compiler extensibility, Transform scripts can also be lowered down to LLVM IR, (JIT-) compiled and linked to compiler libraries should compiler performance become a critical concern.

Since transformations may not always succeed, the transform interpreter provides an error handling mechanism similar to exceptions.[2] A transform may signal a silenceable or a definite error. In the former case, the interpreter will skip the remainder of the current region and return control flow to the parent transform operation. This operation may decide to suppress the error or report it for further handling. Silenceable errors typically indicate a failed precondition or at least that the payload has not been modified irreversibly. Definite errors cannot be suppressed and are immediately reported, aborting the interpreter.

A detailed description of the Transform dialect extensibility mechanisms is available in the documentation.[3]

6.3.1 *Dealing with a Mutable IR*

Contrary to many purely functional approaches [90, 92, 103], the Transform dialect is embedded into a practical compiler infrastructure operating on a payload IR that is implemented mutably for efficiency reasons. A transformation in MLIR may erase the payload operation to which a handle is pointing, ei-

1 MLIR interfaces provide dynamic polymorphism, similarly to OOP.

2 As of the time of writing, MLIR does not model exceptions. We resort to implicit error-checking semantics in all transforms. All transforms check if an error has been signaled and are implicitly no-ops in that case.

3 https://mlir.llvm.org/docs/Tutorials/transform/

ther to replace it with a newly created operation or to remove it irreversibly, leaving the handle *dangling*. To prevent invalid access through such handles, we introduce the notion of *handle invalidation*, akin to the concept of iterator invalidation known in C++. Therefore, a transform must indicate whether it invalidates its operands, which internally corresponds to indicating a "memory deallocation" side effect on the operation. Invalidating a handle also invalidates any other handles to the same payload or any of its parts, e.g., a nested operation. Invalidated handles cannot be used as operands anymore.

To enable chaining of transforms, transforms that invalidate their operands typically return new handles to their results, such as in lines 6 and 8 in Figure 6.1a. For instance, a loop reversal transformation would invalidate the handle to the loop and return the handle to the reverted loop if its implementation recreates the loop operation and could preserve the handle if the reversal is done in place. Returning new handles from a transformation brings the representation conceptually close to functional approaches, especially given the single-assignment property of the IR [2], but the underlying payload IR remains mutable.

To help the user avoid accessing invalidated handles in the Transform script, the interpreter keeps track of invalidation, including handles invalidated indirectly due to nesting in the payload IR discovered by traversing the payload IR along with the handle/operation mapping, and reports errors. Static analysis is also possible as described in Section 6.3.4.

Additionally, a transform may subscribe to "operation replaced" and "erased" events from the MLIR rewrite driver, used by large passes such as dialect conversions and peephole optimizations. In reaction to these events, the transform may prevent handle invalidation by updating the handle to point to the replacement operation or to point to the empty set of operations, depending on the desired logic.

6.3.2 *Extensibility*

The Transform dialect builds on MLIR's extensibility, allowing advanced users to define new transformation operations associated either with existing or new custom IR transformations implemented in the compiler. These transforms can naturally mix with each other, as with any other dialect.

When modifications of the compiler are undesirable, too challenging or simply impossible, one can nevertheless create new transform abstractions as combinations of existing transforms. Indeed, since transforms are mere operations, they can be organized into macros or functions using abstractions already present in MLIR. For example, the `named_sequence` operation on line 1 of Figure 6.1a defines a macro that can be expanded in any other place using the `include` operation.[4]

In Halide and TVM, the schedule and computational program are written closely side-by-side, sharing the same variable scope, and the schedule can directly refer to computational variables. But this means also that schedules are specific to a single program. In contrast, in our approach, Transform scripts are not program-specific and separated from the computational program, making the compositions of transforms easily reusable and further composable. This, in turn, opens the door for building libraries of composed compiled transforms and distributing them, potentially separately from the compiler.

Building upon these extensibility mechanisms, we explore concrete scenarios where transform scripts provide value beyond loop transformations in Figures 6.2, 6.3 and 6.4. Note that these examples use simplified transform operations that, while not existing exactly in this form, illustrate how such transformations could be composed using the Transform dialect's mechanisms.

```
named_sequence @map_to_gpu(%module) {
  // Identify regions of code suitable for GPU execution
  %kernels = gpu.identify_kernels %module #criteria
  // Map computations to GPU thread blocks with specific dimensions
  %mapped = gpu.map_to_blocks %kernels sizes=[128, 1, 1]
  // Apply GPU-specific pipeline optimization
  %optimized = gpu.pipeline %mapped
  // Lower to NVIDIA's NVVM dialect for final code generation
  %lowered = gpu.lower_to_nvvm %optimized
}
```

Figure 6.2: Transform script orchestrating the process of mapping computations to GPU accelerators through a sequence of transformations.

These examples demonstrate how Transform scripts can encode expert knowledge about hardware targets, domain-specific optimizations, and specialized transformation sequences. Furthermore, since transforms are represented as ordinary compiler

4 In the implementation, macros are preferred to functions in many cases as they lead to simpler flow in the interpreter.

```
named_sequence @optimize_layout(%fun) {
  // Find all tensor load operations in the function
  %tensors = match.op "tensor.load" in %fun
  // Transform the memory layout into blocked format for locality
  %blocked = layout.to_blocked %tensors block_size=[32, 32]
  // Add padding to ensure alignment and avoid edge cases
  %padded = layout.pad %blocked padding=4
  // Convert memory accesses into vectorized reads
  %vectorized = vector.transfer_read %padded vector_size=8
}
```

Figure 6.3: Transform script demonstrating systematic memory layout transformations for hardware optimization, showing the progression from identification through blocking, padding, and vectorization.

```
named_sequence @ml_inference_optimize(%fun) {
  // Fuse element-wise operations
  %fused = fusion.extract_fusion_groups %fun
  // Quantize computations
  %quantized = quant.prepare_and_quantize %fused
  // Target specific hardware features
  %vectorized = vector.transfer %quantized target="avx512"
}
```

Figure 6.4: Transform script showing a specialized optimization sequence for machine learning inference, combining fusion, quantization, and hardware-specific vectorization.

IR, they can be subject to compiler rewrites themselves. For example, the GPU mapping transformation shown in Figure 6.2 could be lowered to more specific operations depending on the target architecture - high-level block mapping operations might be rewritten into combinations of vendor-specific operations for NVIDIA or AMD GPUs. Similarly, traditional loop transformations like tiling can be "lowered" to the canonical combination of loop strip-mining and interchange instead of being implemented directly, and these loop interchanges may cancel out with eventual further interchanges or other transforms present using regular pattern-driven peephole optimization.

The separation of Transform scripts from the computational program makes these transformations reusable across different programs with similar patterns. This enables building libraries of compiled transforms that can be distributed separately from the compiler, allowing domain experts to share optimization strategies without requiring deep compiler expertise from users.

6.3.3 *Composability with Pre- and Post-Conditions*

Most compiler transformations come with assumptions about the input IR that are required for the transformation to be successfully applied. When these pre-conditions are not satisfied, defensively written transformations will not modify the IR and potentially warn the user, but poorly written transformations may result in miscompilations or introduce subtle bugs. The Transform dialect provides a natural place to explicitly declare pre-conditions as well as specify post-conditions guaranteed by the transformation to ensure working compositions. Pre- and post-conditions are expressed using transform operation attributes as well as the types of handles, both of which are fully user-extensible in MLIR.

One of the most common uses for the pre-and post-conditions is for a set of *progressive lowering* transformations. Each of these removes a subset of operations belonging to one dialect and introduces new operations, potentially from another "lower" dialect. For example, a `convert_scf_to_cf` lowering transform replaces structured control flow (SCF) dialect operations, such as loops and conditionals, with classical branch-based control flow defined in the CF dialect. Building a full compilation flow in MLIR requires composing multiple lowerings until all operations use the desired set of the final target dialects, such as LLVM IR or SPIR-V dialects. Determining the right order of lowerings is usually an expensive, trial-and-error process subject to the phase ordering issues [109] as there is no programmatically accessible information about the operations that are being lowered out and those that are being introduced.

The Transform dialect allows developers to make pre- and post-conditions explicitly available by listing operations that are being added and removed by each lowering transform as shown in Figure 6.5. In this example, the next lowerings should take care of converting CF and arithmetic operations toward the desired target dialects. One can also statically observe that any loop transformations operating on SCF, such as interchange or unrolling, must be ordered before `convert_scf_to_cf` ,allowing to check for phase ordering violations statically. We have implemented a prototype tool for checking statically if a composition of transforms violates the specified pre-conditions of any individual transform.

To further facilitate extensibility, it is also possible to not list specific operation names in the pre-and post-conditions, but

```
1 transform.convert_scf_to_cf(input: {scf.*}) ->
2   (result: {cf.branch, cf.switch, cf.cond_branch,
3         arith.addi, arith.cmpi, arith.index_cast})
```

Figure 6.5: Pre-/post-conditions of `convert_scf_to_cf` declare which kinds of payload operations are consumed and removed (all operations from the `scf` dialect), and which new operations are introduced by this transform (the operations listed explicitly in lines 2 and 3).

operation interfaces instead. Operation interfaces,[5] are introduced in MLIR to group similarly behaving operations, such as operations that perform a memory allocation or have a specific side-effect.

ADVANCED PRE- AND POST-CONDITION Oftentimes, pre- and post-conditions of a transformation require more detail than purely demanding the presence of certain operations. The Transform dialect supports advanced pre- and post-conditions via integration with the declarative IR Definition Language (IRDL) [21]. IRDL is primarily used to declaratively specify operations of dialects with their types and invariants. Figure 6.6 shows the IRDL definition of the `memref.subview` operation, that converts one memory reference type to another type which represents a reduced-size view of the original memory.

To specify advanced transform pre-and post-conditions, we leverage IRDL's capability to further constrain operations and types for *existing* operations without changing their definition.

The `expand_strided_metadata` transformation, shown in Figure 6.7, modifies `memref` dialect operations so that complex strided address computations are factored out, and the remaining accesses are akin to trivial flat pointers [100]. These simplified accesses are characterized by trivial `subviews` where the access offsets, sizes, and strides are empty. Figure 6.6 with highlights shows the pseudo operation that we introduce to represent this constrained, where the `offset`, sizes and `stride` operands of a `subview` are guaranteed to have cardinality zero. Note that we only use this IRDL definition to be able to specify the post-condition in Figure 6.7 and that we *do not* actually introduce a new operation.

5 https://mlir.llvm.org/docs/Interfaces/

```
Dialect memref {
Operation subview .constr {
  Attributes(
    static_offsets: Variadic<!indexAttr>,
    static_sizes:   Variadic<!indexAttr>,
    static_strides: Variadic<!indexAttr>)
  Operands(
    input:    !memrefType,
    offset:   Variadic<!index  , 0 >,
    sizes:    Variadic<!index  , 0 >,
    strides:  Variadic<!index  , 0 >)
  Results(view: !memrefType)
  CPPConstraint "checkMemrefConstraints()" }}
```

Figure 6.6: IRDL definition of the `memref.subview` operation. Highighted parts are added in a copy of definition that expresses the post-condition of the `transform.expand_strided_metadata` transformation.

```
transform.expand_strided_metadata(input: {memref.*}) ->
  (result: {memref.subview.constr,
    memref.extract_strided_metadata.constr,
    memref.extract_aligned_pointer_as_index.constr,
    memref.reinterpret_cast.constr,
    affine.min, affine.apply, arith.constant_index})
```

Figure 6.7: constraints on input and output handles of the operation `transform.expand_strided_metadata`

CHECKING PRE- AND POST-CONDITIONS DYNAMICALLY

Existing MLIR transformations do not declare their pre- and post-conditions explicitly. To help with the process of adding such declarations, we leverage IRDL's capability to automatically generate constraint verifiers. These verifiers can be used to dynamically check pre- and post-conditions.

These dynamic checks are performed while transforming a concrete input program and are also useful even when pre- and post-conditions have been specified explicitly – and have been checked with our static tool. We do this, as we can not check if the specified pre- and post-conditions are actually accurate specifications for the concrete transformation implementations usually written in C++. Therefore, the dynamic checking can serve as an additional tool to detect bugs in transformations.

6.3.4 *The Transform IR*

The implementation as a dialect in MLIR enables multiple usage scenarios, where users either write Transform scripts directly

in the MLIR format or alternatively use one of the multiple alternative MLIR frontends, including Julia [59] or Python [51].

Exposing the Transform script as IR also means that we can use compiler analyses and transformations on the Transform script itself. This can be used in multiple scenarios.

STATIC ANALYSIS OF HANDLE INVALIDATION Since transforms are just MLIR operations, static analysis for handling invalidation is readily available as of-the-shelf "use after free" dataflow analysis. It suffices to express handle definition as an "allocate" side effect and handle invalidation as a "free" effect on some notional memory location and to express handles pointing to the same or nested payload operations as (partial) aliasing. *This showcases the benefit of expressing Transform scripts as regular compiler IR.*

COMPOSITION, SIMPLIFICATION AND CONSTANT PROPAGATION Named macros described in Section 6.3.2 are function-like objects that can be easily processed by existing function transformations. Indeed they may be implemented by simply calling the inliner pass and instructing it to always inline functions. Since macros don't support recursion, which is itself verified by checking for cycles in their call graph, inlining is always possible.

This, in turn, enables other simplification using MLIR peephole optimizer and constant folder. Similarly to any operation, transform operations can define local simplification rules, e.g., unrolling by 1 or tiling by 0 are noops, so is tiling by a larger size than the previously applied tiling. Constant parameters and typing information can be propagated through the script enabling further simplification using the pre-existing simplification driver. Note that performing this on the Transform IR removes the need for the transform to be applied to the payload, most often saving on compile time for advanced transformations requiring expensive analyses.

AUTOMATED PIPELINE CONFIGURATION VIA INTROSPECTION Some transformations are meaningful at different levels of abstraction. Automatic differentiation (AD), a process crucial for machine learning training, is such a transformation and is often implemented as a compiler pass [63]. AD produces instructions computing a derivative of a given variable. The full derivative is systematically a sum of partial derivatives with

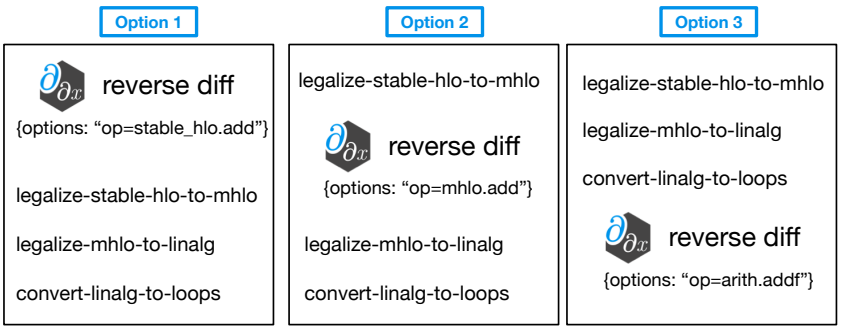


Figure 6.8: Three alternative options for when to perform the automatic differentiation transformation. We use a transform to introspect the pipeline and automatically infer the appropriate transformation option.

respect to different values. However, the notion of a sum is ambiguous as there are multiple kinds of additions implemented in various MLIR dialects. The AD pass needs to create "add" instructions of the right kind for IR to remain compatible with the rest of the transformation pipeline. This, in turn, requires the AD pass to know its place in the pass pipeline and adjust accordingly. In the JAX framework [24], the IR is progressively rewritten from StableHLO[6] to MHLO[7], to Arithmetic, and LLVM dialects at various stages, each of which defines an "add" operation. Depending on when the AD pass is scheduled, it must produce the operation from the corresponding dialect. While the pass could analyze the IR to understand which "stage" it is in, such an analysis could be imprecise since multiple dialects can co-exist in the same translation unit, and maintaining fine-grained control is important.

Using a Transform script, we can precisely control the abstraction level at which we want to perform AD, as shown in Figure 6.8. However, we must manually configure the automatic differentiation pass [63] that operates on MLIR and has been parameterized by the kind of addition operation to emit. To avoid manually specifying this unnecessary detail, we introspect the Transform script to infer the parameter based on the position in the script. This is implemented as an ordinary compiler transformation performing a simple automatic traversal over the Transform IR.

6 https://openxla.org/stablehlo
7 https://www.tensorflow.org/mlir/hlo_ops

6.3.5 *Summary*

In this section, we introduced the Transform dialect for composing and controlling compiler transformations. We discussed various features, including handle invalidation, extensibility, ensuring composability via pre-and post-conditions, and the representation of Transform scripts as MLIR IR.

6.4 A PYTHON FRONTEND FOR TRANSFORM SCRIPTS

High-performance machine learning demands sophisticated optimization capabilities. While several frameworks address this need, JAX [24] in particular has become a cornerstone of ML research by building on NumPy's familiar API and adding powerful capabilities like automatic differentiation, vectorization (vmap), and just-in-time compilation (jit). While JAX achieves high performance by transforming Python code through multiple compilation stages, optionally using MLIR as its compiler infrastructure, the optimization process itself remains largely opaque to users who might want to tune performance for specific hardware targets or application patterns.

The Transform dialect in MLIR provides precisely the kind of fine-grained control over compiler optimizations that performance engineers need when optimizing ML models. With its ability to express complex transformation sequences and target specific operations for optimization, it is best suited to enable engineers to precisely tune JAX computations for specific hardware. Moreover, since JAX already uses MLIR internally, the Transform dialect could integrate naturally with JAX's compilation pipeline. However, accessing this power is not straightforward, requiring significant effort and knowledge. Transform scripts can be written directly in the MLIR textual format or generated programmatically using MLIR's Python bindings, which are partially auto-generated from the TableGen specification. While writing Transform IR directly provides complete control over the transformation process, it requires a deep understanding of MLIR's syntax and semantics. Even for MLIR experts, writing correct Transform scripts is challenging as they must carefully manage concepts such as handle invalidation to prevent invalid access to transformed operations. Furthermore, handwriting transformation patterns creates a disconnect between the JAX program being optimized and the transformations being applied. There is no direct way to identify which operations in

the JAX program should be targeted by specific transformations. This makes it difficult to maintain and update optimization strategies as programs evolve. A more intuitive system that allows developers to mark operations in their JAX code for specific optimizations would create a clearer connection between computation and optimization intent.

To address this challenge, we developed a specialized frontend that builds upon MLIR's Python bindings and is designed to work alongside JAX computations. While the Python bindings require detailed knowledge of MLIR's internal structure and the Transform dialect's specific operations, our frontend provides a more intuitive interface while maintaining full expressiveness. It offers an ergonomic API that enables natural chaining of transformations, allowing multiple operations on handles to be expressed concisely in a single line rather than requiring multiple SSA-style assignments in IR. The frontend leverages Python's type system to catch certain classes of invalid handle accesses as type errors during development, while more complex cases requiring the analysis mentioned before in Section 6.3.1 are reported after the generation of the Transform IR. Operations in the JAX computation can be targeted either by matching their structure in the generated IR or by using explicit tags to mark operations that should be transformed later.

Our frontend provides a Python interface that makes it natural to express transformation sequences while leveraging Python's syntax for control flow and abstraction. Consider a batch matrix multiplication in JAX that we want to optimize, shown in Listing 6.1. This example demonstrates several key features of our frontend:

LINE 2 Computations can be tagged in the source JAX code for later transformation using the tag function, here applied to a batch matrix multiplication.

LINE 9 The `@requires_normalform` decorator ensures the IR is in the required form before applying transformations, automatically inserting enabling transformations if needed. We discuss normal forms in more detail in Section 6.4.1.

LINE 10 Transformations can be composed into reusable functions like `optimize_loops`, which encapsulates a common optimization pattern.

LINES 14-20 Using `transform.alternatives`, the frontend enables expressing fallback strategies - here attempting to

```
1  def simple_matmul(a:Array[32,32], b:Array[32,32]) -> Array[32,32]:
2      return tag(jax.matmul(a, b), "compute")
3
4  def schedule(module: OpHandle):
5      # First lower linalg IR to scf loops
6      compute = module.match_tagged_op("compute")
7      loops = compute.lower_to_scf()
8
9      @requires_normalform(PerfectLoopNest)
10     def optimize_loops(loop_nest: OpHandle):
11         loop_i, loop_j, tile_i, tile_j = loop_nest.tile([16, 16])
12         tile_j.interchange(tile_i) # Swap inner loop order
13         loop_k = tilei.get_nested_loop()
14         with transform.alternatives() as alt:
15             with alt.case():
16                 # Try micro-kernel if available for these sizes
17                 loop_k.replace_with_library("libxsmm")
18             with alt.case():
19                 # Otherwise do standard optimizations
20                 loop_k.unroll()
21
22
23     optimize_loops(loops)
24
25 # Optimize using schedule and compute result
26 result = jit(simple_matmul, schedule)(mat_a, mat_b)
```

Listing 6.1: Example of the Transform dialect frontend showing how transformations can be composed in Python. The code demonstrates tagging operations in JAX code, composing transformations with normalform requirements, and fallback strategies using alternatives.

replace the computation with an optimized library implementation first, falling back to standard loop optimizations if the library doesn't support the specific sizes. This demonstrates how the frontend enables integration with external optimization libraries while maintaining a robust fallback strategy.

LINE 26 The integration with JAX's JIT compilation is seamless, allowing the schedule to be provided alongside the computation for optimization.

INTEGRATION WITH JAX To understand how this code integrates with JAX's compilation pipeline, we illustrate the overall architecture of our frontend in Figure 6.9. The computational part expressed in Python follows JAX's standard lowering path to MLIR's Linalg dialect. Simultaneously, the schedule describing the optimization strategy is lowered to Transform IR through our frontend. Both IRs then serve as inputs to the Transform

interpreter, which orchestrates the application of the specified transformations.

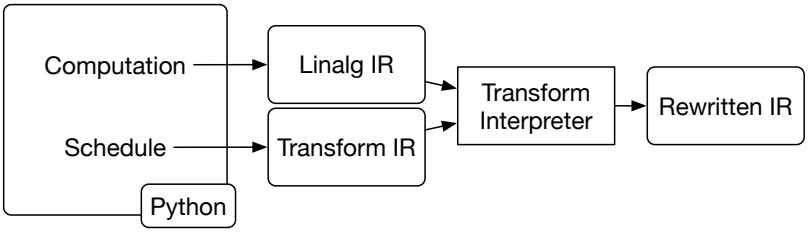


Figure 6.9: Integration of the Transform dialect frontend with JAX. The separation of computation and schedule enables independent specification of optimizations, which are applied to the computational IR by the Transform interpreter.

This architecture necessarily maintains separate compilation paths: the computational code follows JAX's standard lowering path, preserving critical features like automatic differentiation, while the schedule is independently lowered to Transform IR. To connect these separate paths, we use tags like the one applied to the matrix multiplication, which creates a bridge between the computation and its optimization strategy. These tags, implemented as custom JAX primitive, survive JAX's lowering process and enable precise targeting of operations in the schedule, regardless of surrounding code. Furthermore, the separation of the schedule allows for the composition of complex optimization strategies using Python's native control flow constructs, while the Transform interpreter ensures their correct application to the IR.

As shown in Figure 6.10, the compilation process progressively transforms the code through several stages:

①→② The JAX computation is lowered to MLIR's Linalg dialect, which provides a mathematical abstraction of linear algebra operations. During this lowering, the tagged matrix multiplication becomes a `linalg.matmul` operation that preserves the "compute" tag from the original JAX code.

②→③ After matching the tagged operation, the high-level matrix multiplication is converted to explicit nested loops using MLIR's Structured Control Flow (SCF) dialect through the `lower_to_scf()` transformation.

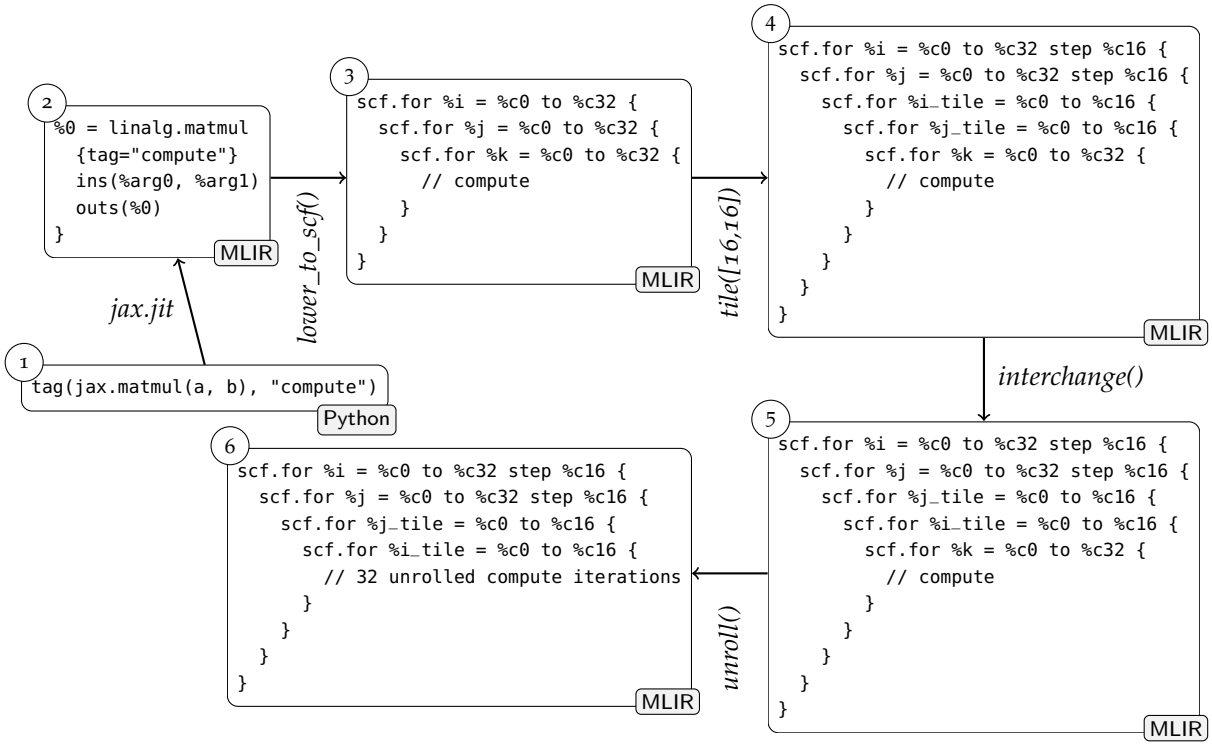


Figure 6.10: Progressive transformation of matrix multiplication from high-level linalg to optimized loop nest.

③→④ The schedule applies loop tiling with specific sizes to improve cache locality, introducing additional loop levels with controlled iteration ranges.

④→⑤ Loop interchange optimizes memory access patterns by swapping the order of the inner loops.

⑤→⑥ Finally, unrolling the innermost loop creates multiple copies of the computation to reduce loop overhead.

Throughout this transformation sequence, our frontend's handle-based API enables the natural chaining of transformations while automatically managing handle invalidation under the hood. For example, when tiling transforms a loop nest, the `loop_nest` handle that initially referred to the outermost loop is invalidated as the loop is replaced. To meet user expectations, we automatically update this handle to refer to the new outermost loop, providing an intuitive interface that matches the conceptual operation of tiling. The preservation of the "compute" tag throughout these transformations enables precise tracking of operations, while the frontend's handle management ensures that

each transformation correctly operates on the evolving IR structure. This demonstrates how our frontend provides fine-grained control over the optimization process while maintaining a clean, intuitive interface that shields users from the complexities of MLIR's internal operation replacement mechanics.

This concrete example demonstrates how our frontend enables the natural expression of complex transformation sequences while handling the intricacies of IR manipulation under the hood. The resulting code exposes optimization opportunities that would be impossible to express through JAX's standard compilation pipeline.

While we demonstrated these capabilities using JAX computations, the Transform dialect is dialect-agnostic and could similarly drive transformations on other MLIR dialects. Indeed, Hagedorn et al. [28] have shown how functional pattern-based computations in RISE (the corresponding MLIR dialect was introduced in Chapter 4) can be systematically transformed using rewrite mechanisms. The Transform dialect provides exactly the kind of orchestration required for such transformations, enabling precise control over the rewriting process from high-level functional patterns to optimized implementations - if the corresponding rewrites were implemented in MLIR. This flexibility stems from the Transform dialect's design as a general orchestration mechanism for MLIR transformations, independent of the specific computational representations being transformed.

6.4.1 *Normalforms: Automating Enabling Transformations*

A significant challenge when composing transformations is that many transformations require the IR to be in a specific format before they can be applied. For instance, a loop fusion transformation might require perfectly nested loops, or a vectorization transformation might need specific memory access patterns. Previously, users must manually apply sequences of "enabling" transformations to prepare the IR for the desired optimization.

Our analysis of real-world Transform scripts revealed that up to 65% of transformations in complex sequences were enabling transformations rather than the actual performance-improving optimizations intended by the user. This not only makes Transform scripts verbose and difficult to maintain but also requires deep knowledge of MLIR's internal IR representation and transformation requirements.

Inspired by term rewriting systems, we introduce the concept of *normalforms* to the Transform dialect. A normalform formally specifies a set of properties that must hold for the IR. For example, the "NestedLoops" normalform might require that all loops are perfectly nested such that no operations exist between the loop levels.

Transformations can now declare their required normalforms as part of their preconditions as shown in Listing 6.2.

```
@transform(required_normalform=PerfectLoopNest)
def interchange(self: OpHandle, outer: OpHandle) -> OpHandle:
  # Creates fusion transform operation
  op = InterchangeOp(self.mlir_value, outer.mlir_value)
  # Update frontend handle representation
  self.mlir_value = op.new_outer
  outer.mlir_value = op.new_inner
  return self
```

Listing 6.2: Interchange transform definition with normalform precondition annotation.

The dynamic checking of normalforms leverages the constrained IRDL definitions introduced earlier. For simple normalforms, we use these definitions to check at runtime whether the IR satisfies the required properties. When violations are detected, the system automatically invokes the necessary enabling transformations.

For example, a normalform requiring specific operand types or operation attributes can be directly expressed through IRDL constraints and checked dynamically. This works well for local properties, which can be verified by examining individual operations.

However, more complex normalforms, such as perfectly nested loops or specific dataflow patterns, require global analysis of the IR structure. For these cases, we use static reasoning based on the Transform dialect's specification system. Each transformation declares which normalforms it requires and maintains. The system then uses this information to determine necessary enabling transformations through static analysis before the transformation sequence is executed.

This two-tiered approach - dynamic checking through IRDL for local properties and static analysis for structural properties - provides a robust and efficient mechanism for managing IR formats throughout transformation sequences. While it does not guarantee absolute correctness, it significantly increases the probability that planned optimizations will be applicable by

detecting common violations early and inserting appropriate enabling transformations.

The static analysis helps identify likely transformation sequences needed to establish required Normalforms, while the dynamic IRDL checks verify concrete IR properties at runtime. This combination helps users focus on their optimization intent rather than IR manipulation details. While in complex cases additional enabling transformations might still be needed despite the automated assistance, the system significantly reduces the complexity of Transform scripts and makes them more robust and maintainable. When a required Normalform cannot be achieved, the system provides detailed error messages that help users understand transformation requirements at a higher level of abstraction rather than dealing with low-level IR properties.

The Transform dialect frontend represents a significant step toward making MLIR's powerful optimization capabilities more accessible to domain experts while maintaining the system's full expressiveness. By providing an intuitive Python interface that integrates with JAX, automating common transformation patterns through normalforms, and simplifying handle invalidation, the frontend substantially reduces the barrier to entry for using the Transform dialect. The system proved particularly valuable in practical applications, from rapid prototyping of transformation sequences to integration with autotuning frameworks, where its automatic handle management simplified the implementation of optimization space exploration.

Looking forward, this frontend design opens new opportunities for expanding the Transform dialect's reach beyond compiler experts to domain specialists who can leverage their application knowledge to guide optimizations. While challenges remain in areas such as debugging complex transformation sequences and providing even more intuitive abstractions, the frontend architecture shows promise in bridging the gap between high-level programming models and low-level compiler optimizations without sacrificing control or performance. The following evaluation section will demonstrate how these capabilities translate to practical impact through five comprehensive case studies.

6.5 EVALUATION

We explore 5 case studies with different compiler scenarios. We investigate the overhead of the Transform dialect and highlight the usefulness of pre- and post-conditions for building robust compiler pipelines before showcasing the fine-grained control helpful for detecting performance problems, generating high-performance code, and automatically exploring compiler optimizations.

6.5.1 *Case Study 1: Expressing arbitrary pass pipelines as Transform scripts*

First, we want to establish that we can represent traditional pass pipelines with the Transform dialect and measure its overheads due to the interpretation at compile time.

Prior work on scheduling languages including Halide [79] and ELEVATE [90] validated the effectiveness of utilizing schedule-based compilation strategies for single kernel optimization. However, these approaches are either domain-specific or have not been integrated into larger realistic code generation settings. We explore the feasibility of scaling such schedule-based compilation approaches to modern compiler infrastructures, focusing on large-scale programs such as full machine learning models. In this case study, we measure the overhead of using our Transform scripts that are interpreted at compile time and compare this to the traditional way in MLIR to control compiler transformations: pass pipelines that are specified as command line arguments to the compiler.

To evaluate the overhead of the Transform dialect, we examine five machine learning models implemented using the MLIR-based TensorFlow compiler ecosystem, as listed in Table 6.1. We use a standard pass pipeline, shown in Figure 6.11, that converts from the MLIR Tensor Operation Set Architecture (TOSA) dialect[8] to the Linalg dialect [100], as described in [5].

To obtain a Transform script describing the identical compilation flow, we modified MLIR to automatically create a Transform script of a pass pipeline. The script uses the generic `transform.apply_registered_pass` transform operation to invoke MLIR passes. We compare MLIR's built-in pass manager system with the Transform dialect by running the identical pass

8 https://www.mlplatform.org/tosa

Model	# Ops	Compile Time (ms)	
		MLIR	Transform
Squeezenet [34]	126	16.6	16.9
GPT-2 [75]	2861	185.4	190.0
Mobile BERT [94]	4134	316.7	317.7
Whisper (decoder only) [74]	847	457.5	462.3
BERT-base-uncased [18]	1182	1315.3	1348.6

Table 6.1: ML models converted to TOSA from TensorFlow using `flatbuffer_translate -tflite-flatbuffer-to-mlir` and `tf-opt -tfl-to-tosa-pipeline`. The use of the Transform dialect introduces $\leq$ 2.6% compile time overhead.

```
mlir-opt --pass-pipeline=builtin.module(
  func.func(tosa-optional-decompositions),
  canonicalize,
  func.func(tosa-infer-shapes,tosa-make-broadcastable,
            tosa-to-linalg-named),
  canonicalize,
  func.func(tosa-layerwise-constant-fold,
            tosa-make-broadcastable),
  tosa-validate,
  func.func(tosa-to-linalg,tosa-to-arith,tosa-to-tensor),
  linalg-fuse-elementwise-ops,
  one-shot-bufferize)
```

Figure 6.11: Standard MLIR pass pipeline for converting IR of the TOSA dialect to the Linalg dialect.

pipelines, which is the worst-case scenario for the Transform dialect, as we do not make use of any of its useful features to precisely control the compilation process – instead, we are measuring the pure overhead.

Our measurements in Table 6.1 and Figure 6.12 show that the Transform dialect introduces very little compile-time overhead, up to 2.6% compared to the default pass pipeline. Most of the passes used in these compilation flows are relatively simple, we expect even less compile time overhead when more complex passes such as register allocation are invoked.

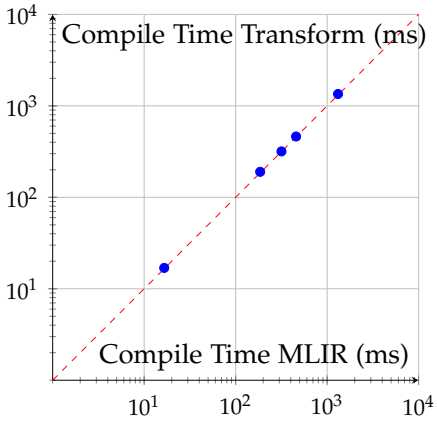


Figure 6.12: Compile time comparison of different machine learning models using unchanged MLIR vs using transform dialect to drive the compilation process.

6.5.2 *Case Study 2: Building robust pipelines for lowering a soup of dialects*

In this case study, we are interested in highlighting the importance of pre- and post-conditions for building robust and flexible compilation pipelines in MLIR.

As we discussed in Section 6.3, compiler transformations often have specific assumptions on their input, which are not explicitly expressed or checked. Because of that, when exploring possible pass pipelines, developers often encounter compiler errors, which are hard to relate to the underlying problem of an invalid pass order.

In this case study, we specifically investigate programs represented using the MLIR compiler infrastructure that are composed of a mix of different dialects. For instance, arithmetics are represented using the `arith` dialect, indexing using the `index` dialect, memory using the `memref` dialect, loops using the `scf` dialect, and functions using the `func` dialect. Listing 6.3 shows a simple program that leverages all of these dialects.

This function accepts a reference to a two-dimensional array in memory (`memref`) and creates a 4x4 rectangular *view* of a part of it at the given offset and potentially with strides. All values in the view are then set to the value 42. Such (sub)views support access with local indexing without any additional index computations and modifications of the loop.

Lowering this simple IR for execution already requires running specific passes to progressively lower these dialects to LLVM. An example of a minimal pass pipeline is:

```
func.func @chunkTo42(A: memref<64x64xf64>) {
  %chunk = memref.subview %A[/*offsets=*/ 0, 0]
    [/*sizes=*/ 4, 4][/*strides=*/ 1, 1] :
    memref<64x64xf64> to memref<4x4xf64, ...>
  %value = arith.constant 42.0
  scf.forall (%i, %j) = (0, 0) to (4, 4) {
    memref.store %value, %chunk[%i, %j]
  }
}
```

Listing 6.3: Example MLIR program using multiple dialects to set values in a subview of memory to a constant.

① `convert-scf-to-cf`
Lower structured control flow (`scf.forall`) to basic blocks and branching instructions.

② `convert-arith-to-llvm`
Lower arithmetic operations to their LLVM dialect counterparts.

③ `convert-cf-to-llvm`
Lower control flow (e.g. `cf.br`) to LLVM counterparts.

④ `convert-func-to-llvm`
Lower functions to an LLVM compatible format (e.g. return multiple values as a structure).

⑤ `expand-strided-metadata`
Externalize non-trivial addressing from memrefs.

⑥ `finalize-memref-to-llvm`
Lower trivially indexed memrefs to LLVM pointers.

⑦ `reconcile-unrealized-casts`
Eliminate temporary type cast operations introduced by previous passes when possible.

Unfortunately, this pipeline fails as soon as we slightly change the input program by having the view created at the non-zero `%offset : index` provided as an additional function argument. The compiler reports that it failed to legalize a *builtin.unrealized_conversion_cast* operation that was explicitly marked illegal. This message does not point towards a solution, making the user resort to a painstaking inspection of the IR after each pass.

This error is due to an `affine.apply` operation that has been introduced during the `-expand-strided-metadata` pass. It is used to model the indexing behavior of the now slightly more

complex `memref.subview` operation. The following pass ⑥ cannot lower the `affine.apply` operation accepting `index` types to LLVM dialect types, so it inserts type casts around it, assuming it will be handled by a successor pass that would insert the reverse casts. In its absence, the final pass ⑦ cannot remove these type casts that do not cancel out and reports the error.

Even this close IR inspection may be surprising to the user since they did use the `affine` dialect, mostly used to represent constructs amenable to the polyhedral model [20, 60], in the input program. Moreover, this dialect may be seen as operating on a conceptually higher level than memory address indexing due to an overly simplistic interpretation of MLIR memory references as mere pointers, and thus unexpected to be produced by a *lowering* pass. This example illustrates the challenge of composing robust pass pipelines.

To solve this problem ad-hoc, we can add the `lower-affine` pass after `-expand-strided-metadata`. This requires additional lowering passes to handle all `affine` dialect operations, including another application of ②.

In the Transform dialect, we address this issue using pre- and post-conditions encoded in the corresponding transform operations and types, as shown in Table 6.2. Pre-conditions indicate the payload operations that are expected in the input and will be removed, other operations will not be modified. Post-conditions indicate new payload operations that will be produced.

Transform Operation	Pre-conditions	Post-conditions
① `convert-scf-to-cf`	{*scf.∗*}	{*cf.*{*branch, cond_branch*}, *arith.*{*addi, cmpi, ...*}, *cast*}
② `convert-arith-to-llvm`	{*arith.∗*}	{*llvm.*{*add, fadd, bitcast, fdiv, sdiv, udiv, ...*}, *cast*}
③ `convert-cf-to-llvm`	{*cf.∗*}	{*llvm*{*func, br, call, cond_br, switch, unreachable*}, *cast*}
④ `convert-func-to-llvm`	{*func.∗*}	{*llvm.*{*alloca, call, constant, func, load, store, undef, ...*}, *cast*}
⑤ `expand-strided-metadata`	{*memref.∗*}	{*memref.*{*subview.constr*}, *llvm.*{*load, ...*}, `affine.apply`}
⑥ `finalize-memref-to-llvm`	{*memref.subview.constr*}	{*llvm.*{*add, alloca, br, call, constant, load, ptrtoint, ...*}, *cast*}
⑦ `reconcile-unrealized-casts`	{*cast*}	{}

Table 6.2: Pre-/post-conditions conditions indicate payload operations removed/introduced by a transform, or additional constraints from Figure 6.6. The *affine.apply* operation potentially introduced by ⑤ is not removed by any following pass. Thus the final IR is {*llvm.∗, affine.apply*} and not only LLVM dialect.

Given the final condition of only using the LLVM dialect, {*llvm.∗*}, our static checking tool reports an error in this pipeline

as it identifies that an *affine.apply* operation produced by ⑤ will remain after the pipeline. The pre- and post-conditions of the Transform dialect support users to develop robust lowering pipelines that are known to work for all possible inputs.

6.5.3 *Case Study 3: Debugging Performance Problematic Optimization Patterns*

Besides ensuring that pipelines are correctly producing code, performance engineers often also have to deal with the counter-productive effects of program optimizations, chasing and eliminating them.

For instance, while introducing additional peephole optimization patterns for StableHLO, which is a part of the Enzyme AD workflow, we observed that a combination of over 100 work-reducing and enabling transformations yielded counter-productive results in one of the LLMs we were trying to optimize, with up to 9% overall performance penalty compared to the JAX/XLA baseline. However, these optimization patterns are designed either for obvious work reduction (e.g., not adding tensor elements produced by padding with zero) or to enable other transformations (permute tensor transpose to enable it to be folded into a matrix multiplication that supports transposed operands).

In order to identify which of the individual patterns was counter-productive, we attempted to perform a binary search over the pattern set. Since the set of optimization patterns is expressed in C++, it required manual code modifications and up to 10 minutes per individual pattern for a fresh compilation, linking and packaging on a 4x24-core Xeon Platinum 8160 (Skylake SP) platform with 196 GB RAM using LLVM.[9] While re-compilation time of a single file is negligible, linking and packaging a 5.4 GiB self-contained tool that includes parts of TensorFlow, LLVM and a Python interpreter (common for production-oriented "hermetic" builds) consumes most of the time: 31s for linking and 164s for compressed packaging. To alleviate this, we leveraged the Transform dialect support for pattern application by associating each pattern with a transform operation, as shown in Listing 6.4.

This allowed us to only deploy the compiler once and to automate the binary search over the pattern set by simply removing

9 LLVM toolchain `3b5e7c83a6e` from Mar 14, 2024

```
transform.apply_patterns to %func {
  transform.pattern.add_of_zero_pad
  transform.pattern.negate_of_transpose
  transform.pattern.matmul_of_transpose
  // more patterns
}
```

Listing 6.4: Transform dialect representation of optimization patterns enabling quick exploration without compiler rebuilds.

parts of the pattern list in the Transform script, with each iteration of binary search taking up to 4 seconds for compilation of the model. Due to this, we identified the counterproductive transformation as "fold reshape/transpose into full reduce". While the full additive reduction of a tensor into a scalar can be implemented effectively regardless of the tensor shape (assuming associativity of floating point addition, e.g., `-ffast-math`, as is common for ML workloads), removing the leading reshape/-transpose operations strictly reduces work. However, in this case, this adversely affected the fusion heuristic in the back-end XLA compiler[10] that produced larger, less cache-efficient fusion clusters.

This case study demonstrates the usefulness of flexible and fine-grained compiler control that enables quick exploring of the space of transformations, in this instance, to detect a performance problem that affects only a single but important input program.

6.5.4 *Case Study 4: Fine-Grained Control of Performance Optimizations*

Ultimately, we are interested in building compilers that generate high-performance code. The Transform dialect enables fine-grained control to specify compiler transformations, e.g., here for a single loop nest of a layer of the Resnet-50 model [31].

In this case study, we discuss an existing alternative in the form of OpenMP pragmas for controlling compiler optimizations. We show the limitations of OpenMP and where the Transform dialect allows us to go further by controlling and composing arbitrary compiler transformations.

OpenMP provides directives to perform loop transformations that can have significant performance benefits, such as parallelization or vectorization. These transformations also include

10 https://openxla.org/xla

tiling, as shown in Figure 6.13. Such transformations are naturally equally available in the Transform dialect. Unlike OpenMP, however, the Transform script is not written directly together with the computational program. Instead, we explicitly match the "for" loop, as in line 2 of Figure 6.14, which we then tile in line 4.

Since the trip count of the loop along `i` (196) is not divisible by the corresponding tile size (32), tiling will result in conditionals being introduced into the loop body, or alternatively, the last iterations are being peeled off. Having explicit handles in the Transform dialect allows us to precisely control this behavior. We first split the loop into the divisible part and the remainder in line 3 and gain a handle for both of these loops, which are not visible in the original code. We then unroll only the remainder loop, named `%rest`, in line 9. Using OpenMP, we only have a limited way of composing transformations, so applying a transformation to a nested loop resulting from tiling is not possible. We measured the performance of the generated code optimized via OpenMP and Transform and observed almost identical performance, with the OpenMP version having a median runtime of 0.48 seconds and the Transform version of 0.49 seconds.

```
1 for (int b=0; b<6; b++) {
2   #pragma omp tile sizes(32, 32)
3   for(int i=0; i<196; i++) {
4     for(int j=0; j<256; j++) {
5       for(int k=0; k<2305; k++) {
6         C[b,i,j] += A[b,i,k] * B[b,k,j];}}}}
```

Figure 6.13: OpenMP pragmas for tiling a single loop nest.

While OpenMP is limited to performing a fixed set of loop optimizations with the Transform dialect, we can go further. We introduce a new transform replacing a small fixed-size matrix multiplication, such as that formed by inner loops after tiling, with a call to a microkernel library [32]. This new specialized transform is easily added into the compiler using the MLIR plugin mechanism and mixed with other transformations, as shown in line 7 of Figure 6.14. We furthermore wrap this transformation into the `alternatives` construct in lines 6–8, as it may fail when the microkernel library doesn't have an implementation with the required sizes. Here we give no additional alternative, so if the replacement with a library call fails, the input code remains unchanged. With the optimized microkernel, we achieve a significantly better performance of 0.017 seconds, over 20 times

```
1  transform.named_sequence @transform_main(%module) {
2    %i_loop = match.op {second} "scf.for" in %module
3    %main, %rest = loop.split %i_loop div_by 32
4    %tiled:2 = loop.tile %main
5                {tile_sizes=[32, 32]}
6    transform.alternatives {
7      transform.to_library %tiled#2 "libxsmm"
8    }, { }
9    loop.unroll %rest {full}
10 }
```

Figure 6.14: Transform script performing in lines 2–5 the same optimization as the OpenMP version above. In line 7, the script attempts to replace the nested loop code with a library call resulting in a significant performance win.

faster than the tiled versions. This small experiment showcases the additional possibilities the Transform dialect opens to integrate highly specialized optimizations, such as replacements with dedicated library calls, over more rigid alternatives such as OpenMP.

6.5.5 *Case Study 5: Performance Exploration with State-of-the-Art Autotuning Methods*

```
1  transform.named_sequence @transform_main(%module) {
2    %batch_loop = match.op {first} "scf.for" in %module
3    %tiled:4 = loop.tile %batch_loop
4                {tile_sizes=[tile0,tile1,tile2,tile3]}
5    transform.alternatives {
6      transform.to_library %tiled#4 "libxsmm"
7    }, {
8      transform.assert vect
9      transform.vectorize %tiled#4
10   }, { }
11   loop.unroll %rest {full}
12 }
```

Figure 6.15: Transform script with the parametric tile sizes that are automatically chosen using an autotuning tool.

In this case study, we demonstrate how Transform scripts facilitate optimization space exploration. We employ BACO [33], a state-of-the-art Bayesian autotuning tool, to automatically search for optimal tiling configurations. The search applies to the loop nest from Section 6.5.4, using the Transform script in Figure 6.15 with tuning parameters and constraints as shown in Figure 6.16. Figure 6.17 shows the performance evolution of the optimization

process. The search gradually finds better and better values for the tile size parameters reaching a final speedup of 1.68.

```
tuning_parameters: [
  tile0: {range:[0, B], constraints:[B % tile0 == 0]}
  tile1: {range:[0, M], constraints:[M % tile1 == 0]}
  tile2: {range:[0, N], constraints:[N % tile2 == 0]}
  tile3: {range:[0, K], constraints:[K % tile3 == 0]}
  vect:  {range:[0, 1], constraints:[
    where(tile3 % vector_size != 0, vect == 0)]} ]
```

Figure 6.16: Definition of the tuning parameters used in Figure 6.15 with constraints: tile sizes must divide their dimension, and vectorization is disabled if the trip count of the innermost loop is not divisible by the machine vector size.

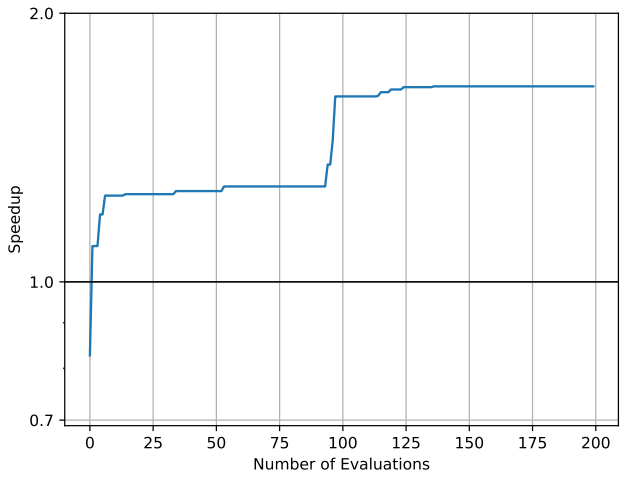


Figure 6.17: Performance evolution of batch matrix multiplication when searching for the best tile size parameters in the Transform script specifying the optimization.

This case study shows how easy it is to integrate Transform with autotuning and similar machine-learning driven exploration tools to quickly explore optimization spaces.

6.6 CONCLUSION

We presented and evaluated the Transform dialect, demonstrating how it can fundamentally change the way we structure compiler optimization pipelines. Moving beyond traditional pass managers, this system enables direct, fine-grained control over transformations while maintaining MLIR's extensible nature.

Through five case studies, we validated that this tight integration with MLIR not only maintains low overhead but enables the construction of robust and flexible optimization pipelines by exposing and orchestrating previously hidden compiler transformations.

The Transform dialect represents a significant step toward making compiler optimizations more accessible and controllable. By providing a unified interface to MLIR's transformation capabilities, it enables performance engineers to compose sophisticated optimization strategies without requiring deep compiler expertise or implementation of new passes. The approach is now part of upstream MLIR, where it continues to evolve with community support, and our case studies are available as peer-reviewed artifacts. This demonstrates that compiler transformations can be exposed and orchestrated systematically while maintaining the robustness and efficiency expected from production-quality compiler infrastructure.

The strength of the Transform dialect lies in its deep integration with MLIR's infrastructure, providing practical access to the framework's transformation capabilities. The following chapter explores an alternative approach with Elevate in MLIR, which investigates transformation composition from a more principled foundation. Together, these approaches help us understand the design space for transformation systems in modern compilers.

7

A PRINCIPLED REWRITING SYSTEM FOR MLIR

This chapter is partially based on "Sidekick compilation with xDSL" by Fehr, Weber, Ulmann, Lopoukhine, Lücke, Degioanni, Steuwer, and Grosser published at CGO'25, where I contributed the principled rewriting system presented here.

Modern compiler infrastructures like MLIR offer powerful transformation capabilities, as demonstrated by the Transform dialect in the previous chapter. While the Transform dialect provides a practical solution for scheduling compiler transformations, its implementation relies on a complex set of features, including cloning mechanisms and special cases for different transformation types. Moreover, the semantic meaning of transform operations varies significantly—from simple rewrites to complex analysis passes—making it challenging to reason about their correctness or provide formal guarantees about transformation sequences.

This chapter explores an alternative, more principled approach by integrating Elevate, a rewriting system built on the core concept of composition. Unlike the Transform dialect's feature-rich but pragmatic approach, Elevate achieves similar capabilities by composing a minimal set of rewriting primitives into complex transformations. This compositional foundation not only reduces implementation complexity but also provides a pathway to formal verification of rewrites and transformation sequences.

7.1 INTRODUCTION

Program transformations are fundamental to modern compiler optimization, yet effectively composing and managing complex transformations remains challenging. More recent approaches like Halide offer scheduling APIs to control optimizations but fail to do this in a way that lets users build their own abstractions and compose these APIs to model new optimizations.

Elevate, a language designed specifically for program transformations, was developed to address this challenge. It introduces a small set of built-in primitives that are composed to form more intricate transformations. In practice, Elevate complements functional languages like Rise, where computations are expressed and then optimized by composing rewrites.

However, the integration of such compositional rewriting systems with modern compiler infrastructures presents a significant challenge. The root problem lies in the divergent approaches to managing intermediate representations during the optimization process. Elevate is designed to work in tandem with systems such as Lift and Rise, which operate on an immutable AST (Abstract Syntax Tree) structure, leveraging efficient copy-on-write techniques. This approach allows a rewrite to fail at an arbitrary point and seamlessly continue with an alternate rewrite at the previous AST state via backtracking. This enables Elevate's key feature: systematic exploration of different optimization sequences through backtracking. In contrast, MLIR employs a destructive rewriting approach, directly modifying the IR without providing a built-in mechanism for retaining intermediate state.

Our solution bridges this gap through xDSL [22], a compiler infrastructure that exposes MLIR's core concepts through a flexible Python interface. Rather than modifying MLIR directly, we leverage xDSL to implement an immutable IR design that supports Elevate's rewriting approach while maintaining compatibility with MLIR's ecosystem. This design preserves program state across transformations through efficient copy-on-write mechanisms, enabling safe exploration of different optimization sequences.

The resulting system combines the strengths of both approaches: Elevate's principled composition of transformations and MLIR's extensive dialect ecosystem. Programs can seamlessly transition between MLIR's traditional optimization passes and our compositional rewriting system, enabling sophisticated optimization techniques like backtracking search while preserving access to MLIR's full range of transformations.

7.2 BACKGROUND: DIVERGENT APPROACHES TO PROGRAM TRANSFORMATION

MLIR implements a hierarchical SSA IR optimized for use in production settings. This representation is implemented predominantly mutable such that, for instance, the operands of an operation can be modified freely by an MLIR transformation. While the attributes field of an operation is mutable through the support of addition and removal, attributes themselves are immutable objects. This enables hashing and uniqing of attributes and thus comparing attributes of operations through a simple

pointer comparison to improve compile time performance [72]. With this exception, MLIR predominantly operates on a mutable IR structure.

The mutable nature of MLIR's IR structure has several important consequences for transformations. Primarily, it allows for direct modifications of the existing IR structure without the need for creating new copies, leading to more efficient memory usage. Optimizations can perform in-place updates to operations, blocks, and regions, which is particularly beneficial when designing non-local transformations that affect big portions of a program. Additionally, MLIR supports transformations that temporarily break certain IR invariants and restore them later in the transformation process.

It's worth noting that MLIR's mutable IR structure, while efficient for successful transformations, introduces complexities in managing failed or partially completed transformations. In scenarios where a transformation attempt might not succeed, the mutable nature of the IR can lead to partial modifications that can not be restored automatically, as MLIR does not provide a rollback process out of the box. To mitigate this, MLIR transformations require careful implementation to ensure proper handling of failure cases before starting any modifications. To avoid such problems, the MLIR rewriting system is focused on supporting small individual transformations that are applied at all possible locations. Thus, through this foundation, MLIR reinforces the dependence on difficult-to-customize fixed sequences of optimizations.

Exploring different possible optimizations for a single program is challenging in this system, as augmenting it with flexible state recovery after the fact presents substantial practical challenges. Due to the destructive nature of transformations, reverting to a previous state typically requires explicit copies of the IR.

For instance, consider a scenario where multiple optimization sequences need to be evaluated. Each sequence might require a separate copy of the entire IR, leading to significant memory overhead, especially for large-scale programs. This is memory-intensive and impractical for large programs, limiting the efficient exploration of alternative transformation sequences. Due to the destructive nature of transformations, reverting to a previous state typically requires explicit copies of the IR. For instance, consider a scenario where multiple optimization sequences need to be evaluated. Each sequence might require a

separate copy of the entire IR, leading to significant memory overhead, especially for large-scale programs. This approach is not only memory-intensive but also computationally expensive thus limiting the efficient exploration of alternative transformation sequences. The Transform dialect partially addresses this challenge through its `transform.alternatives` operation, which explicitly clones parts of the IR to enable reverting to previous states when trying different transformation sequences. While this provides a practical solution for exploring alternatives, it still requires careful management of IR copies and can be memory-intensive for large programs.

Elevate represents a more principled solution to these challenges through strategic rewriting. The foundation for this is the Elevate Strategy, a function to encode program transformations with the following type: `type Strategy = p : P => (Success(p' : P) | Failure)`, with `P` as the type of the rewritten program. The result of a strategy on successful application is `Success` containing the new rewritten program `p'` or `Failure` otherwise. For both options, the original input `p` is not modified.

This structure is clearly designed to operate on an immutable representation such as a Rise AST. Its immutable nature also simplifies the implementation of complex rewriting strategies, as each rewrite step can be treated as a pure function, producing a new IR state without side effects.

This facilitates easier exploration of multiple optimization paths without the need for explicit state management.

Elevate is designed to be extensible, allowing users to encode their own rewrites as strategies, build their own abstractions, and compose strategies to target specific domains or optimization goals. Elevate's approach addresses some of the challenges faced by systems like MLIR, particularly in the realm of exploring diverse optimization strategies. By maintaining immutability and providing high-level abstractions for expressing transformations, Elevate offers a flexible platform for experimenting with and composing complex program optimizations. This design philosophy enables developers and researchers to express, combine, and reason about optimization strategies in a more modular and verifiable manner [25, 73].

While Elevate offers a powerful framework for expressing and applying rewrites of programs, its approach faces significant challenges when faced with the structure of general-purpose compilers. The fundamental assumptions underlying Elevate's design do not align with the requirements and constraints of

current compilers. One major challenge lies in the prevalence of side-effecting transformations in traditional compiler infrastructures. Unlike the purely functional approach, many established compilers rely on in-place modifications of the IR for efficiency. These side effects are often deeply ingrained in the compiler's architecture, making it difficult to directly apply Elevate's immutable rewrite model.

The differing paradigms also extend to the way optimizations are sequenced and applied. While Elevate emphasizes composable, fine-grained rewrite rules, many general-purpose compilers employ coarse-grained, monolithic optimization passes that operate on the entire program simultaneously. This fundamental difference in granularity presents significant challenges when attempting to integrate Elevate's strategies into existing compiler frameworks.

Moreover, the performance expectations and resource constraints of production compilers often prioritize compilation speed and memory efficiency over the flexibility offered by Elevate's approach. The overhead introduced by maintaining immutable program representations and exploring alternative paths for optimization may not be acceptable in scenarios where rapid compilation is crucial.

7.3 BRIDGING ELEVATE AND MLIR: AN INTEGRATED APPROACH

The design philosophy of Elevate relies on specific guarantees from the underlying compiler infrastructure. Two key requirements for effective integration with the underlying infrastructure are:

LOCAL REWRITES The infrastructure must ensure that each rewrite operation affects only its intended target within the IR. This isolation prevents unintended side effects and maintains the integrity of the transformation process. MLIR has no support for prohibiting modifications to unrelated locations.

STRATEGY FAILURE RECOVERY The failure of strategies is not a special case in Elevate but an essential requirement for its modeling of composition. Hence, the infrastructure needs to implement robust mechanisms to handle and recover from failures. As the failure of strategies is prevalent the performance of the overall rewriting system strongly depends on the imple-

mentation of failure recovery. MLIR rewrites may fail after and there are no features available in MLIR to undo these modifications. MLIR Rewrites are typically designed to check the requirements of a rewrite before starting any modifications, but in the context of Elevate, this is not sufficient. Strategies are generally composed of a multitude of rewrites, any of which might fail. Consider the scenario of a rewrite that fails before modifying the IR, which follows other successful rewrites that already modified the IR. This still yields a strategy that fails at an intermediate point with partial modifications that need to be rolled back. MLIR offers the option to clone operations such that the original operation could be recovered on the failure of a rewrite. This approach requires multiple expensive copies of operations, even in the context of simple strategies on small programs.

Modifying the existing MLIR foundation to satisfy both of these requirements represents a significant engineering effort. Instead, we focus on bridging the two systems by designing a new immutable IR for MLIR.

7.3.1 *An immutable hierarchical IR for MLIR*

The transition from a mutable IR to an immutable one in an industrial compiler such as MLIR requires extensive and pervasive modifications to the existing framework. This redesign affects not only the core data structures but also the entire ecosystem of passes, analyses, and optimizations built upon them. The inherent mutability permeates through various layers of the compiler, from front-end parsing to back-end code generation. Implementing immutability requires redesigning memory management strategies in the compiler and restructuring optimization passes to operate on new versions of the IR rather than modifying them in place. This paradigm shift will also impact performance characteristics, potentially introducing overhead in memory usage and computation time.

Hence, modifying the IR structure of MLIR for a research prototype is not a practical approach to provide a foundation for Elevate's strategic rewriting approach. Instead, we leverage the xDSL [22] compiler infrastructure. xDSL exposes the main MLIR concepts and high-level design decisions in a Python interface. Thus, it enables the easy implementation and modification of core MLIR constructs while leveraging Python's productivity.

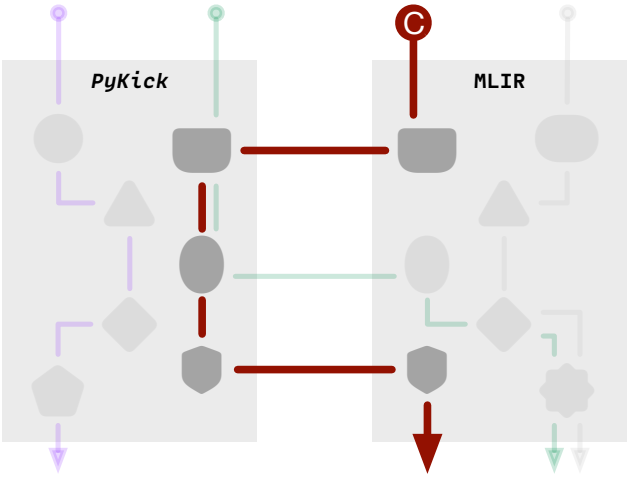


Figure 7.1: Integrating Elevate-based rewriting with MLIR using xDSL. The compilation process flows through MLIR dialects, transitions to xDSL for Elevate's strategic rewriting on the immutable representation, and then seamlessly returns to MLIR for further processing.

This makes experimentation with different IR design approaches in the context of MLIR practical. Instead of modifying numerous C++ files per approach, with little opportunity for reuse between completely different designs and handling a complicated build system, in xDSL, the core IR can be flexibly switched by extending or replacing a single Python file. xDSL empowers researchers to switch between different core IR designs using a flag to flexibly benchmark them. Furthermore, xDSL seamlessly interoperates with MLIR through the textual format. This interoperability enables performing select passes in xDSL while executing the rest in MLIR, allowing for a hybrid approach to optimization as shown in Figure 7.1. Since xDSL can flexibly interact with MLIR in this manner, it does not limit benchmarking the design on real-world problems. For these reasons, we leverage xDSL to implement and experiment with novel IR designs that support the requirements of Elevate while maintaining compatibility with the broader MLIR ecosystem.

For each mutable IR structure, we create an immutable counterpart using Python's frozen dataclasses. This approach ensures that IR objects cannot be altered after creation, guaranteeing that any rewrite results in new IR instances rather than in-place modifications. To maintain interoperability with existing xDSL components, such as analyses and visualization tools, we provide functions for converting between mutable and immutable representations.

The MLIR IR design incorporates numerous "back edges", where child nodes reference their parents. This structure poses a challenge for rewriting immutable representations, as replacing a node necessitates the replacement of all nodes that reference it to maintain consistency. In the worst case, these updates cascade through the whole IR and induce a complete replacement. To address this, we experimented with removing these back edges to parents in our immutable IR design. This approach offers a more performant rewriting system, as it reduces the cascading effect of modifications. However, it comes at the cost of making certain rewrites more challenging to express, particularly those that rely on parent context.

Upon further investigation, we ultimately adopted the solution of an automatic update mechanism that allows for quiet, transactional updates of these back edges. This approach maintains immutability for all practical purposes while efficiently handling parent references. Our system thus balances the performance benefits of an immutable structure against the expressiveness required for complex transformations while addressing the practical challenges of IR manipulation in the context of strategic rewriting systems.

7.3.2 *A New Rewriting APIs to increase reuse*

To facilitate the expression of rewrites that minimize the generation of new IR constructs, we introduce the `from_op` rewriting API. This approach is common in functional programming and allows the creation of new operations on the basis of an existing operation that reuses most of the original operation's components, changing only the specific elements passed as arguments. To illustrate this approach, we express the rewrite rule for commuting the operands of an addition operation as follows:

```
match addOp(op_type=addi | addf, operands=[lhs, rhs]):
    return from_op(addOp, operands=[rhs, lhs])
```

This expression preserves all types, attributes, and uses from the original matched addition operation, modifying only the operand order. This approach lifts the user from the responsibility to check whether the original operation contained any attributes that need to be preserved. Furthermore, this rewrite is polymorph over all possible operand and result types that are valid for addition operations. For instance, this rewrite works for the addition of two 32-bit integers and for the addition of two 16-bit floats as well. This approach simplifies the expression

of rewrites and facilitates the reuse of existing IR components, thereby improving memory efficiency.

7.3.3 *Traversal Strategies: Navigating MLIR's IR structure*

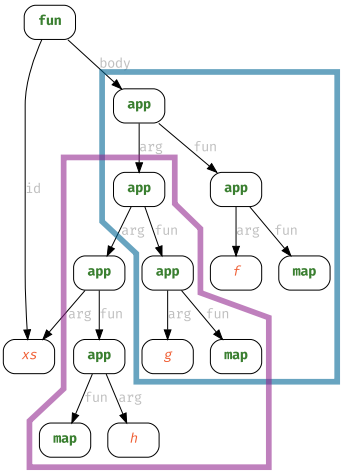
Generally, an Elevate strategy is applicable at multiple locations in a program and only a specific application yields the most beneficial performance results. For instance, the Rise program `threemaps` shown in Figure 7.2a contains two possible locations where two maps could be fused, either the first two (highlighted in blue) or the last two (highlighted in purple). Hence, programmers require a way to precisely specify where a strategy is to be applied.

PRECISELY CONTROLLING APPLICATION OF STRATEGIES For this reason, Elevate introduces special *traversal* strategies that enable applying a strategy at a specific location in the program AST. Formally, traversals are strategy transformers of type `Traversal[P] = Strategy[P] => Strategy[P]` that accept a strategy as an argument and apply it at specific locations. Inspired by [101], Elevate introduces the traversals `all` and `one`, which apply a given strategy to all or one sub-expression respectively and fail if this is not possible. In the example, using the `one` traversal twice enables the fusion of the last two maps: `one(one(map-fusion))(threemaps)`.

Furthermore, Rise-specific traversals are introduced to enable more precise specification of the AST location of strategy application. As Rise leverages a λ-calculus representation, they introduce strategies to traverse function abstraction and application: `body`, `function`, and `argument`. They enable traversing the Rise AST along the arrows shown in Figure 7.2a. With `body(arg(map-fusion))(threemaps)` we leverage these to build another possible traversal to apply `map-fusion` to the last two maps.

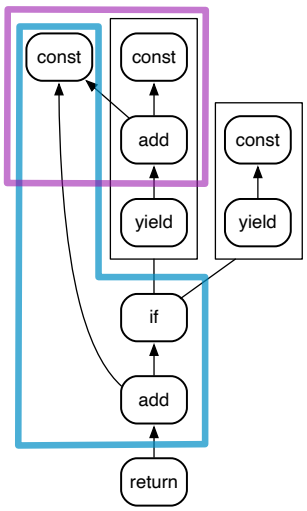
REBUILDING THE AST A Strategy, by definition, returns a replacement for the AST node it was applied to. When a traversal applies a strategy to a child node, it expects to receive a modified version of that node as a result. This modified node then replaces the original in the AST structure.

Traversals like `all`, `one`, `body`, `function`, and `argument` each have specific rules for selecting which child nodes to target. When a traversal is applied, it attempts to use the given strategy




```
fun(xs ⇒ map(f)(map(g)(map(h)(xs))))
```

(a) Two possible locations for applying the *map-fusion* strategy within the same program represented as Rise AST. [28]



(b) Two possible locations for applying the *commute-add* strategy within a simple program represented in MLIR shown in Listing 7.1.

Figure 7.2: Target representation-specific traversals are required to precisely specify the application location of strategies.

on these selected nodes. If the strategy application is successful, the traversal rebuilds the original AST node it was apply to and integrates the resulting modified child node into the appropriate position.

The composition of multiple traversals creates a chain where each subsequent traversal works on the result of the previous one. This chaining allows for the construction of complex transformations that can target very specific locations within the AST. The final output of this process is a new AST that incorporates the effects of the applied strategy at the precise location specified by the composed traversals.

Consider the example of fusing the last two maps using the composition `body(arg(map-fusion))(threemaps)`. Here, the `body` strategy is applied to the topmost `fun` and applies the rest of the strategy to its body and is responsible for building a new `fun` with the replacement for the body. Equally, the `arg` traversal is applied to the topmost `app` node in the AST, applies the rest of the strategy (map-fusion) to the argument, and is responsible for building a replacement `app` node with the result of the `map-fusion` strategy.

REDESIGNING TRAVERSALS FOR MLIR In contrast to Rise's ability to efficiently leverage its simple tree-based AST structure for traversals, MLIR presents a more complex landscape for traversal design. MLIR's hierarchical nature, characterized by nested structures and control flow graphs, necessitates a broader approach to traversals.

```
1  %a = arith.constant  42
2  %cond = arith.constant true
3  %b_or_c = scf.if %cond {
4    ^bb_then:
5    %b = arith.constant 2
6    %ab = arith.add %a, %b
7    scf.yield %b
8  }, {
9    ^bb_else:
10   %c = arith.constant 3
11   scf.yield %c
12 }
13 %result = arith.add %a, %b_or_c
14 return %result
```

Listing 7.1: MLIR program demonstrating multiple potential locations to apply the commute-add rewrite: One nested within an if-region and another at the top level, illustrating how precisely targeting a location requires navigation of both SSA def-use chains and nested region boundaries.

For instance, the MLIR program shown in Listing 7.1 contains two possible locations for application of the `commute-add` strategyFigure 7.2b illustrates the program's structure through two types of relationships: arrows represent def-use chains between operations (showing how values flow through the program), while the black squares indicate region nesting (showing how operations are hierarchically contained within other operations). This dual representation highlights why simple def-use traversal is insufficient: while both addition operations (line 6 & 13 in Listing 7.1) use the value `%a`, the inner addition is only reachable by first traversing into the if operation's region.

As demonstrated, hierarchical SSA IR employed by MLIR is more heterogeneous than the Rise AST representation and thus requires a broader approach to traversals. It is not sufficient to limit traversals to the def-use relations of operations as the representation leverages several other types of connections between IR elements. Firstly, operations may contain an arbitrary, not necessarily statically known, number of operands which we model using the traversal `operand` parameterized with an index. Operations may contain any number of regions which each consist of a number of blocks. Each block contains a list of operations and may branch to other blocks in the same re-

gion. As users require precise control on traversing all of these structures, we introduce different types of *nested traversals*:

- `OpsTraversal` - Models how a given strategy is applied to the operations in a block. On success returns a replacement for the block it was applied to. An example is `opN`, which applies the `s` to the first operation in the block.
- `BlocksTraversal` - Models how a given `OpsTraversal` it to be applied to the blocks of a region. On success returns a replacement for the region it was applied to. An example is `blockN`, which applies `opsT` to the first block in the region.
- `RegionsTraversal` - Models how a given `BlocksTraversal` is to be applied to the regions of an operation. On success returns a replacement for the operation it was applied to. An example is `regionN`, which applies the `blocksT` to the first region of the operation.

Fully composing instances of all types of nested traversals yields a traversal as defined by Elevate that models traversal from a given operation to operations nested into a region of it. For instance, the strategy `regionN(0, blockN(0, opN(0, s)))` applies the given strategy `s` to the first operation of the first block of the first region of the operation it is applied to.

To illustrate how these traversals can be composed to target specific operations, let's consider reaching the addition operation nested within the if-region in line 6 in our example from Listing 7.1 and Figure 7.2b. To apply the `commute-add` strategy to this specific operation, we need to traverse into the first region of the if operation, navigate to the first block (the "then" block) and apply it to the second operation in that block (the addition). We can express this precise traversal path using our nested traversals:

```
regionN(0, blockN(0, opN(1, commute-add)))(mlir_program)
```

This composition applies `commute-add` to the nested addition operation `%ab = arith.add %a, %b`, transforming it into `%ab = arith.add %b, %a`, while leaving the outer addition operation unchanged. To target the outer addition instead, we would simply use `opN` without the region and block traversals, as it appears directly in the top-level block.

Building on this foundation, we introduce the nested traversals `regionOne`, `blockOne`, and `opOne`. We build the more general `one` traversal for MLIR With similar semantics to Elevate as

```
one(s) = (operandOne(s) ; regionOne(blockOne(opOne(s)))}
```

with ; as sequential composition. Effectively, this tries to apply the given strategy s to each operand and each immediately nested operation until it succeeds. We proceed similarly for the `all` traversal.

This modeling allows to reuse the complete program traversals as defined by Elevate:

```
def topDown[P]: (s <+ one(topDown(s)))(p)

def bottomUp[P]: (one(bottomUp(s)) <+ s)(p)

def allTopDown[P]: (s ‘;‘ all(allTopDown(s)))(p)

def allBottomUp[P]: (all(allBottomUp(s)) ‘;‘ s)(p)
```

Notably, we introduce the `blockCFG` BlocksTraversal, which allows for the traversal of blocks not in the order they are ordered into a region, which has no semantics meaning besides the entry block, but through successors. This traversal follows the order of the blocks that will be visited during execution. In the presence of conditional branches, the eventual visiting order depends on runtime values and can not be inferred completely. In this case, the order is approximated by visiting the successor blocks in the order they are stated in branching operations. In contrast to `blockOne`, which traverses blocks in the order they are listed into a region, this traversal might visit fewer blocks because not all blocks are always reachable.

7.4 EVALUATION

To deepen the understanding of the cost, scalability, and generality of rewrite systems, we evaluate our work with respect to the following two research questions:

- **RQ1** Can existing MLIR rewrites be modeled using composable rewrites in Elevate?
- **RQ2** Does composable rewriting scale to large applications?
- **RQ3** How do immutable rewrites impact compile time?

7.4.1 *Case Study: Rephrasing MLIR Transformations as Individual Rewrites*

To evaluate the applicability of our composable rewriting system, we conducted a case study involving the porting of existing MLIR rewrites. This study aims to assess the system's capacity to express common compiler optimizations and identify potential limitations or areas for future development.

We selected a diverse set of MLIR optimizations, ranging from arithmetic expansion that can be applied without further analysis to complex polyhedral optimizations from the affine and linalg dialects that require extensive analysis. We also included classical compiler optimizations that are more independent of specific dialects, such as dead code elimination. This spectrum allows us to evaluate the effectiveness of our system across varying levels of transformation complexity and generality. The porting process involves reimplementing these transformations using our system, including the `from_op` API. We systematically analyzed each transformation, focusing on the challenges encountered during the implementation process and the ability to faithfully represent the logic of the original transformation.

ARITHMETIC EXPANSION Our system proved highly effective in modeling arithmetic expansion rewrites, which transform complex arithmetic operations into sequences of simpler operations. We expressed all 11 rewrites included as part of the arithmetic MLIR dialect. These rewrites represent simple local patterns that match a single operation and can always be applied to this type of operation. For example, converting a floor division operation `%result : !i32 = arith.floordivsi(%0 : !i32, %1 : !i32)` into the operations shown in Listing 7.2

These transformations are particularly well-suited for our system as they require only local context and are applied without additional analysis or global program information.

POLYHEDRAL OPTIMIZATIONS Our attempts to port polyhedral optimizations from MLIR's affine and linalg dialects revealed significant challenges. Despite considerable effort, the existing optimizations proved resistant to portation to our system. The fundamental issue lies in their inherent complexity and deep integration with MLIR's infrastructure. MLIR's implementation of these transformations is tightly coupled with its analysis infrastructure, making it difficult to isolate the trans-

```
%2 = arith.constant 1 : !i32]
%3 = arith.constant 0 : !i32]
%4 = arith.constant -1 : !i32]
%5 = arith.cmpi(%1, %3) "ne"
%6 = arith.select(%5, %2, %4)
%7 = arith.subi(%6, %0)
%8 = arith.divsi(%7, %1)
%9 = arith.subi(%4, %8)
%10 = arith.divsi(%0, %1)
%11 = arith.cmpi(%0, %3) "ne"
%12 = arith.cmpi(%0, %3) "sgt"
%13 = arith.cmpi(%1, %3) "ne"
%14 = arith.cmpi(%1, %3) "sgt"
%15 = arith.andi(%11, %14)
%16 = arith.andi(%12, %13)
%17 = arith.ori(%15, %16)
%result = arith.select(%17, %9, %10)
```

Listing 7.2: Result of applying an arithmetic expansion rewrite to a floor division operation. The original single floor division operation is expanded into a sequence of simpler operations: a regular division, followed by operations to handle the floor rounding behavior through conditional subtraction. This demonstrates how arithmetic expansion rewrites transform complex operations into sequences of more primitive operations.

formation logic into localized strategies. However, we can still incorporate such transformations by treating them as black-box transformations that operate on entire loop nests. While this approach sacrifices the benefits of fine-grained composition of transformations, it allows us to leverage existing, well-tested polyhedral optimizations within our framework.

DIALECT-INDEPENDENT OPTIMIZATIONS Our investigation of dialect-independent passes, such as dead code elimination and common subexpression elimination, revealed interesting insights about our system's capabilities. These passes traditionally operate on a global program view, analyzing properties across multiple operations and blocks. While some of these optimizations initially appeared to require monolithic, whole-program transformations, we found that many could be reformulated as sequences of local rewrites. For instance, dead code elimination can be redesigned as a series of localized decisions by leveraging MLIR's region-based value scoping, which naturally constrains the analysis scope. While this reformulation operates differently and is less performant than MLIR's existing solution, it demonstrates the feasibility of expressing traditionally global optimizations in a localized manner. However, certain

optimizations, particularly interprocedural analyses, remain challenging to express in our framework due to their inherent need for whole-program context. As outlined above, these are still reusable if treated as black-box transformations of the whole module.

To complement our xDSL-based implementation, we designed an approach to integrate our strategic rewriting framework directly with MLIR's existing infrastructure [55]. While our primary implementation leverages xDSL's Python interface for prototyping the immutable IR design, a small additional prototype in MLIR's C++ codebase demonstrates how existing transformations can be integrated as strategies. This prototype uses MLIR's native cloning mechanisms to preserve program versions, providing a simpler but far less efficient alternative to our immutable IR design.

```
1  // C++ MLIR transformation
2  LogicalResult mlirTileLoops(Operation *op) {
3    // ... existing MLIR implementation ...
4  }
5
6  // Strategy wrapper
7  class MLIRTiling : public Strategy {
8  Result apply(Operation *op) override {
9    Operation *clone = op->clone(); // preserve original
10   if (succeeded(mlirTileLoops(clone))) {
11     op->erase();
12     return Success(clone);
13   } else {
14     clone->erase();  // clean up clone, op remains valid
15     return Failure();
16   }
17 };
```

Listing 7.3: Integration of existing MLIR C++ transformations as strategies. The wrapper converts MLIR's boolean success indicators and direct IR modifications into Elevate's Success/Failure strategy results.

This direct C++ implementation demonstrates the core concept: transformations that modify MLIR's IR in place can be wrapped as strategies by cloning the input operation before modification (line 9). The strategy succeeds if the transformation modified the program, returning the modified clone (line 12), and fails otherwise (line 15), cleaning up the unused clone (line 14). Since the original operation `op` remains unmodified in the failure cases, subsequent transformations can continue operating on it, maintaining our strategy composition guarantees. While this approach lacks the efficiency benefits of our immutable IR design, it demonstrates that existing MLIR trans-

formations can be safely integrated into our strategic rewriting framework through this black-box wrapping technique.

```
strategy = TopDown(is("scf.for") ;
  MLIRTiling([32, 32, 32, 32]) ; innermost(is("scf.for") ;
  (MLIRToLibrary <+ MLIRVectorize <+ id)) ; MLIRLoopUnroll)
```

Listing 7.4: Strategy performing loop tiling followed by alternative optimizations (library call generation or vectorization), with loop unrolling as fallback.

In a similar spirit to the transformation sequence in Figure 6.15 expressed using the Transform dialect, we use this wrapping technique to compose MLIR's internal transformation functions into more complex optimization sequences. The strategy shown in Listing 7.4 demonstrates this by combining several MLIR transformations: it first uses the predicate strategy `is` top-down to match an `scf.for` operation, then applies loop tiling with specific tile sizes. Then, it attempts to either convert the innermost loops to library calls or vectorize them (falling back to the identity transformation if both fail) and finally unrolls the remaining loops. As shown in the previous chapter with the Transform dialect, these are common MLIR transformations, but our approach enables their direct composition through strategy combinators while reusing their existing implementations.

In summary, this case study demonstrates that our rewriting system can express a wide range of compiler optimizations without introducing fundamental limitations, as existing transformations can always be integrated as black-box operations when local decomposition proves impractical. Simple, locally-scoped transformations can be expressed with comparable complexity to their original implementations. More complex optimizations reveal a spectrum of applicability: some traditionally global analyses can be reformulated to work within our system's constraints, albeit possibly with performance trade-offs. However, transformations deeply integrated within the underlying infrastructure prove challenging to decompose into local rewrites and must be treated as black-box transformations. This investigation suggests that our approach is most effective for transformations that can be naturally decomposed into local rewrites while still allowing the integration of more complex, monolithic optimizations when necessary. The successful reformulation of dead code elimination into local rewrites suggests interesting opportunities for future work in redesigning other traditionally global analyses into sequences of localized transformations,

enabling their reuse in different contexts, incorporation into search-based optimization strategies, and finer-grained control over their application—for instance, selectively applying specific canonicalization rules rather than running an entire program-wide canonicalization pass.

7.4.2 *A case study: Matching structures in machine learning models*

Machine learning compilers frequently need to identify complex computational patterns in their input graphs to replace them with optimized implementations. A prominent example is the identification of attention layers in transformer-based models. These layers, which consist of multiple matrix multiplications and represent a significant portion of the model's runtime, can be replaced with highly optimized implementations [17, 42].

Figure 7.3 illustrates this optimization process: on the left, we see the attention implementation as it appears in the compiler's high-level IR a complex subgraph of matrix multiplications, transposes, and other operations (highlighted in red). The compiler aims to identify this pattern and replace it with a single optimized Attention operation, as shown on the right. Our analysis of ONNX Runtime's pattern matcher implementation [69] reveals how challenging this identification process becomes with traditional approaches.

The ONNX Runtime codebase shows how a simple 80-line Python implementation that matched an attention pattern grew to over 400 lines as it needed to handle increasingly common variations in attention layer implementations. This growth primarily stemmed from accumulating ad-hoc conditional statements to handle each new pattern variant. The complexity is evident in Figure 7.3, where slight variations in operation order, the presence of additional reshapes, or different approaches to normalization all need to be handled by the pattern matcher.

This pattern matching challenge presents an ideal case study for our strategic rewriting approach. The identification of attention layers requires matching complex sequences of operations while being flexible enough to handle variations in their implementation.

Our system addresses this challenge by composing small reusable pattern matching strategies. At the most basic level, these strategies match individual nodes in the computational graph, such as matrix multiplications or the addition of bias (left out for clarity in Figure 7.3). These atomic matchers serve

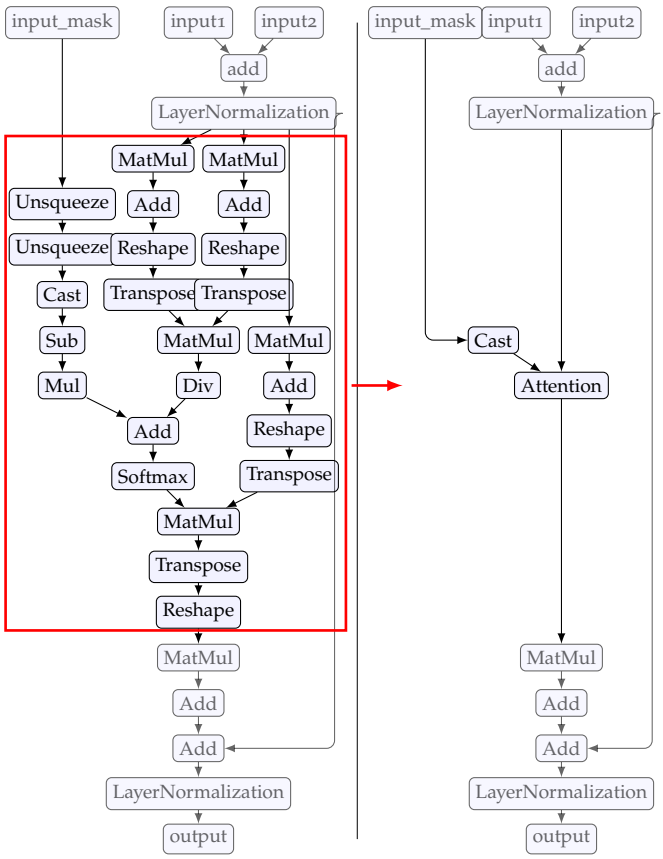


Figure 7.3: Transformer Attention Layer Fusion

as building blocks that can be composed into more complex patterns. We express each variation of an attention layer implementation as a separate composition, with common sub-patterns shared between them.

To create a single comprehensive attention matcher, we compose these individual attention strategies using the leftChoice (infix notation <+) combinator as shown in Listing 7.5. This combinator attempts to match patterns in order, trying each subsequent pattern only if the previous ones fail. This allows us to concisely express a single matcher that identifies any supported attention variant

```
def matchAttention =
    DotProductAttention()
      <+ AdditiveAttention()
      <+ LocationAwareAttention()
```

Listing 7.5: Matching different forms of attention [4, 13, 56]

Each attention variant is defined by composing atomic matchers using the sequential composition operator (;) as shown in

Listing 7.6. These definitions directly reflect the computational structure of different attention mechanisms. For instance, dot product attention follows a simple sequence of matrix multiplications and softmax, while additive attention requires dimension adjustments (squeeze/unsqueeze), linear layers, and a tanh activation, as shown in Listing 7.5.

```
1 DotProductAttention =
2     Matmul() ; Softmax() ; Matmul()
3
4 AdditiveAttention = Squeeze() ; Linear(tanh(add(Linear(), Linear(),
      Bias()))) ; Softmax() ; Unsqueeze() ; Matmul()
5
6 Linear = add(Matmul(), Bias())
```

Listing 7.6: Definitions of specific attention matching strategies

This compositional approach enables significant code reuse through shared sub-patterns, such as the Linear matcher used multiple times in additive attention. Unlike traditional conditional statements in languages like Python, these matchers can be freely composed and reused across different attention variants. Adding support for new attention variations becomes a matter of composing existing matchers in new ways or defining new specialized matchers when needed rather than adding new branches to a growing set of ad-hoc conditional statements.

Our attention-matching strategy successfully identifies all attention variants that were previously handled by the original ONNX Runtime implementation while requiring significantly less code. The compositional nature of our patterns made it easier to verify their correctness and debug matching failures.

This case study demonstrates how our strategic rewriting approach addresses a significant challenge in modern compiler optimization: matching complex, varied computational patterns like attention mechanisms. By enabling the construction of composable, reusable matching strategies, our approach not only solves the immediate problem of attention pattern matching but also provides a framework for handling similar pattern matching challenges in other domains. The significant reduction in code complexity compared to ONNX Runtime's implementation, while maintaining full functionality, validates the effectiveness of strategic rewriting for real-world compiler optimizations.

While our current implementation successfully demonstrates the benefits of composable pattern matching, there are opportunities to further optimize the matching process itself. We could apply program transformation techniques to the matchers

themselves: inlining shared sub-patterns, eliminating redundant matching attempts, or reordering matching strategies based on their likelihood of success. These optimizations could significantly improve the performance of our pattern matching while maintaining its compositional benefits and readability.

7.4.3 *Compile time impact of immutable rewrites*

Strategic rewriting is based on the principle that transformations can fail, requiring the ability to backtrack to previous program versions and try alternative approaches. This backtracking capability requires preserving the results of previous rewrites, which raises significant performance concerns. The naive approach of cloning the entire IR before each transformation would ensure the ability to roll back, but the computational and memory costs of such complete copies quickly become impractical.

Our solution to this challenge is an immutable IR design based on copy-on-write principles. In a compiler's IR, multiple parts of a program may reference the same sub-computations - for example, the result of one operation might be used as input by several other operations, creating a DAG structure rather than a tree. Our design preserves this sharing structure and, crucially, extends it across different versions of the program created by successive rewrites. Rather than copying entire program representations, we only create new nodes when modifications occur, allowing unmodified portions to be shared with previous versions. This approach maintains the ability to revert to any previous program state while significantly reducing the overhead compared to complete cloning.

The performance characteristics of this immutable IR design depend heavily on program structure and transformation patterns. Real-world programs exhibit varying degrees of sharing in their IR, and many transformations modify only small portions of the program. Consider loop tiling, where a loop nest is restructured with different tile sizes to improve cache locality: while the loop bounds and nesting structure change, the actual computations in the loop body remain unchanged and can be shared across different tiling variants. This example illustrates a key property of our immutable IR design: the copying overhead naturally scales with the locality of the transformation rather than the size of the program.

We evaluate three strategies for managing IR state during program transformation: direct IR mutation that provides optimal

performance but prevents backtracking, complete IR cloning that enables backtracking but incurs maximum copying costs, and our immutable IR implementation that achieves backtracking through selective copying. Through this comparison, we demonstrate that our approach provides the safety guarantees needed for strategic rewriting while keeping copying overhead proportional to the scope of each transformation

To evaluate our immutable IR design, we construct synthetic benchmarks that systematically vary both program size and IR structure. Our benchmarks consist of nested `arith.if` operations containing chains of logical `arith.andi` operations on boolean constants. By controlling the nesting of conditionals and the length of the operation chains, we can generate programs with different degrees of IR connectivity.

While synthetic, these benchmarks capture important structural characteristics of real compiler IRs. The if-statement nesting creates a hierarchical program structure typical of control flow in real programs, while the chains of AND operations within each block represent computational dependencies. This allows us to evaluate how our IR management strategy performs across a spectrum of program structures, from highly localized modifications where changes are contained within a single conditional block to transformations that may propagate across multiple levels of the program hierarchy. We apply two common compiler transformations to these programs: constant folding of the `arith.andi` operations and conditional elimination that inlines the body of an `arith.if` when its condition is known to be true.

These transformations provide a controlled environment to evaluate how different IR management strategies handle both localized changes (constant folding) and structural modifications that cross hierarchical boundaries (conditional elimination). By varying the program structure and measuring both runtime and memory overhead, we can assess how each approach scales with increasing program complexity and operation dependencies.

Figure 7.4 compares three approaches to IR management: destructive rewriting (gray), which offers no backtracking capabilities; complete cloning (blue), which maintains constant but prohibitively high overhead regardless of program structure; and our immutable IR design (green). The vertical lines indicate typical mean operation use dependencies found in climate stencils and BERT small models. The results reveal a crucial insight: while complete cloning enables backtracking, it does so at a consistently high rewriting time that remains prohibitive

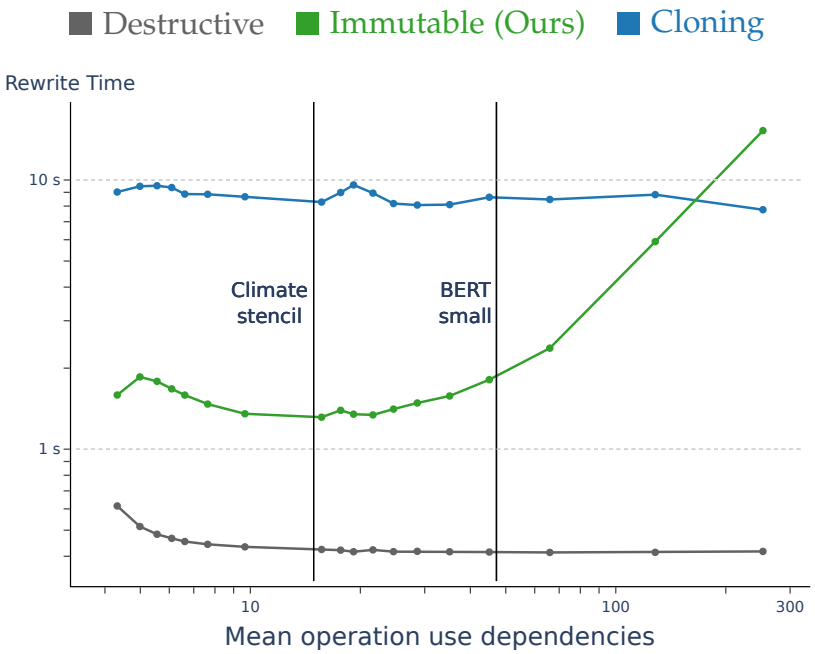


Figure 7.4: Rewrite time comparison of IR management strategies during program transformation. Our immutable IR approach (green) maintains reasonable overhead compared to complete cloning of the IR (blue) while enabling backtracking capabilities that are impossible with destructive rewriting (gray). Vertical lines indicate typical operation dependency values in climate stencils and BERT small models.

regardless of the program's characteristics. Our immutable IR design shows increasing overhead as operation dependencies grow, directly reflecting how our sharing mechanism behaves. When an operation is modified, all operations that depend on its result must also be recreated to maintain IR consistency, even if their computation remains unchanged. Therefore, as the average number of dependencies per operation increases, more IR must be recreated for each transformation, leading to higher overhead. Importantly, for realistic programs like climate stencils and BERT models, which exhibit moderate levels of operation dependencies, this rewrite time overhead remains well below that of complete cloning while still enabling backtracking capabilities.

The memory usage characteristics, shown in Figure 7.5, reveal complementary insights. While destructive rewriting maintains minimal memory overhead, complete cloning shows consistently high memory usage. Our immutable IR design demonstrates increasing memory overhead as operation dependencies grow,

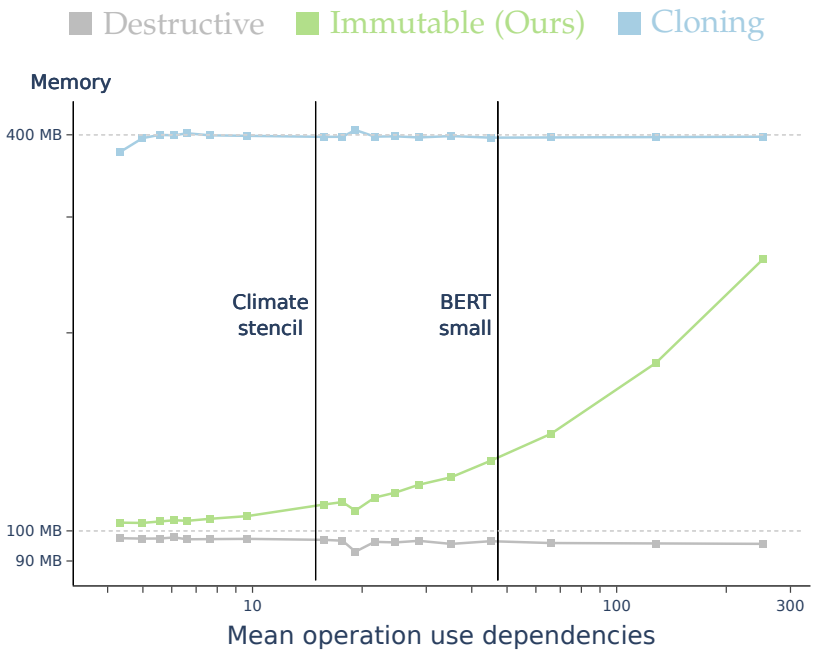


Figure 7.5: Memory footprint of different IR management approaches. While destructive rewriting (gray) provides optimal memory usage but no backtracking and complete cloning (blue) shows prohibitive memory costs, our immutable IR design (green) achieves efficient memory usage through the sharing of unmodified program fragments, especially within realistic dependency ranges (vertical lines) typical of real-world applications.

directly reflecting our sharing mechanism's behavior. However, for the dependency levels typical in real-world applications like climate stencils and BERT models, the memory overhead remains moderate through our sharing strategy.

This case study demonstrates how our immutable IR design successfully addresses a fundamental challenge in compiler design: enabling backtracking without incurring the fixed, high costs of complete program cloning. By sharing unmodified program fragments across versions, we achieve both the ability to backtrack transformations and practical performance. Traditional destructive updates, while efficient, provide no mechanism for backtracking, limiting their utility in explorative optimization scenarios. Complete cloning enables backtracking but maintains a consistently high overhead regardless of program structure, making it impractical for production use. Our approach bridges this gap, making backtracking feasible in real compiler infrastructures while maintaining reasonable resource

usage during transformation sequences. However, since our approach still incurs non-negligible overhead compared to destructive updates, it should be selectively applied to compiler phases where backtracking capabilities enable valuable optimization opportunities rather than used universally throughout the compilation pipeline.

7.4.4 *Future Work*

A promising direction for future research lies in leveraging our strategic rewriting system's backtracking capabilities to implement more sophisticated program optimization searches. Current MLIR rewriting relies on a greedy driver that applies rewrites based on static benefit scores assigned to individual transformations. This approach fundamentally limits the exploration of transformation sequences: users must carefully avoid introducing rewrites that could produce equivalent expressions, such as a transformation A and its inverse A^{-1}, as this would risk infinite loops.

In contrast, our immutable IR design enables the safe exploration of such bidirectional transformations. This capability is crucial for hill-climbing optimization strategies where a transformation might temporarily 'worsen' the program to enable other beneficial rewrites. For example, consider the expression $transpose(A \times^{-1} B)$ with $\times^{-1}$ representing fused matrix multiplication and transposition. While fusing the transposition into the matrix multiplication is generally beneficial for performance, this fusion must be undone to enable the algebraic simplification where the inner and outer transpose operations cancel each other. While systems like Lift and Rise leverage such exploration patterns, traditional compiler frameworks typically must restrict themselves to a carefully curated subset of transformations that can be orchestrated without backtracking capabilities, potentially missing optimization opportunities.

Several research directions emerge from this observation:

- Evaluation of search strategies that systematically explore compiler transformation sequences, including temporary detours through suboptimal program states
- Integration of learning-based approaches to guide the exploration of transformation combinations

- Design of better heuristics for evaluating transformation sequences rather than individual rewrites
- Extension of our system to support parallel exploration of multiple transformation paths

7.5 CONCLUSION

This chapter has demonstrated that bridging the gap between compositional rewriting systems and traditional compiler infrastructures is both feasible and valuable. Through our integration of Elevate with MLIR, we have shown how seemingly incompatible approaches to program transformation—immutable, compositional rewriting and destructive IR modifications—can be reconciled to create a powerful optimization framework. These modifications yield the following key benefits:

- Composition of complex transformation sequences with rollback capabilities
- Direct operation on MLIR's intermediate representation while maintaining transformation composability
- Strategic transformations across MLIR's extensive ecosystem of domain-specific dialects

This work represents a significant evolution beyond our previous contribution of the Transform dialect, offering a more principled approach to scheduling transformations. While the Transform dialect excels in its deep integration with MLIR's existing infrastructure, our strategic rewriting system provides stronger formal foundations and enables explorative optimization through safe backtracking. The trade-off between integration and formalism manifests in practical terms: our approach introduces computational overhead that may be unnecessary for simple, deterministic transformations but proves valuable for complex optimization scenarios requiring the exploration of multiple transformation paths.

Building upon the verification guarantees established earlier in this thesis, this integration of principled rewriting systems with practical compiler infrastructure demonstrates how modern compilers can combine formal foundations with powerful transformation capabilities. This represents a significant step toward compiler frameworks that are both theoretically well-founded and practically applicable.

8

DISCUSSION

This thesis addressed two fundamental challenges in modern compiler design: integrating principled program representations and transformations into production compilers and providing users with meaningful control over optimization decisions. These challenges arise from two key factors: the growing complexity of application domains like machine learning and the increasing diversity of hardware architectures. Together, these factors push the boundaries of traditional compilation techniques, requiring new approaches to program representation and optimization. Our work tackled these challenges through four key contributions:

1. Integrating Rise, a pattern-based representation, into MLIR
2. Developing a static verification framework for MLIR transformations
3. Creating a practical rewriting system with the Transform dialect
4. Designing a principled rewriting system in xDSL based on Elevate

Each contribution not only advanced the state of the art in compiler technology but also revealed crucial insights about the practical realities of evolving compiler infrastructure. These insights, distilled into lessons learned, offer guidance for future compiler research and development.

8.1 LESSONS LEARNED

The lessons we have learned span a range of topics and highlight the tensions between theoretical ideals and practical constraints, the value of incremental progress, and the importance of bridging academic innovations with industrial practice. We now examine these lessons learned for each of our four contributions.

8.1.1 *Compositional Program Representation*

In Chapter 4, we demonstrated the integration of Rise, a functional pattern-based intermediate representation, into the MLIR framework. We showed how to represent high-level computational patterns within an SSA-based infrastructure, compile them efficiently to imperative code, and leverage both domain-specific and general-purpose optimizations. This work bridged the gap between academic innovations in program representation and practical compiler infrastructure. From this experience, we draw several lessons:

INFER TRANSLATIONS FROM MATHEMATICAL FOUNDATIONS
A significant challenge in our work was the manual definition of lowering rules between high-level graph representations (like TensorFlow) and our pattern-based representation. This process was not only time-consuming but also error-prone, requiring careful consideration to maintain semantic correctness across the different levels of abstraction. However, our experience with Rise revealed a promising opportunity: its mathematical foundations could potentially automate these translations. By formalizing the mathematical specifications of high-level operations and the semantics of our pattern-based IR, we could derive these translations automatically rather than defining them by hand. This approach extends beyond Rise, suggesting a broader principle for compiler design: build intermediate representations on mathematical foundations to enable automated, correct translations between semantic levels. Such an approach could dramatically improve compiler extensibility and adaptability, allowing quick integration of new frameworks while maintaining rigorous correctness guarantees.

ENHANCE, DON'T REPLACE: THE POWER OF INTEGRATION
Our work with Rise demonstrates that significant innovations in compiler technology can be achieved through careful integration with existing ecosystems. Rather than building a completely new compiler infrastructure, we showed how a different approach to program representation could be incorporated into an existing framework. This integration approach allowed us to leverage the strengths of established compiler technology while gradually introducing novel capabilities. It suggests that compiler evolution doesn't always require revolutionary changes but can proceed through the thoughtful integration of new ideas with

proven practices. This principle encourages compiler designers to look for opportunities to enhance existing systems with new paradigms rather than always starting from scratch.

EMBRACING INCREMENTAL PROGRESS: THE POWER OF PARTIAL SOLUTIONS Our implementation of Rise in MLIR revealed an important lesson about bridging theoretical models with practical compiler infrastructures. While we successfully integrated Rise's core concepts, we encountered limitations in representing its full type system, particularly dependent typing. This feature, key to Rise's powerful static reasoning capabilities about array lengths and other properties, couldn't be fully preserved in our MLIR implementation due to the limited expressiveness of MLIR's built-in type system.

However, this limitation did not negate the value of the integration. On the contrary, it highlighted an important principle in compiler design: partial implementation of advanced features can still yield significant benefits. Despite the absence of dependent typing, our Rise dialect in MLIR successfully demonstrated the power of pattern-based representations and enabled new optimization opportunities.

This experience underscores that the integration of academic models into production compilers often happens gradually and requires pragmatic trade-offs. However, these trade-offs don't necessarily undermine the model's core benefits. It also reveals an exciting open problem in compiler design: How can we enhance type systems in frameworks like MLIR to support more expressive features like dependent typing?

8.1.2 *Static Transformation Verification*

Our work on static verification introduced compile-time checks for MLIR's Pattern Definition Language (PDL), complementing existing runtime verification. Through SMT-based constraint solving and a dedicated analysis, we enabled the verification of structural properties and basic constraints before pattern application.

FUZZING TO FILL THE ADOPTION GAP The scarcity of PDL patterns in production systems highlights a fundamental challenge in compiler research: how to rigorously evaluate new techniques before they see widespread adoption? Our response - developing a systematic fuzzing infrastructure - demonstrates

that methodological testing approaches can bridge this gap. While real-world examples remain the gold standard, carefully designed synthetic benchmarks can provide meaningful coverage and uncover real issues. The discovery of actual bugs in MLIR through this approach demonstrates that rigorous methodology can validate compiler techniques long before they see widespread adoption.

SEMANTIC VERIFICATION SHOULDN'T BE AN AFTERTHOUGHT Our work exposes a fundamental challenge in compiler verification: while structural properties like SSA form can be automatically verified, full semantic verification remains a much harder problem. This is particularly acute in extensible compiler frameworks like MLIR, where operation semantics are either undefined or, at best, inferrable through their lowering to lower-level operations. Since these lowerings are implemented in arbitrary C++ code, proving semantic preservation becomes intractable. This reveals a fundamental tension in compiler design: extensibility and current implementation practice stand in opposition to semantic verifiability. The lesson here is clear: semantic verification should be considered from the ground up in compiler design, not treated as an add-on feature. Currently, we have no comprehensive solution to this dilemma, highlighting a critical area for future research in compiler design.

8.1.3 *A practical rewriting system for MLIR*

The Transform dialect demonstrates how to expose compiler transformations as first-class operations. By making transformations explicit in the IR and deeply integrating them with MLIR's infrastructure, we enable fine-grained control over optimizations and make previously hidden compiler capabilities reusable. Our case studies validate this approach across different scenarios, from ensuring robust dialect lowering to enabling systematic performance optimization. From this work, we infer several lessons for compiler design:

TRANSFORMATION STEPS MUST BE INTROSPECTABLE Our experience with the Transform dialect showed that representing individual transformation steps as distinct operations - rather than hiding them in black-box passes - enables step-by-step reasoning about program evolution. Traditional compiler passes often combine multiple transformations into a single unit, ob-

scuring the intermediate results and making it difficult to understand their effects. By making these states explicit, we can systematically debug transformations, identify unintended interactions between optimizations, and compose transformations in novel ways. This visibility enables reuse across different optimization scenarios while making the compilation process more transparent, understandable, and maintainable.

DECOUPLE MECHANISM AND POLICY Traditional compilers often embed optimization heuristics - like when to inline or what tile sizes to choose - directly into their transformation implementations. This tight coupling makes it difficult to adapt the compiler as requirements change or new hardware emerges. The Transform dialect continues a line of thinking established by systems like Halide and Elevate (as discussed in Chapters 2 and 3), which recognizes the importance of separating transformation mechanisms from optimization policies. Building on these foundations, we demonstrate how this separation principle can be realized within a production compiler framework: separating the implementation of transformations (mechanisms) from the decisions about when and how to apply them (policies). As shown in our tiling case study, this separation allows the same transformation capabilities to be controlled by different policies - from manual expert guidance to automated search strategies - without modifying the compiler. Heuristics can be moved from the compiler implementation into explicit, configurable components, making them easier to understand, modify, and tune for specific scenarios.

MIND THE GAP BETWEEN INTENT AND IMPLEMENTATION The Transform dialect taught us an important lesson about compiler interfaces: while making transformations explicit and composable enables compiler experts to control optimizations effectively, it remains challenging to operate for domain experts. Even with these capabilities exposed, users still need extensive knowledge of MLIR's dialects and transformations to express optimization strategies. Our experience shows that successful compiler interfaces must bridge the gap between domain expertise and compiler implementation. While our contribution provides a foundation through explicit and composable transformations, the challenge of building abstractions that capture user intent without requiring deep compiler knowledge remains crucial for the future of compiler design.

POWER THROUGH COMPOSITION Our work with the Transform dialect revealed the benefits of treating transformations as composable units rather than monolithic passes. By breaking down complex transformations into smaller, well-defined pieces, we enable more flexible optimization strategies. While our current implementation requires manual specification of transformation effects, it demonstrates how composition allows us to build complex transformation sequences from simpler building blocks. Users can mix and match these components to create customized optimization strategies, and new transformations can be added by combining existing pieces in novel ways. The specification of transformation properties, like their expected inputs and effects, helps users understand how pieces can be combined safely and effectively. This insight directly informed our work in Chapter 7, which further develops these compositional principles. The ability to compose smaller, well-understood transformations, each with clear properties, into larger optimization strategies has proven essential for handling the complexity of modern compilation challenges.

8.1.4 *A principled Rewriting System for MLIR*

In Chapter 7, we developed an integration between Elevate's principled rewriting system and MLIR's infrastructure. By implementing an immutable IR design and adapting Elevate's rewriting mechanisms to work within MLIR's framework, we enabled features like transformation rollback and strategic exploration. While our evaluation demonstrated the feasibility of this approach, it also revealed important limitations and trade-offs that inform future compiler design.

DEMONSTRATE PRINCIPLES IN SIMPLIFIED SETTINGS Our initial attempts to integrate Elevate directly with MLIR's C++ infrastructure failed due to fundamental tensions with MLIR's mutable design. Instead, we succeeded by demonstrating these ideas in xDSL, a simpler framework based on MLIR's principles. This taught us an important lesson about validating new approaches: while the end goal might be integration with production systems, demonstrating feasibility in a controlled setting can be more valuable than struggling with incidental complexity. Our successful integration with xDSL proved that principled rewriting and industrial compiler design can coexist, paving the way for future integration with full MLIR.

COMPOSITION HANDLES COMPLEXITY ELEGANTLY Our work on attention pattern matching demonstrated the power of compositional design in compiler transformations. By decomposing complex pattern matching logic into reusable components, we reduced hundreds of lines of ad-hoc code to clear, composable pieces. Using Elevate's approach, we showed how composition can elegantly solve seemingly complex requirements - from IR traversal strategies to failure management - without requiring special mechanisms. This reinforces composition as the fundamental principle for building robust and flexible transformation systems.

MATCH TOOLS TO TASKS Our implementation of an immutable IR with copy-on-write semantics demonstrated that principled approaches can be made practical but not without significant trade-offs. While we achieved acceptable performance for complex optimization scenarios requiring backtracking, the 20% overhead compared to destructive rewriting makes this approach impractical for simple transformations. Just as we wouldn't use a sledgehammer to hang a picture, compiler frameworks should provide multiple transformation mechanisms - using principled approaches where their benefits of safety guarantees and exploration capabilities justify the compile time penalty while keeping a smaller hammer ready for straightforward transformations.

MAKE EXPLORATION STRATEGIC Our support for backtracking in transformation sequences revealed important insights about optimization strategies in modern compilers. While previous work demonstrated that some optimizations require temporary regression to reach optimal solutions - challenging the traditional greedy approaches - blindly exploring all possibilities isn't practical. The ability to explore multiple transformation paths is powerful but must be guided by clear strategies to avoid compilation time explosions. Just as a chess player doesn't consider every possible move but focuses on promising strategies, effective compiler exploration must combine the freedom to backtrack with intelligent guidance about which paths are worth exploring.

8.2 LIMITATIONS

While our contributions advance the state of the art in several areas of compiler design, it's important to acknowledge their limitations. Some of these limitations not only highlight areas for future work but also reveal fundamental challenges in bridging theoretical approaches with practical compiler implementation. We organize our discussion of limitations according to our four main contributions, examining where our approaches fall short and what these shortcomings tell us about compiler design.

8.2.1 *Compositional Program Representation*

TYPE SYSTEM EXPRESSIVENESS GAPS Our integration of Rise into MLIR revealed significant limitations in representing advanced type system features. While MLIR's type system is extensible, it lacks support for implementing dependent types, which are crucial for expressing Rise's sophisticated reasoning about array shapes. For other features, we were forced to encode type information through alternative means, such as attributes, making the representation more cumbersome and reducing the static guarantees available in the original Rise system. This gap in expressiveness highlights a fundamental challenge in bridging high-level, mathematically oriented program representations with production compiler frameworks.

UNREALIZED REWRITING POTENTIAL While our integration successfully demonstrated the feasibility of representing Rise patterns in MLIR, we were unable to port the extensive library of rewrite rules from Rise's Scala implementation. These rules embody the key power of pattern-based representation by enabling systematic program optimization, but they could not be effectively expressed within MLIR's existing rewriting infrastructure. This limitation directly motivated our subsequent work on both practical and principled rewriting systems, highlighting the need for more expressive transformation frameworks.

8.2.2 *Sane Rewriting*

LIMITED TO STRUCTURAL VALIDATION While our framework successfully validates structural properties and some basic constraints, it can not validate the semantic correctness of transformations. The lack of formal semantics for MLIR operations,

where semantics are primarily defined through their lowering behavior in C++ implementations, makes semantic verification particularly challenging. Adding semantic verification capabilities to an extensible framework like MLIR as an afterthought presents significant challenges, as it would require additional infrastructure for specifying and reasoning about operation semantics. While this is an active area of current research[1], a comprehensive solution would require formal specifications or verified lowering rules integrated into the core framework design.

8.2.3 *A practical rewriting system for MLIR*

The Transform dialect represents a significant step forward in providing explicit control over production compiler transformations. However, our experience implementing and using this system revealed several important limitations that must be addressed for broader adoption and impact.

BEYOND SURFACE-LEVEL ABSTRACTIONS Despite our implementation of a Python frontend that provides higher-level abstractions, the framework still requires users to understand deep compiler concepts. Domain experts must grasp the intricacies of multiple intermediate representations and their interactions, as the underlying compiler concepts remain deeply entrenched in the interface. This presents a significant barrier to adoption by domain experts who understand their optimization requirements but lack compiler expertise.

RELIANCE ON MANUAL SCHEDULE CONSTRUCTION While we successfully demonstrate integration with autotuners for exploring parameter combinations and enabling/disabling specific transformation steps, the Transform framework lacks support for auto-scheduling - the automatic synthesis and exploration of entirely new transformation sequences. This limitation is particularly relevant for complex optimization scenarios where the space of possible transformation combinations is too large to be explored manually and where automated discovery of novel transformation strategies could yield significant benefits.

1 Collaborators are currently preparing submissions proposing better approaches for the formal specification of MLIR operation semantics.

LEGACY ERROR HANDLING IMPEDANCE The Transform dialect exposes a fundamental mismatch between its error-handling system and the error-reporting capabilities of the internal transformations it exposes. While the dialect implements a systematic approach to error handling, it necessarily interacts with numerous pre-existing internal compiler transformations. Although we enhanced many of these transformations with proper error reporting capabilities, the sheer number of existing transformations means that many still lack adequate error feedback, having been originally designed for use within opaque compiler passes rather than explicit transformation sequences. This impedance between the dialect's error-handling expectations and legacy transformation behavior creates significant challenges for users trying to debug transformation sequences, as failures in underlying transformations may provide insufficient context for understanding and resolving issues. This challenge is particularly acute when complex transformation sequences fail, as the actual source of the error might be obscured by inadequate error reporting in the underlying compiler infrastructure.

These limitations highlight important areas for future research. Particularly promising directions include developing higher-level abstractions that better insulate users from compiler complexity, advancing auto-scheduling capabilities for automated strategy discovery, and improving error-handling infrastructure across all levels of the compilation stack. Addressing these limitations would significantly enhance the practical utility of the Transform dialect while maintaining its fundamental benefits of explicit and composable transformation control.

8.2.4 *A principled Rewriting System for MLIR*

While our integration of principled rewriting through Elevate demonstrates the feasibility of bringing formal rewriting concepts to industrial compiler frameworks, our implementation reveals several important limitations that must be considered.

SHARED CHALLENGES WITH PRACTICAL REWRITING Despite its different theoretical foundation, our principled rewriting system faces similar adoption barriers as the Transform dialect. Users still need to manually construct transformation sequences, as the system does not provide automatic schedule synthesis. Furthermore, effective use of the system requires a deep understanding of both the IR semantics and the rewrit-

ing strategies available through Elevate. While Elevate's formal foundations provide stronger correctness guarantees, they do not reduce the expertise required to effectively guide program optimization.

PERFORMANCE COST OF IMMUTABILITY The immutable IR design, while enabling crucial features like transformation rollback and strategic exploration, incurs significant performance overhead. Our copy-on-write implementation achieves acceptable performance for complex optimization scenarios requiring backtracking but still shows a 20x slowdown compared to traditional mutable approaches. This trade-off between principled design and performance remains a fundamental challenge for adoption in performance-critical compilation workflows.

LIMITED MLIR INTEGRATION A key limitation of our current implementation is its separation from MLIR's core infrastructure. The pervasive nature of mutability throughout MLIR's IR design necessitated our implementation in xDSL with a redesigned immutable IR structure. While this approach successfully demonstrates the feasibility of Elevate-based rewriting in multi-IR environments, the lack of direct MLIR integration limits the immediate practical impact of our work. Integration with MLIR's existing infrastructure would enable better interoperability with the growing ecosystem of MLIR dialects and transformations. A promising path forward exists through mechanisms similar to the `transform.alternatives` operation we introduced as part of the Transform dialect. It implements localized IR copying for limited backtracking scenarios. However, adapting this approach for our use case still requires significant refinement, as the current implementation only supports operations with regions that are isolated from above, whereas our rewriting system requires much finer granularity of control. While the scalability of such an approach for extensive rewriting sequences remains an open question, it represents a viable path toward implementing Elevate's capabilities directly within MLIR's existing infrastructure rather than requiring a completely separate implementation.

8.3 TOWARD PRINCIPLED YET PRACTICAL COMPILER DESIGN

The work presented in this thesis demonstrates steps toward a compiler framework that combines systematic program repre-

sentation, verified transformation, and meaningful user control. In such a framework, computations are represented as compositions of well-understood patterns, making their mathematical structure explicit and amenable to principled optimization. Transformations are statically verified to be correct, enabling safe composition of optimization strategies. Rewriting systems expose the necessary control to users, allowing both manual expertise and automated tools to guide and explore optimization decisions.

MLIR's extensible architecture and growing ecosystem make progress toward this vision practical. As our implementations and case studies demonstrate, principled techniques can enhance existing compilation pipelines rather than replace them. This integration capability transforms isolated academic advances into practical tools that can immediately benefit real-world compilation tasks.

However, significant challenges remain. The path toward fully realizing this vision requires advances in several areas. The verification of transformation sequences needs to scale beyond basic properties to capture complex optimization interactions. Transformation systems must find ways to expose meaningful control without requiring users to become experts in multiple levels of intermediate representations. While our contributions demonstrate the feasibility of this vision, they also highlight how much work remains in creating compiler infrastructure that truly enables systematic optimization through principled yet practical means.

8.4 CLOSING THOUGHTS

Modern compiler design faces increasing pressure to evolve beyond traditional optimization approaches. While machine learning has brought many of these challenges to the forefront, there is a concerning trend toward developing ML-specific compilation solutions that don't generalize beyond their narrow domain. Our work has deliberately taken a different approach. Although we demonstrated our techniques in the context of ML compilation, the underlying principles – flexible program representation, principled and practical transformation control, and static verification of rewrites – address fundamental aspects of compiler design that are valuable across domains.

Our work with MLIR has demonstrated both opportunities and challenges in enhancing production compilers with aca-

demic approaches. The Transform dialect's upstream adoption shows that providing explicit control over transformations fills a real need in practice. At the same time, our experiences with pattern-based representations and static verification revealed significant engineering challenges: maintaining performance while preserving principled foundations requires careful design choices and often complex implementation trade-offs.

These experiences underscore the importance of bridging the gap between theoretical advances and practical implementation. As we have shown, it's possible to introduce principled techniques into production compilers, but doing so requires a nuanced understanding of both academic ideals and industrial constraints. Success in future compiler design will depend on finding this balance and developing approaches that are both theoretically sound and practically viable.

As computing continues to specialize and diversify, the path forward lies not in accumulating domain-specific workarounds but in developing principled solutions that generalize across domains, thus not only addressing current challenges but also preparing us for future paradigms.

BIBLIOGRAPHY

[1] *AMD matrix cores (amd-lab-notes) - AMD GPUOpen*. URL: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-readme/#example-1-v-mfma-f32-16x16x4f32.

[2] Andrew W. Appel. "SSA is Functional Programming." In: *ACM SIGPLAN Notices* 33.4 (1998), pp. 17–20.

[3] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. "Strategy Preserving Compilation for Parallel Functional Code." In: *CoRR* abs/1710.08332 (2017).

[4] Dzmitry Bahdanau. "Neural machine translation by jointly learning to align and translate." In: (2014). arXiv: 1409.0473.

[5] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. "SMT-Based Translation Validation for Machine Learning Compiler." In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 386–407. ISBN: 978-3-031-13188-2.

[6] Paul Barham and Michael Isard. "Machine Learning Systems are Stuck in a Rut." In: *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 2019, pp. 177–183.

[7] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. "Efficient and portable einstein summation in SQL." In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–19.

[8] Uday Bondhugula. "High Performance Code Generation in MLIR: An Early Case Study with GEMM." In: *CoRR* abs/2003.00532 (2020).

[9] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael F. P. O'Boyle. "mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR Using Program Synthesis." In: *2023 32nd Interna-*

tional Conference on Parallel Architectures and Compilation Techniques (PACT). 2023, pp. 39–50.

[10] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. "VST-Floyd: A separation logic tool to verify correctness of C programs." In: *Journal of Automated Reasoning* 61 (2018), pp. 367–422.

[11] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "TVM: An automated End-to-End optimizing compiler for deep learning." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[13] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. "Attention-based models for speech recognition." In: *Advances in neural information processing systems* 28 (2015).

[14] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. "Principles of maude." In: *Electronic Notes in Theoretical Computer Science* 4 (1996), pp. 65–89.

[15] M Cole. "Algorithmic skeletons: a structured approach tothe management of parallel computation." PhD thesis. Ph. D. dissertation, University of Edinburgh, UK, 1988.

[16] Bruce Collie and Michael F.P. O'Boyle. "Program Lifting using Gray-Box Behavior." In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 60–74.

[17] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. "Flashattention: Fast and memory-efficient exact attention with io-awareness." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805.

[19] Chris Fallin. *Ægraphs: Acyclic E-graphs for Efficient Optimization in a Production Compiler*. EGRAPHS 2023 - E-Graph Research, Applications, Practices, and Human-factors Symposium. Accessed: 2024. 2023. URL: https://cfallin.org/pubs/egraphs2023_aegraphs_slides.pdf.

[20] Paul Feautrier and Christian Lengauer. "Polyhedron Model." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1581–1592. ISBN: 978-0-387-09766-4.

[21] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. "IRDL: an IR definition language for SSA compilers." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 199–212. ISBN: 9781450392655.

[22] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Lücke, Théo Degioanni, Michel Steuwer, and Tobias Grosser. "Sidekick compilation with xDSL." In: *Accepted at the 2025 International Symposium on Code Generation and Optimization, CGO 2025*.

[23] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. "SPIRAL: Extreme Performance Portability." In: *Proceedings of the IEEE* 106.11 (2018), pp. 1935–1968.

[24] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing." In: *Systems for Machine Learning* 4.9 (2018).

[25] Rongxiao Fu. "Type systems for safe strategic rewriting." In: (2024).

[26] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies." In: *International Journal of Parallel Programming* 34 (2006), pp. 261–317.

[27] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A data-movement-aware scheduling language for gpus." In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 71–82.

[28] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 92:1–92:29.

[29] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High performance stencil code generation with lift." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. Ed. by Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle. ACM, 2018, pp. 100–112.

[30] Guoliang He and Eiko Yoneki. "CuAsmRL: Optimizing GPU SASS Schedules via Deep Reinforcement Learning." In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 2025, pp. 493–506.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[32] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. "LIBXSMM: accelerating small matrix multiplications by runtime code generation." In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 981–991.

[33] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. "BaCO: A Fast and Portable Bayesian Compiler Optimization Framework." In: *Proceedings of the 28th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems, Volume 4. ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 19–42. ISBN: 9798400703942.

[34] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[35] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. "Exocompilation for productive programming of hardware accelerators." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 703–718.

[36] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Bernstein, and Jonathan Ragan-Kelley. "Exo 2: Growing a Scheduling Language." In: *arXiv preprint arXiv:2411.07211* (2024).

[37] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. "INSPIRE: The insieme parallel intermediate representation." In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*. Ed. by Christian Fensch, Michael F. P. O'Boyle, André Seznec, and François Bodin. IEEE Computer Society, 2013, pp. 7–17.

[38] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. "The tensor algebra compiler." In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 77:1–77:29.

[39] Julien Klaus, Mark Blacher, and Joachim Giesen. "Compiling tensor expressions into einsum." In: *International Conference on Computational Science*. Springer. 2023, pp. 129–136.

[40] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. "Guided Equality Saturation." In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024), pp. 1727–1758.

[41] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML." In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 179–191.

[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. "Efficient memory management for large language model serving with pagedattention." In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 611–626.

[43] Chris Lattner. "LLVM and Clang: Next generation compiler technology." In: *The BSD conference*. Vol. 5. 2008, pp. 1–20.

[44] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *CGO*. IEEE Computer Society, 2004, pp. 75–88.

[45] 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: *CGO*. 2021.

[46] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. "AnyDSL: a partial evaluation framework for programming high-performance libraries." In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018).

[47] Roland Leißa, Marcel Köster, and Sebastian Hack. "A graph-based higher-order intermediate representation." In: *CGO*. IEEE Computer Society, 2015, pp. 202–212.

[48] Roland Leißa, Marcel Ulrich, Joachim Meyer, and Sebastian Hack. "MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age." In: *arXiv preprint arXiv:2411.07443* (2024).

[49] Xavier Leroy. "Formal verification of a realistic compiler." In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782.

[50] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert-a formally verified optimizing compiler." In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[51] Maksim Levental, Alok Kamatar, Ryan Chard, Kyle Chard, and Ian Foster. *nelli: a lightweight frontend for MLIR*. 2023. arXiv: 2307.16080 [cs.PL].

[52] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. "Provably correct peephole optimizations with alive." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pp. 22–32.

[53] Nuno P Lopes and John Regehr. "Future Directions for Optimizing Compilers." In: *arXiv preprint arXiv:1809.02161* (2018).

[54] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. "Alive2: bounded translation validation for LLVM." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 65–79. ISBN: 9781450383912.

[55] Martin Lücke. "Fine Grained Control of Program Transformations via Strategic Rewriting in MLIR." International Symposium on Code Generation and Optimization (CGO) Student Research Competition. 2021.

[56] Minh-Thang Luong. "Effective approaches to attention-based neural machine translation." In: *arXiv preprint arXiv:1508.04025* (2015).

[57] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu,

and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." In: *arXiv preprint arXiv:1603.04467* (2015).

[58] William M. McKeeman. "Peephole optimization." In: *Commun. ACM* 8.7 (1965), pp. 443–444.

[59] Jules Merckx. "Building Bridges: Julia as an MLIR Frontend." Master's thesis. Ghent University, 2024.

[60] *MLIR Affine dialect online documentation*. 2020. URL: https://mlir.llvm.org/docs/Dialects/Affine/.

[61] *MLIR Language Reference*. 2024. URL: https://mlir.llvm.org/docs/LangRef/#blocks.

[62] Naums Mogers. "Guided rewriting and constraint satisfaction for parallel GPU code generation." PhD thesis. University of Edinburgh, 2023.

[63] William Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485.

[64] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. "Polygeist: Raising C to Polyhedral MLIR." In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 45–59.

[65] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. "SoftBound: Highly compatible and complete spatial memory safety for C." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258.

[66] George C Necula. "Translation validation for an optimizing compiler." In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 2000, pp. 83–94.

[67] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure." In: *J. ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN: 0004-5411.

[68] Robert Nieuwenhuis and Albert Oliveras. "Proof-producing congruence closure." In: *International Conference on Rewriting Techniques and Applications*. Springer. 2005, pp. 453–468.

[69] *ONNX runtime*. 2024. URL: https://github.com/microsoft/onnxruntime.

[70] *opencompl/xdsl-smt: The implementation of an SMTLib dialect for xDSL*. https://github.com/opencompl/xdsl-smt.

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019.

[72] *Properties in MLIR*. https://mlir.llvm.org/OpenMeetings/2023-02-09-Properties.pdf.

[73] Xueying Qin, Liam O'Connor, Rob van Glabbeek, Peter Höfner, Ohad Kammar, and Michel Steuwer. "Shoggoth: A Formal Foundation for Strategic Rewriting." In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 61–89.

[74] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. "Robust Speech Recognition via Large-Scale Weak Supervision." In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, July 2023, pp. 28492–28518.

[75] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners." In: (2019).

[76] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." In: *ACM Transactions on Graphics (TOG)* 31.4 (2012), pp. 1–12.

[77] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." In: *ACM Trans. Graph.* 31.4 (2012), 32:1–32:12.

[78] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. "Halide: Decoupling algorithms from schedules for high-performance image processing." In: *Communications of the ACM* 61.1 (2017), pp. 106–115.

[79] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[80] Jonathan Millard Ragan-Kelley. "Decoupling algorithms from the organization of computation for high performance image processing." PhD thesis. Massachusetts Institute of Technology, 2014.

[81] Fabrice Rastello et al. *SSA-Based Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1441962018.

[82] River Riddle. *PDL - Pattern Descriptor Language.pdf*. https://drive.google.com/file/d/17WYUvlmCzNTiqLaxWf_uz4GiLm3QVoEV/view. [Accessed 13-12-2024]. 2021.

[83] Norman A. Rink and Jeronimo Castrillon. "TeIL: a type-safe imperative tensor intermediate language." In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 57–68. ISBN: 9781450367172.

[84] Grigore Roșu and Traian Florin Șerbănuță. "An overview of the K semantic framework." In: *The Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434.

[85] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. "Optimizing data reshaping operations in functional IRs for high-level synthesis." In: *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference*

on Languages, Compilers, and Tools for Embedded Systems. 2022, pp. 61–72.

[86] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. "Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations." In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 287–298.

[87] Lukas Siefke, Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. "Systematically extending a high-level code generator with support for tensor cores." In: *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*. GPGPU '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022. ISBN: 9781450393485.

[88] Richard M Stallman. "GNU compiler collection internals." In: *Free Software Foundation* 46 (2002).

[89] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code." In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 205–217. ISSN: 0362-1340.

[90] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. "RISE & shine: Language-oriented compiler design." In: *arXiv preprint arXiv:2201.03611* (2022).

[91] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation." In: *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, 2016, 15:1–15:10.

[92] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation." In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, 2017, pp. 74–85.

[93] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "Tiling optimizations for stencil computations using rewrite rules in lift." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.4 (2019), pp. 1–25.

[94] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices*. 2020. arXiv: 2004.02984 [cs.CL].

[95] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. "Llm-vectorizer: Llm-based verified loop vectorizer." In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 2025, pp. 137–149.

[96] Arun Thangamani, Vincent Loechner, and Stéphane Genaud. "A Survey of General-purpose Polyhedral Compilers." In: *ACM Trans. Archit. Code Optim.* 21.4 (Nov. 2024). ISSN: 1544-3566.

[97] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations." In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196.

[98] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating value-graph translation validation for LLVM." In: *Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation*. 2011, pp. 295–305.

[99] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. "Joint scheduling and layout optimization to enable multi-level vectorization." In: *IMPACT, Paris, France* (2012).

[100] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. "Structured Operations: Modular Design of Code Generators for Tensor Compilers." In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2022, pp. 141–156.

[101] Eelco Visser. "Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5." In: *International Conference on Rewriting Techniques and Applications*. Springer. 2001, pp. 357–361.

[102] Eelco Visser and Zine-el-Abidine Benaissa. "A Core Language for Rewriting." In: *Electronic Notes in Theoretical Computer Science* 15 (1998). International Workshop on Rewriting Logic and its Applications, pp. 422–441. ISSN: 1571-0661.

[103] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: Fast and extensible equality saturation." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021).

[104] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. "SUIF: an infrastructure for research on parallelizing and optimizing compilers." In: *SIGPLAN Not.* 29.12 (Dec. 1994), pp. 31–37. ISSN: 0362-1340.

[105] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. "Equality Saturation for Tensor Graph Superoptimization." In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 255–268.

[106] Huihui Zhang, Anand Venkat, Protonu Basu, and Mary Hall. "Combining polyhedral and ast transformations in chill." In: *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT*. Vol. 16. 2016.

[107] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. "Formalizing the LLVM intermediate representation for verified program transformations." In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2012, pp. 427–440.

[108] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. "Ansor: Generating {High-Performance} tensor programs for deep learning." In: *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 2020, pp. 863–879.

[109] Oleksandr Zinenko. *MLIR Is Not an ML Compiler – And Other Common Misconceptions*. 2023. URL: https://llvm.org/devmtg/2023-10/slides/techtalks/Zinenko-MLIRisNotAnMLCompiler.pdf.