The MLIR Transform Dialect

Your Compiler Is More Powerful Than You Think

Martin Paul Lücke
martin.luecke@ed.ac.uk
University of Edinburgh
Edinburgh, Scotland, UK

Oleksandr Zinenko
contact@ozinenko.com
Google DeepMind
Paris, France

William S. Moses
wsmoses@illinois.edu
University of Illinois
Urbana-Champaign, IL, United States
Google DeepMind
Cambridge, MA, United States

Michel Steuwer
michel.steuwer@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Albert Cohen
albertcohen@google.com
Google DeepMind
Paris, France

Abstract

To take full advantage of a specific hardware target, performance engineers need to gain control on compilers in order to leverage their domain knowledge about the program and hardware. Yet, modern compilers are poorly controlled, usually by configuring a sequence of coarse-grained monolithic black-box passes, or by means of predefined compiler annotations/pragmas. These can be effective, but often do not let users precisely optimize their varying compute loads. As a consequence, performance engineers have to resort to implementing custom passes for a specific optimization heuristic, requiring compiler engineering expert knowledge.

In this paper, we present a technique that provides fine-grained control of general-purpose compilers by introducing the *Transform dialect*, a controllable IR-based transformation system implemented in MLIR. The Transform dialect empowers performance engineers to optimize their various compute loads by composing and reusing existing—but currently hidden—compiler features without the need to implement new passes or even rebuilding the compiler.

We demonstrate in five case studies that the Transform dialect enables precise, safe composition of compiler transformations and allows for straightforward integration with state-of-the-art search methods.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Theory of computation** → *Program constructs.*

Keywords: MLIR, Transform Dialect, Transform Scripts, Controllable Compiler

ACM Reference Format:
Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. 2025. The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3696443.3708922

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 the intermediate representation (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 require 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 such as Halide [33] and TVM [5] address these concerns in the specific domains of image processing and machine learning. 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 [39].

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 contain 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 transform steps without compromising on their usability from other native code. By introducing the Transform dialect, we open 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 to fine-grained compiler transformation control (Section 3).
- A system of pre- and post-conditions to statically detect problems in compiler pipelines (Section 3.3).
- An evaluation of the Transform dialect with five case studies (Section 4) highlighting its low computational overhead and robustness, as well as its usefulness for detecting performance problems, generating high-performance code, and exploring optimization spaces.

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 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 [33], TVM [5], TACO [21], Lift [35] & RISE [15], and more, but also of new compiler frameworks to facilitate their development, most prominently MLIR [24]. Our hardware landscape has evolved as well, resulting in a growing diversity of accelerator architectures including GPUs, Google's TPU, Groq's LPU, or Cerebras' wafer-scale engine, only to name a few.

Optimizing programs in this new landscape is more challenging as it requires tweaking and adjusting optimizations for the target hardware architecture as well as 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.

2.1 Controlling Transformations Today in MLIR

MLIR [24] has emerged as a popular framework for building domain-specific compilers. It closely follows LLVM [23], but crucially allows to define custom abstractions for representing specific computations as well as custom compiler transformations. Programs are represented in a hierarchical SSA-based form[1] and are structured as control-flow graphs (CFG) of control-flow-free basic *blocks*. Each block has a set of *block arguments* and contains a sequence of *operations*,

[1]https://mlir.llvm.org/docs/LangRef

```
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 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.

each of which can produce multiple return *values*. Block arguments and values representing the results of prior operations are used as the operation's *operands*. Finally, operations might have *regions* containing a hierarchically nested CFG.

MLIR is extremely customizable, allowing users to define their own operations and types, arranged into libraries called *dialects*. It provides a rich plugin mechanism allowing new definitions to be injected without recompiling the compiler, as dynamic libraries or using declarative specifications [11].

Compiler transformations in MLIR are usually bundled into *passes*, following the design of LLVM. Passes in modern compilers are designed to be reusable in different pass pipeline configurations, enabling performance engineers to explore which pass pipeline configuration leads to the best performance for their program and target scenario.

However, the control via passes is coarse-grained as there is no universal way to restrict passes to a targeted part of the program. Furthermore, rearranging pass pipelines is restricted by implicit dependencies between passes that are not explicitly expressed beyond textual documentation.

A significant driver for implementing optimizations in MLIR is the domain of linear algebra computations found in ML workloads, particularly those expressed using *structured operations* in Linalg and related dialects [37]. Initially, high-impact transformations such as tiling and fusion, were orchestrated using a system designed for peephole optimization [27] 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 replaced them with a transformed equivalent. As more fine-grained controls are lacking, such patterns are applied greedily to the entire program until a fixed point is reached. This approach 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 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.

2.2 Scheduling Languages for Separating Computations and Optimization Decisions

Halide [33] 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 [15] 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 to enable the fine-grained control of compiler optimizations for a wide variety of settings. In the following, we describe such a scheduling language for MLIR implemented as the *Transform dialect*.

3 Transform Dialect: Representing and Controlling Transformations using IR

We introduce the Transform dialect by example, designing an optimization as composition of existing transformations and applying it to a program as shown in Figure 1. The Transform script shown in Figure 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, 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.

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 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.[2] 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.

[2]MLIR interfaces provide dynamic polymorphism, similarly to OOP.

Since transformations may not always succeed, the transform interpreter provides an error handling mechanism similar to exceptions.[3] 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.[4]

3.1 Dealing with a Mutable IR

Contrary to many purely functional approaches [15, 35, 38], 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, either 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, 8 in Figure 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 [1], 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 3.4.

[3]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 had been signaled and are implicitly no-ops in that case.

[4]https://mlir.llvm.org/docs/Tutorials/transform/

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.

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 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. Since transforms are 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 1a defines a macro that can be expanded in any other place using the `include` operation.[5]

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.

Furthermore, since transforms are represented as ordinary compiler IR, they can be subject to compiler rewrites themselves. For example, a loop tiling transformation itself 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.

3.3 Composability with Pre- and Post-Conditions

Most compiler optimizations 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

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

```
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 2. 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).

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 [40] as there are 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 2. In this example, the next lowerings should take care of converting CF and arithmetic operations towards 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 *operation interfaces* instead. Operation interfaces,[6] are introduced in MLIR to group similarly behaving operations, such as operations that perform a memory allocation or have a specific side-effect.

[6]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 3. IRDL definition of the `memref.subview` operation. Highlighted 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 4. constraints on input and output handles of `transform.expand_strided_metadata`

Advanced Pre- and Post-Condition. Some pre- or post-conditions of a transformation require more detail than just requiring the presence of certain operations. The Transform dialect supports advanced pre-/post-conditions by integrating the declarative IR Definition Language (IRDL) [11]. IRDL is primarily used to declaratively specify operations of dialects with their types and invariants. Figure 3 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 4, modifies `memref` dialect operations so that complex strided address computations are factored out and the remaining accesses are akin to trivial flat pointers [37]. These simplified accesses are characterized by trivial `subview` operations where the access offsets, sizes and strides are empty. Figure 3 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 4 and that we *do not* actually introduce a new operation.

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.

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 [28] or Python [25].

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. Transforms being merely MLIR operations, static analysis for handle 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 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 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 throughout 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.

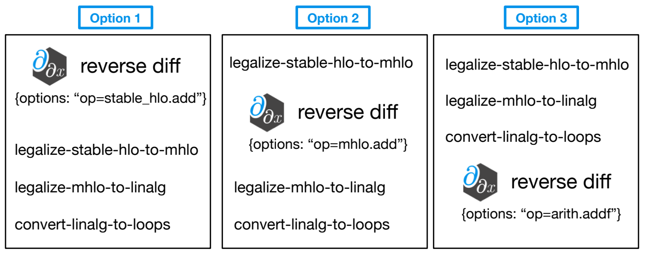

Figure 5. Three alternative options for when to perform the automatic differentiation transformation. We use a transform to introspect the pipeline and automatically infer the appropriate transformation option.

Automatically configuring transformation pipelines 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 [29]. AD produces instructions computing a derivative of a given variable. The full derivative is systematically a sum of partial derivatives with 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 [13], the IR is progressively rewritten from StableHLO[7] to MHLO[8], 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 5. However, we must manually configure the automatic differentiation pass [29] 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.

[7]https://openxla.org/stablehlo

[8]https://www.tensorflow.org/mlir/hlo_ops

Table 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.

Model	# Ops	Compile Time (ms)	
		MLIR	Transform
Squeezenet [19]	126	16.6	16.9
GPT-2 [32]	2861	185.4	190.0
Mobile BERT [36]	4134	316.7	317.7
Whisper (decoder only) [31]	847	457.5	462.3
BERT-base-uncased [8]	1182	1315.3	1348.6

3.5 Summary

In this section, we introduced the Transform dialect for composing and controlling compiler transformations. We discussed various features, including handle invalidation, extensiblity, ensuring composability via pre-/post-conditions, and the representation of Transform scripts as MLIR IR. Next, we are evaluating transform using a number of case studies.

4 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 explore compiler optimizations.

4.1 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 [33] and ELEVATE [15] 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.

We evaluate the overhead of using the Transform dialect using five machine learning models listed in Table 1 and implemented with the MLIR-based TensorFlow compiler ecosystem. The following pass pipeline from the MLIR Tensor

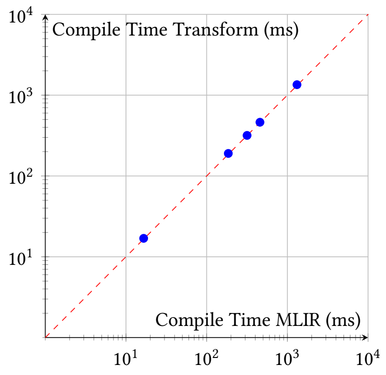

Figure 6. Compile time comparison of different models using regular MLIR vs using Transform Dialect.

Operation Set Architecture (TOSA) dialect[9] to the Linalg dialect [37] is described in [3]:

```
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)
```

To obtain a Transform script describing the identical compilation flow, we modified MLIR to automatically create a Transform script of a pass pipeline that 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 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 1 and Figure 6 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.

4.2 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 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

[9] https://www.mlplatform.org/tosa

```
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]
  }
}
```

Figure 7. An example of an MLIR function using core dialects.

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 out 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. An example of a simple program that leverages all of these dialects is presented in Figure 7. 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:

① `convert-scf-to-cf`
Lower structured control flow (`scf.forall`) to basic blocks and branching instructions.

② `convert-arith-to-llvm`
Lower arithmetic dialect 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 provided as an additional function argument, `%offset : index`. This results in the following error: *"failed to legalize operation builtin.unrealized_conversion_cast 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 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 [4, 10], in the input program. Moreover, this dialect may be seen as operating on a conceptually higher level than memory address indexing due to 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 to compose robust pass pipelines.

An ad-hoc solution to this problem is to simply add the `lower-affine` pass and other transitively required lowering passes after the `-expand-strided-metadata` pass to lower all created operations from the `affine` dialect, 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 2. Pre-conditions indicate the payload operations that are expected in the input and will be removed, other operations won't be modified. Post-conditions indicate new payload operations that will be produced. 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.

4.3 Case Study 3: Debugging Performance Problematic Optimization Patterns

Besides ensuring that pipelines are correctly producing code, performance engineers often also have to deal with counterproductive effects of program optimizations, chasing and eliminating them.

For instance, while introducing additional peephole optimization patterns for StableHLO that is a part of the Enzyme AD workflow, we observed that a combination of over 100 work-reducing and enabling transformations yielded counterproductive results in one of the LLMs we were trying to optimize, with up to 9% overall performance penalty compared

Table 2. Pre-/post-conditions conditions indicate payload operations removed/introduced by a transform, or additional constraints from Figure 3. 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.

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	$\{cast\}$	$\{\}$

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 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.[10] 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 follows.[11]

```
transform.apply_patterns to %func {
  transform.pattern.add_of_zero_pad
  transform.pattern.negate_of_transpose
  transform.pattern.matmul_of_transpose
  // more patterns
}
```

This allowed us to only deploy the compiler once and to automate the binary search over the pattern set by simply removing 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. Thanks to this, we identified the counter-productive 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[12] that produced larger, less cache-efficient fusion clusters.

This case study demonstrates the usefulness of flexible and fine-grained compiler control that enables quickly exploring the space of transformations, in this instance to detect a performance problem that affects only a single, but important, input program.

```
for (int b=0; b<6; b++) {
  #pragma omp tile sizes(32, 32)
  for(int i=0; i<196; i++) {
    for(int j=0; j<256; j++) {
      for(int k=0; k<2305; k++) {
        C[b,i,j] += A[b,i,k] * B[b,k,j];
      }
    }
  }
}
```

Figure 8. OpenMP pragmas for tiling a single loop nest.

4.4 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., for a single loop nest of a layer of the ResNet-50 model [16]. 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 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 tiling, as shown in Figure 8. Such transformations

[10] LLVM toolchain `3b5e7c83a6e` from Mar 14, 2024
[11] https://github.com/EnzymeAD/Enzyme-JAX/commit/396c8c1384a32fa726121e5fed877776e3c6ee6b
[12] https://openxla.org/xla

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 the line 2 of Figure 9, 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 allow 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.

While OpenMP is limited to perform 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 [17]. 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 9. 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 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 0.017s, over 20 times 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.

4.5 Case Study 5: Performance Exploration with State-of-the-Art Autotuning Methods

In this case study, we demonstrate how we can use Transform scripts to quickly explore an optimization space. For this, we use the state-of-the-art bayesian autotuning tool BACO [18] to automatically explore the search space for tiling the loop nest shown in Section 4.4 according to the tuning parameters and constraints shown in Figure 11. Figure 12 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.

```
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 9. 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.

```
transform.named_sequence @transform_main(%module) {
  %batch_loop = match.op {first} "scf.for" in %module
  %tiled:4 = loop.tile %batch_loop
              {tile_sizes=[tile0,tile1,tile2,tile3]}
  transform.alternatives {
    transform.to_library %tiled#4 "libxsmm"
  }, {
    transform.assert vect
    transform.vectorize %tiled#4
  }, { }
}
```

Figure 10. Transform script with the parametric tile sizes that are automatically chosen using an autotuning tool.

```
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 11. Definition of the tuning parameters used in Figure 10 with constraints: tile sizes must divide their dimension and vectorization is disabled if the trip count of the inner-most loop is not divisible by the machine vector size.

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.

5 Related Work

Controlling Compiler Optimizations. There are many alternative approaches to control compiler optimizations ranging from manual annotations, such as OpenMP, over simple compiler flags, such as `-O3`, to more complex compiler flags, such as Polly's influencing heuristics [14]. These knobs can be used to search for the best performing combination of optimizations, e.g., tuning flags via Monte-Carlo search [22] or machine learning methods [7]. Exposing compiler heuristics to make them accessible is an active research area [34].

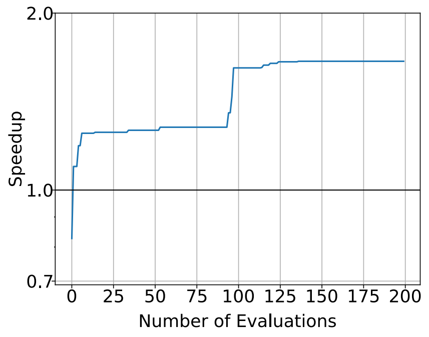

Figure 12. Performance evolution of batch matrix multiplication when searching for the best tile size parameters in the Transform script specifying the optimization.

URUK [6] and Clay [2] define directives for source-to-source polyhedral transformation. SPIRAL [12] provides uses high-level synthesis to produce efficient kernels from abstract descriptions. Source matching and rewriting has also been proposed for MLIR [9].

The Transform dialect offers a significantly finer-grained control of compiler transformations. As our final case study showed this can also be combined with autotuning methods to explore the optimization space.

Scheduling Languages. Halide [33], TVM [5], TACO [21], and Rise [15] are examples of domain-specific compilers that have scheduling languages allowing performance experts to control compiler optimizations. Besides being domain-specific, the transformations exposed by the compilers are also fixed. In contrast, the Transform dialect is an extensible scheduling language available to generic compilers.

ELEVATE [15] is a scheduling language expressing compiler optimizations as compositions of formal rewrite rules. It provides formal reasoning capabilities for compiler transformations [30]. While generic and extensible, ELEVATE requires optimizations expressed as formal rewrites, requiring computational programs to be pure to apply safely. The Transform dialect enables composing and controlling of compiler transformations over a wider range of scenarios, in a pre-existing compiler framework.

Exo [20] makes scheduling languages a programmable extension to the compiler, by using algebraic rewrites within a core language instead of a specialized code generator template, and by generalizing Halide schedules to imperative code and relying on an SMT solver to verify side-effecting computations. Unlike the Transform dialect, Exo does not handle extensible intermediate representations (the core language is fixed), and thus cannot operate at the different levels of abstraction needed to operate on the pass pipeline itself, to reuse existing transformation passes, and does not decouple effecting transformations from their validation.

6 Conclusion

We presented and evaluated the MLIR Transform dialect. It allows for fine-grained control of compiler optimizations while maintaining the extensible nature of MLIR. Considering 5 case studies we have demonstrated its low overhead, and usefulness to define robust, flexible and performant compiler pipelines.

Its implementation in the most popular domain-specific compiler framework opens up many interesting applications to further exploit the already existing power of compilers in as many scenarios as possible.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 2346519. This work is also supported by the Alan Turing Institute.

A Artifact Appendix

A.1 Abstract

This presents the artifact to supplement the CGO 2025 paper "The MLIR Transform Dialect - Your compiler is more powerful than you think". It includes the code and scripts to facilitate the reproduction of the experiments presented in the paper. Additionally it contains the tool `mlir-transform-opt` that represents the MLIR compiler infrastructure completely controllable via a transform script as presented in the paper.

A.2 Artifact Check-List (Meta-Information)

- **Tool:** MLIR infrastructure including the transform dialect and corresponding transformations.
- **Transformations:** Custom MLIR pass pipelines, transform scripts
- **Data set:** Models such as Bert, Mobile-Bert, ResNet, Whisper, SqueezeNet, GPT2, LLaMa 2
- **Run-time environment:** Docker container
- **Hardware:** General x86/arm-based hardware, we used an Apple M3.
- **Execution:** Scripts for automated and customized experiments
- **Metrics::** Compile time, runtime performance, demonstration of certain workflows
- **Output:** CSV files, performance metrics, graphs
- **Experiments:** Case studies on Transform dialect usage in MLIR
- **How much disk space required (approximately)?:** 20 GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes if the provided docker image is used, when building from source 3h
- **How much time is needed to complete experiments (approximately)?:** 2-4h depending on search time in case study 5.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.13373470 [26]

A.3 Description

A.3.1 How Delivered. The artifact is delivered as an archived docker image on Zenodo (for amd64 architectures, other architectures require building from source). Additionally a public GitHub repository contains all necessary code and scripts, along with a Dockerfile for setting up the environment from scratch.

A.3.2 Software Dependencies. Docker and GNU make available in `$PATH`.

A.3.3 Data Sets. The data sets include various machine learning models that are automatically downloaded during the building of the artifact.

A.4 Installation

Option 1: Loading the Provided Docker Image.

```
# Download docker image
  wget path/to/zenodo/transform_artifact.tar.gz
# Unpack docker image
  tar -xvf transform_artifact.tar.gz
# Load the docker image
  docker load --input transform_artifact.tar
# Run the image interactively from the /home folder
     in the image
  docker run -it -w /home transform_artifact zsh
```

Option 2: Building the Docker Image from Source.

```
# Clone the repository
  git clone https://github.com/martin-luecke/
     cgo25_transform_artifact.git && cd
     cgo25_transform_artifact
# Initialize all git submodules
  git submodule update --init
# Build the docker image - ~3h
  make all
# Run the image interactively from the /home folder
     in the image
  make run
```

A.5 Experiment Workflow

Running All Experiments.

```
# Run all experiments:
  /home/run_all.sh
```

This script runs the experiments of case studies 1,3,4 and 5 and automatically saves the results in the folder `/home/results` of the docker container.

A.6 Notes

The file `README.txt` in the artifact contains detailed information on the expected results and options for customization for each of the case studies. Additionally, it provides step-by-step instructions on how to run the individual case studies, ensuring that users can easily understand and execute them as intended.

References

[1] Andrew W Appel. 1998. SSA is functional programming. *Acm Sigplan Notices* 33, 4 (1998), 17–20. https://doi.org/10.1145/278283.278285

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 128–138. https://doi.org/10.1145/2854038.2854048

[3] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 386–407. https://doi.org/10.1007/978-3-031-13188-2_19

[4] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. https://doi.org/10.48550/arXiv.2003.00532 arXiv:2003.00532 [cs.PF]

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[6] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th annual international conference on Supercomputing*. 151–160. https://doi.org/10.1145/1088149.1088169

[7] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 92–105. https://doi.org/10.1109/CGO53902.2022.9741258

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[9] Vinicius Espindola, Luciano Zago, Hervé Yviquel, and Guido Araujo. 2023. Source matching and rewriting for MLIR using string-based automata. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–26. https://doi.org/10.1145/3571283

[10] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[11] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: an IR definition language for SSA compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 199–212. https://doi.org/10.1145/3519939.3523700

[12] 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. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[13] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[14] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1142/S0129626412500107

[15] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[17] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. 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*. 981–991. https://doi.org/10.1109/SC.2016.83

[18] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2024. 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* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 19–42. https://doi.org/10.1145/3623278.3624770

[19] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. https://doi.org/10.48550/arXiv.1602.07360 arXiv:1602.07360 [cs.CV]

[20] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[21] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. https://doi.org/10.1145/3133901

[22] Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul D. Hovland, and Mary W. Hall. 2021. Customized Monte Carlo Tree Search for LLVM/Polly's Composable Loop Optimization Transformations. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2021), St. Louis, MO, USA, November 15, 2021*. IEEE, 82–93. https://doi.org/10.1109/PMBS54543.2021.00015

[23] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[25] Maksim Levental, Alok Kamatar, Ryan Chard, Kyle Chard, and Ian Foster. 2023. nelli: a lightweight frontend for MLIR. https://doi.org/10.48550/arXiv.2307.16080 arXiv:2307.16080 [cs.PL]

[26] Martin Paul Lücke, Michel Steuwer, William Moses, Albert Cohen, and Oleksandr Zinenko. 2024. *Artifact for the paper "The MLIR Transform Dialect - Your compiler is more powerful than you think"*. https://doi.org/10.5281/zenodo.13373470

[27] William M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444. https://doi.org/10.1145/364995.365000

[28] Jules Merckx. 2024. Building Bridges: Julia as an MLIR Frontend. https://libstore.ugent.be/fulltxt/RUG01/003/212/846/RUG01-003212846_2024_0001_AC.pdf. Master's thesis, Ghent University.

[29] William Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12472–12485. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

[30] Xueying Qin, Liam O'Connor, Rob van Glabbeek, Peter Höfner, Ohad Kammar, and Michel Steuwer. 2024. Shoggoth: A Formal Foundation for Strategic Rewriting. *Proc. ACM Program. Lang.* 8, POPL (2024), 61–89. https://doi.org/10.1145/3633211

[31] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 28492–28518. https://proceedings.mlr.press/v202/radford23a.html

[32] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Notices* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[34] Volker Seeker, Chris Cummins, Murray Cole, Björn Franke, Kim M. Hazelwood, and Hugh Leather. 2024. Revealing Compiler Heuristics Through Automated Discovery and Optimization. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira (Eds.). IEEE, 55–66. https://doi.org/10.1109/CGO57630.2024.10444847

[35] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

[36] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 2158–2170. https://doi.org/10.18653/v1/2020.acl-main.195

[37] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2023. Structured Operations: Modular Design of Code Generators for Tensor Compilers. In *Languages and Compilers for Parallel Computing*, Charith Mendis and Lawrence Rauchwerger (Eds.). Springer Nature Switzerland, Cham, 141–156.

[38] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[39] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[40] Oleksandr Zinenko. 2023. MLIR Is Not an ML Compiler – And Other Common Misconceptions. https://llvm.org/devmtg/2023-10/slides/techtalks/Zinenko-MLIRisNotAnMLCompiler.pdf

Received 2024-05-30; accepted 2024-07-22