A Listener-based Approach to Discardable Attribute Conversion/Propagation

EuroLLVM 2025, MLIR Workshop

Matthias Springer – NVIDIA

Some Thoughts on how to Deal with Discardable Attributes

EuroLLVM 2025, MLIR Workshop

Matthias Springer – NVIDIA

Inherent and Discardable Attributes

```
%r = arith.cmpi eq, %a, %b { test.my_attr = 2 : i32 } : index
                ^^           ^^^^^^^^^^^^^^^^^^^^^^^^
             inherent              discardable
```

- *inherent attributes* are inherent to the definition of an operation's semantics. The operation itself is expected to verify the consistency of these attributes. An example is the `predicate` attribute of the `arith.cmpi` op. These attributes must have names that do not start with a dialect prefix.
- *discardable attributes* have semantics defined externally to the operation itself, but must be compatible with the operations's semantics. These attributes must have names that start with a dialect prefix. The dialect indicated by the dialect prefix is expected to verify these attributes. An example is the `gpu.container_module` attribute.

Examples of Discardable Attributes

In MLIR:

- bufferization.buffer_layout = affine_map<(d0) -> (d0 + 5)>
- dlti.dl_spec = #dlti.dl_spec<#dlti.dl_entry<index, 32>>
- transform.silence_tracking_failures
- llvm.loop_annotation = #llvm.loop_annotation<#llvm.loop_unroll<disable = true>>
- llvm.noalias
- __internal_linalg_transform__

In other projects:

- tf.device = "/CPU:0"
- tt.divisibility = 16 : i32

Open Question (Discourse)

How to deal with unknown, discardable attributes in transformations?

1. **Drop discardable attributes:** safe, but information is lost.
2. **Propagate discardable attributes:** attribute may be semantically incorrect on new / in-place updated operation.
3. ~~**Convert attribute:** not possible, attribute is unknown.~~
4. **Block the transformation:** always safe.

Is this a Problem in Practice?

On querying an Operation's intrinsic (core) vs external/user-defined attributes

MLIR

B

bondhugula Oct 2021

This issue came up in the context of this revision: https://reviews.llvm.org/D111837 7 , and it'd be good to get thoughts on this. It's on the propagation of user-defined attributes when ops are replaced during rewrites and transforms.

Oct 2021

1 / 28
Oct 2021

[RFC] Implicit propagation of dialect attributes (best effort)

MLIR

mehdi_amini 1 Jan 2021

Hi all,

I've been trying to address an issue we have with the TensorFlow dialect, but that may apply to other domain as well related to the implicit propagation of dialect attributes.

Jan 2021

1 / 18
Jan 2021

Save unregistered attrs after type conversion. #135084

Open **bartchr808** wants to merge 1 commit into `llvm:main` from `bartchr808:func-conversion-unregistered-attrs`

Conversation 4 | Commits 1 | Checks 10 | Files changed 1

bartchr808 commented 11 hours ago — Member

In Shardy, we rely on unregistered attrs being preserved between conversion of different dialects. At one point, we convert between StableHLO and MHLO. When this occurs, we have a `CallOp` with type `stablehlo.token` that is converted to `mhlo.token`. When this occurs, we lose attributes due to this pattern not preserving them. This change makes sure they are preserved.

[mlir][Tensor] Retain discardable attrs in pack(cast) folder #115772

Merged **qedawkins** merged 1 commit into `llvm:main` from `qedawkins:keep_pack_discardable_attrs` on Nov 12, 2024

Differential > D111837

Retain attributes of original scf::ForOp when folding

Needs Revision | Public

Authored by nimiwio on Oct 14 2021, 2:00 PM.

Details

Reviewers
- nicolasvasilache
- mehdi_amini
- bondhugula

SUMMARY

ForOpIterArgsFolder and ForOpTensorCastFolder construct a new ForOp, but don't retain any pre-existing attributes.

Differential › D126320

[mlir][scf] Retain existing attributes in scf.for transforms

Closed | Public

Authored by antiagainst on May 24 2022, 12:45 PM.

Details

Reviewers
- hanchung
- ThomasRaoux
- mravishankar

Commits rG413fbb045d71: [mlir][scf] Retain existing attributes in scf.for transforms

SUMMARY

These attributes can carry useful information, e.g., pipelines
might use them to organize and chain patterns.

```cpp
/// Convert the destination block signature (if necessary) and lower the branch
/// op to llvm.br.
struct BranchOpLowering : public ConvertOpToLLVMPattern<cf::BranchOp> {
  using ConvertOpToLLVMPattern<cf::BranchOp>::ConvertOpToLLVMPattern;

  LogicalResult
  matchAndRewrite(cf::BranchOp op, typename cf::BranchOp::Adaptor adaptor,
                  ConversionPatternRewriter &rewriter) const override {
    FailureOr<Block *> convertedBlock =
        getConvertedBlock(rewriter, getTypeConverter(), op, op.getSuccessor(),
                          TypeRange(adaptor.getOperands()));
    if (failed(convertedBlock))
      return failure();
    Operation *newOp = rewriter.replaceOpWithNewOp<LLVM::BrOp>(
        op, adaptor.getOperands(), *convertedBlock);
    // TODO: We should not just forward all attributes like that. But there are
    // existing Flang tests that depend on this behavior.
    newOp->setAttrs(op->getAttrDictionary());
    return success();
  }
};
```

```cpp
LogicalResult ForLowering::matchAndRewrite(ForOp forOp,
```

```cpp
  // Let the CondBranchOp carry the LLVM attributes from the ForOp, such as the
  // llvm.loop_annotation attribute.
  SmallVector<NamedAttribute> llvmAttrs;
  llvm::copy_if(forOp->getAttrs(), std::back_inserter(llvmAttrs),
                [](auto attr) {
                  return isa<LLVM::LLVMDialect>(attr.getValue().getDialect());
                });
  condBranchOp->setDiscardableAttrs(llvmAttrs);
  // The result of the loop operation is the values of the condition block
  // arguments except the induction variable on the last iteration.
  rewriter.replaceOp(forOp, conditionBlock->getArguments().drop_front());
  return success();
}
```

Attribute Propagation in MLIR Today

Patterns

- `BranchOpLowering`: cf.br → llvm.br lowering
- `CondBranchOpLowering`: cf.cond_br → llvm.cond_br lowering
- `ForLowering`: scf.for → CF lowering
- `FoldTensorCastPackOp`: fold cast into tensor.pack
- `FoldTensorCastUnPackOp`: fold cast into tensor.unpack
- `ConvertForOpTypes`: type conversion pattern for scf.for
- `ConvertIfOpTypes`: type conversion pattern for scf.if
- `ForOpIterArgsFolder`: fold iter_args of scf.for

Bufferization

- `ForallOpInterface`: bufferization of scf.forall
- `CallOpInterface`: bufferization of func.call
- `ForOpInterface`: bufferization of scf.for

Helpers

- `intrinsicRewrite`: Replace op with llvm.call_intrinsic
- `ForOp::replaceWithAdditionalYields`
- `scf::replaceAndCastForOpIterArg`

Observations

- Discardable attribute handling is tricky in **shared transformations / patterns**.
  - This is not a problem when transformations **and** attributes are defined in your project.
- This problem appeared only in the context of “simple” canoncalizations and conversion patterns so far.
  - Structural conversion patterns: Type conversion of scf.for, scf.if, etc.
  - Folding of cast operations.

Blindly Propagate all Discardable Attributes

Pros / Cons

- Propagation may be semantically incorrect.
    - Attribute and IR can get out of sync.
    - Is it correct to propagate attributes from one op to an op of different type?
    - What if two ops are replaced by one op. How to merge two attributes?
    - A transformation / pattern may create multiple ops and it’s unclear to which to propagate.
- Transformations / patterns must be updated and you may accidentally miss some. Attributes will be silently dropped. (Difficult to debug.)

Example: Analysis Attribute going Out-of-sync

Before loop pipelining:

```
scf.for %arg2 = %c0 to %c4 step %c1 {
  %0 = memref.load %arg0[%arg2] : memref<?xf32>
  %1 = arith.addf %0, %cst : f32
  memref.store %1, %arg1[%arg2] : memref<?xf32>
} {cost_per_iteration = 3}
```

After loop pipelining:

```
%0 = memref.load %arg0[%c0] : memref<?xf32>
%1 = scf.for %arg2 = %c0 to %c3 step %c1 iter_args(%arg3 = %0) -> (f32) {
  %3 = arith.addf %arg3, %cst : f32
  memref.store %3, %arg1[%arg2] : memref<?xf32>
  %4 = arith.addi %arg2, %c1 : index
  %5 = memref.load %arg0[%4] : memref<?xf32>
  scf.yield %5 : f32
} {cost_per_iteration = 3}  // should be 4
%2 = arith.addf %1, %cst : f32
memref.store %2, %arg1[%c3] : memref<?xf32>
```

Attach Metadata with Assume Operation: An Alternative to Discardable Attributes

Example: Triton Axis Analysis

```
func.func @kernel(%arg0: i64 {tt.divisibility = 16 : i32}) {

  %1 = tt.int_to_ptr %arg0
      : i64 -> !tt.ptr<f32>

  …
```

compiler hint, dropping this
may prevent vectorization

Example: Triton Axis Analysis

```
func.func @kernel(%arg0: i64) {
  %0 = tt.assume_div_by %arg0, 16 : i64
  %1 = tt.int_to_ptr %0
        : i64 -> !tt.ptr<f32>
  ...
```

Pros / Cons

- Not applicable to operations: Assumptions are attached to SSA values.
    - Not suitable for marking operations: Your “tag” may move from the op result of a `linalg.matmul` to the op result of a loop nest.
    - Cannot express metrics such as “cost per loop iteration” or “cache hit rate of a load op”.
- Can attribute and IR can get out of sync?
    - Given that transformations must preserve correctness, properties of SSA values are generally more likely to be preserved than properties of operations. (Value describes the result, operation describes how to get to the result.)
- Existing transformations / patterns cannot accidentally drop information.
- Existing transformations / patterns may no longer apply:

  New canonicalization patterns, foldings, analyses, etc. are needed.

What kind of properties are safe assume?

- Transformations must preserve functional correctness.
- Probably safe to assume: Value semantics properties of ints, floats, tensors, vectors.
    - Divisibility of an i32 value.
    - Number of elements in a tensor.
    - Maximum value in a vector.
- Unclear
    - Alignment of a memref value. (A program with different buffers can compute the same thing.)
    - Number of threads waiting for an `!async.token`.

Assume Op as Optimization Barrier

```
%c = tensor.cast %t1
     : tensor<5xi64> to tensor<?xi64>

%a = tt.assume_div_by %c, 16 : tensor<?xi64>

%r = tensor.insert_slice %t2 into %a [0] [?] [1]
     : tensor<?xi64> to tensor<?xi64>
```

folding / canonicalization barrier

Attach Metadata to Transform Dialect Handle: An Alternative to Discardable Attributes

Input IR before Tiling

```
func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>,
                              %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {

  %0 = linalg.matmul  ins(%arg0, %arg1: tensor<128x128xf32>, tensor<128x128xf32>)
                     outs(%arg2: tensor<128x128xf32>) -> tensor<128x128xf32>

  return %0 : tensor<128x128xf32>
}
```

Let’s tile this `linalg.matmul`!

Input IR after Tiling

```
func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>,
                              %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {
  %0 = scf.for %arg3 = %c0 to %c128 step %c4 iter_args(%arg4 = %arg2) -> (tensor<128x128xf32>) {
    %1 = scf.for %arg5 = %c0 to %c128 step %c4 iter_args(%arg6 = %arg4) -> (tensor<128x128xf32>) {
      %2 = scf.for %arg7 = %c0 to %c128 step %c4 iter_args(%arg8 = %arg6) -> (tensor<128x128xf32>)
{
        %e0 = tensor.extract_slice %arg0[%arg3, %arg7] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>
        %e2 = tensor.extract_slice %arg1[%arg7, %arg5] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>
        %e2 = tensor.extract_slice %arg8[%arg3, %arg5] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>

        %3 = linalg.matmul ins(%e0, %e1 : tensor<4x4xf32>, tensor<4x4xf32>)
                          outs(%e2 : tensor<4x4xf32>) -> tensor<4x4xf32>

        %i0 = tensor.insert_slice %3 into %arg8[%arg3, %arg5] [4, 4] [1, 1]
            : tensor<4x4xf32> into tensor<128x128xf32>
        scf.yield %i0 : tensor<128x128xf32>
      }
      scf.yield %2 : tensor<128x128xf32>
    }
    scf.yield %1 : tensor<128x128xf32>
  }
```

Input IR before Tiling

```
func.func @tile_linalg_matmul(%arg0: tensor<128x128xf32>, %arg1: tensor<128x128xf32>,
                              %arg2: tensor<128x128xf32>) -> tensor<128x128xf32> {
  %0 = scf.for %arg3 = %c0 to %c128 step %c4 iter_args(%arg4 = %arg2) -> (tensor<128x128xf32>) {
    %1 = scf.for %arg5 = %c0 to %c128 step %c4 iter_args(%arg6 = %arg4) -> (tensor<128x128xf32>) {
      %2 = scf.for %arg7 = %c0 to %c128 step %c4 iter_args(%arg8 = %arg6) -> (tensor<128x128xf32>)
{
        %e0 = tensor.extract_slice %arg0[%arg3, %arg7] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>
        %e2 = tensor.extract_slice %arg1[%arg7, %arg5] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>
        %e2 = tensor.extract_slice %arg8[%arg3, %arg5] [4, 4] [1, 1]
            : tensor<128x128xf32> to tensor<4x4xf32>

        %3 = linalg.matmul ins(%e0, %e1 : tensor<4x4xf32>, tensor<4x4xf32>)
                           outs(%e2 : tensor<4x4xf32>) -> tensor<4x4xf32>

        %i0 = tensor.insert_slice %3 into %arg
            : tensor<4x4xf32> into tensor<128x
        scf.yield %i0 : tensor<128x128xf32>
      }
      scf.yield %2 : tensor<128x128xf32>
    }
    scf.yield %1 : tensor<128x128xf32>
  }
```

Let’s tile the “same” `linalg.matmul`!

How to “find” the `linalg.matmul` after Tiling?

- We used to attach a discardable attribute during the tiling transformation: `__internal_linalg_transform__`
- Better: Drive the transformation with the transform dialect and store the `linalg.matmul` in a handle.
- Can not only be used for “marking” ops but also to attach additional information. (Transform ops can store additional state in the interpreter.)

Example: Transform Dialect Script

```
transform.named_sequence @__transform_main(%arg1: !transform.any_op {transform.readonly}) {
  %0 = transform.structured.match ops{["linalg.matmul"]} in %arg1
      : (!transform.any_op) -> !transform.any_op
  %1, %loops1:3 = transform.structured.tile_using_for %0 tile_sizes [4, 4, 4]
      : (!transform.any_op) -> (!transform.any_op, !transform.any_op, !transform.any_op,
         !transform.any_op)
  %2, %loops2:3 = transform.structured.tile_using_for %1 tile_sizes [2, 2, 2]
      : (!transform.any_op) -> (!transform.any_op, !transform.any_op, !transform.any_op,
         !transform.any_op)
  transform.yield
}
```

Pros / Cons

- Both operations and values can be annotated.
- Existing patterns can be reused: `transform.apply_patterns`
  - Handle update is fragile: it's based on listener notifications. E.g.: `replaceOpWithNewOp` updates the handle, but `replaceAllUsesWith+eraseOp` drops the handle.
  - Heuristics are needed to skip over cast ops, etc. E.g., replacing `OpA` with `cast(OpA)`.
- Metadata (stored in the transform interpreter) and IR can get out of sync.

Listener-based Attribute Propagation / Conversion

Listeners in MLIR

- Can be attached to builders and rewriters. Fully supported in greedy pattern rewrite, limited supported in dialect conversions.
- Listen to: op creation, op movement, op replacement, etc.

```
notifyOperationReplaced(Operation* oldOp, Operation *newOp);
```

Example: Propagation Listener

```
class DiscardableAttributeConverter : public RewriterBase::Listener {
public:
  void notifyOperationErased(Operation *op) override {
   op->walk([](Operation *op) {
      assert(op->getDiscardableAttrs().empty() &&
             "attempting to drop discardable attribute");
    });
  }

  void notifyOperationReplaced(Operation *op, Operation *replacement) override {
    // Custom handling: Remove discardable attributes from op, add to replacement.
  }
  void notifyOperationModified(Operation *op) override { /* Drop attribute if invalid. */ }
};

DiscardableAttributeConverter listener;
GreedyRewriteConfig config;
config.listener = &listener;
applyPatternsGreedily(op, patterns, config);
```

Pros / Cons

- Existing patterns and transformations with listener support can be reused. In particular: **greedy pattern rewrite**, soon **dialect conversion**.
- No silent dropping of discardable attributes (failed assertion).
- **Attribute propagation / conversion rules are offloaded to the user** of the transformation, who should be aware of the entire compilation pipeline.
- Existing passes (e.g., `-convert-to-llvm`) cannot be reused. You have to write your own pass that populates the patterns and attaches the listener.
- Many core transformations (e.g., CSE) do not have listener support.
- Listener notifications are fragile: Same limitations wrt. `replaceAllUsesWith+eraseOp` as with the transform dialect approach.
- Does not take into account foldings. (No replacement callback.)

Questions?

Blindly Propagate all Attributes
Attach Metadata with Assume Op
Attach Metadata to Transform Handle
Listener-based Attribute Propagation

Inherent Attribute
Discardable Attribute
Attribute Propagation
Attribute Dropping
Attribute Conversion
Greedy Pattern Rewrite
Dialect Conversion

Canonicalization Pattern
CSE
Transform Dialect
Analysis Metadata
Builder / Rewriter Listener
Propagation Listener
`replaceAllUsesWith` / `replaceOp`
`eraseOp`
Fold Operation
Reusing Existing Passes
Correctness of Propagation / Dropping