

MLIR Bufferization: From Tensors to MemRefs

Tutorial – 2023 LLVM Developers' Meeting – Oct. 12, 2023

Matthias Springer (springerm@google.com)

Martin Erhart (martinerhart12@gmail.com)

<http://tiny.cc/3wxbvz>

Google Research

Running Example

<http://tiny.cc/3wxbvz>

llvm/llvm-project@8763343

Feel free to follow along on your laptop.

```
// Batched TOSA matrix multiplication. %A and %B are the
// inputs, %C is the output.
func.func @test_matmul(%A: memref<1x17x19xf32>,
                      %B: memref<1x19x29xf32>,
                      %C: memref<1x17x29xf32>) {

  %A_tensor = bufferization.to_tensor %A restrict
              : memref<1x17x19xf32>
  %B_tensor = bufferization.to_tensor %B restrict
              : memref<1x19x29xf32>

  %0 = tosa.matmul %A_tensor, %B_tensor
      : (tensor<1x17x19xf32>, tensor<1x19x29xf32>) ->
        tensor<1x17x29xf32>

  bufferization.materialize_in_destination
  %0 in restrict writable %C
      : (tensor<1x17x29xf32>, memref<1x17x29xf32>) -> ()

  return
}
```

Running Example

to_tensor is a bufferization-specific unrealized_conversion_cast

<http://tiny.cc/3wxbvz>

llvm/llvm-project@8763343

Feel free to follow along on your laptop.

computation (kernel) is written in tensor IR

output function argument instead of return value

assuming that buffers are allocated by a runtime

```
// Batched TOSA matrix multiplication. %A and %B are the
// inputs, %C is the output.
func.func @test_matmul(%A: memref<1x17x19xf32>,
                      %B: memref<1x19x29xf32>,
                      %C: memref<1x17x29xf32>) {
```

```
%A_tensor = bufferization.to_tensor %A restrict
            : memref<1x17x19xf32>
%B_tensor = bufferization.to_tensor %B restrict
            : memref<1x19x29xf32>
```

```
%0 = tosa.matmul %A_tensor, %B_tensor
      : (tensor<1x17x19xf32>, tensor<1x19x29xf32>) ->
        tensor<1x17x29xf32>
```

```
bufferization.materialize_in_destination
%0 in restrict writable %C
      : (tensor<1x17x29xf32>, memref<1x17x29xf32>) -> ()
```

```
return
}
```

result should be stored in %C

bufferization .to_tensor

- A bufferization-specific `unrealized_conversion_cast`
- Bufferization can be stopped at any point and you can examine the partially bufferized IR
- **restrict**: indicates this op is the only way for the tensor IR to gain access to the memref operand (or an alias thereof)
- **writable**: indicates that the buffer is writable
- There is also `bufferization.to_memref`

Invalid IR (not checked by the verifier):
`%B_view` and `%B_view_tensor` expose aliasing buffers

```
func.func @test(%A: memref<1x17x19xf32>,  
               %B: memref<1x19x29xf32>,  
               %C: memref<1x17x29xf32>) {  
  
  %A_tensor = bufferization.to_tensor %A restrict  
             : memref<1x17x19xf32>  
  %B_tensor = bufferization.to_tensor %B restrict  
             : memref<1x19x29xf32>  
  
  %B_view = memref.subview %B[0, 2, 3] [1, %sz2, %sz3] ...  
  %B_view_tensor = bufferization.to_tensor %B_view restrict  
                  : memref<1x?x?xf32>  
  
  %C_tensor = bufferization.to_tensor %B restrict writable  
             : memref<1x17x29xf32>  
  
}
```

Bufferization Infrastructure

Preprocessing

rewrite_in_destination_passing_style

Bufferization

-eliminate-empty-tensors

-one-shot-bufferize

-buffer-hoisting

-buffer-loop-hoisting

-buffer-results-to-out-params

-drop-equivalent-buffer-results

-promote-buffers-to-stack

Buffer-Level
Optimizations

-buffer-deallocation-pipeline

Deallocation

~~-buffer-deallocation~~

01

Bufferization

Bufferization

- Lower tensor IR to memref IR
- Pass: `-one-shot-bufferize`
- Transform dialect op:
`transform.bufferization.one_shot_bufferize`
- Op interface driven: `BufferizableOpInterface`
- Function calls: recursion is not supported

Bufferization

`-one-shot-bufferize`

Bufferization Pass

```
mlir-opt %s  
--one-shot-bufferize
```

Bufferization

-one-shot-bufferize

```
func.func @test(%t: tensor<8xf32>, %idx: index)  
  -> tensor<8xf32> {  
    %f = arith.constant 5.000000e+00 : f32  
    %0 = tensor.insert %f into %t[%idx] : tensor<8xf32>  
    return %0 : tensor<8xf32>  
  }
```



```
func.func @test(%t: tensor<8xf32>, %idx: index)  
  -> tensor<8xf32> {  
    %0 = bufferization.to_memref %t  
      : memref<8xf32, strided<[?]>, offset: ?>>  
    %f = arith.constant 5.000000e+00 : f32  
    %alloc = memref.alloc() : memref<8xf32>  
    memref.copy %0, %alloc : ...  
    memref.store %f, %alloc[%idx] : memref<8xf32>  
    %1 = bufferization.to_tensor %alloc : memref<8xf32>  
    return %1 : tensor<8xf32>  
  }
```


Pass Options (1)

```
mlir-opt %s  
--one-shot-bufferize="bufferize-function-boundaries"
```

Bufferization

-one-shot-bufferize

```
func.func @test(%t: tensor<8xf32>, %idx: index)  
  -> tensor<8xf32> {  
    %f = arith.constant 5.000000e+00 : f32  
    %0 = tensor.insert %f into %t[%idx] : tensor<8xf32>  
    return %0 : tensor<8xf32>  
  }
```



```
func.func @test(  
  %t: memref<8xf32, strided<[?], offset: ?>>,  
  %idx: index)  
  -> memref<8xf32, strided<[?], offset: ?>> {  
    %f = arith.constant 5.000000e+00 : f32  
    memref.store %f, %t[%idx]  
      : memref<8xf32, strided<[?], offset: ?>>  
    return %t : memref<8xf32, strided<[?], offset: ?>>  
  }
```

Pass Options (2)

```
mlir-opt %s  
--one-shot-bufferize=  
"bufferize-function-boundaries  
function-boundary-type-conversion=identity-layout-map"
```

Bufferization

-one-shot-bufferize

```
func.func @test(%t: tensor<8xf32>, %idx: index)  
  -> tensor<8xf32> {  
    %f = arith.constant 5.000000e+00 : f32  
    %0 = tensor.insert %f into %t[%idx] : tensor<8xf32>  
    return %0 : tensor<8xf32>  
  }
```



```
func.func @test(%t: memref<8xf32>, %idx: index)  
  -> memref<8xf32> {  
    %f = arith.constant 5.000000e+00 : f32  
    memref.store %f, %t[%idx] : memref<8xf32>  
    return %t : memref<8xf32>  
  }
```

Pass Options (3)

```
mlir-opt %s
--one-shot-bufferize=
"bufferize-function-boundaries
function-boundary-type-conversion=identity-layout-map"
```

Bufferization

-one-shot-bufferize

```
func.func @test(
  %t: tensor<8xf32> {bufferization.writable = false},
  %idx: index) -> tensor<8xf32> {
  %f = arith.constant 5.0 : f32
  %0 = tensor.insert %f into %t[%idx] : tensor<8xf32>
  return %0 : tensor<8xf32>
}
```



```
func.func @test(%t: memref<8xf32>, %idx: index)
-> memref<8xf32> {
  %f = arith.constant 5.000000e+00 : f32
  %alloc = memref.alloc() : memref<8xf32>
  memref.copy %t, %alloc : ...
  memref.store %f, %alloc[%idx] : memref<8xf32>
  return %alloc : memref<8xf32>
}
```

Example: DPS op

Destination Passing Style (DPS)

- Ops specify the (tensor) **destination** of a computation.
- DestinationStyleOpInterface
- One-Shot Bufferize tries to perform the computation **in-place** in the future buffer of the destination. If not possible: new allocation.
- Can be seen as **memory SSA**.
- Non-DPS ops: bufferize to new allocations.

tied OpOperand/OpResult pair destination

```
%r = tensor.insert %f into %t[%idx]
      : tensor<5xf32>
```

buffer(%r) := buffer(%t) or new alloc



```
%0 = bufferization.to_memref %t
      : memref<5xf32>
memref.store %f, %0[%idx] : memref<5xf32>
%r = bufferization.to_tensor %0
      : memref<5xf32>
```

Example: DPS op

Destination Passing Style (DPS)

- Ops specify the (tensor) **destination** of a computation.
- DestinationStyleOpInterface
- One-Shot Bufferize tries to perform the computation **in-place** in the future buffer of the destination. If not possible: new allocation.
- Can be seen as **memory SSA**.
- Non-DPS ops: bufferize to new allocations.

tied OpOperand/OpResult pair

destination

```
%r = tensor.insert %f into %t[%idx]
      : tensor<5xf32>
```

There is no guarantee that the
result will end up in *buffer(%t)*!



```
%0 = bufferization.to_memref %t
      : memref<5xf32>
%1 = memref.alloc() : memref<5xf32>
memref.copy %0, %1 : ...
memref.store %f, %1[%idx] : memref<5xf32>
%r = bufferization.to_tensor %1
      : memref<5xf32>
```

Destination Passing Style (DPS)

- Ops specify the (tensor) **destination** of a computation.
- DestinationStyleOpInterface
- One-Shot Bufferize tries to perform the computation **in-place** in the future buffer of the destination. If not possible: new allocation.
- Can be seen as **memory SSA**.
- Non-DPS ops: bufferize to new allocations.

Example: Non-DPS op

```
%r = tensor.from_elements %f0, %f1, %f2  
      : tensor<3xf32>
```



```
%m = memref.alloc() : memref<3xf32>  
memref.store %f0, %m[%c0] : memref<3xf32>  
memref.store %f1, %m[%c1] : memref<3xf32>  
memref.store %f2, %m[%c2] : memref<3xf32>  
%r = bufferization.to_tensor %0  
      : memref<3xf32>
```

BufferizableOpInterface

tensor.insert implements
DestinationStyleOpInterface

bufferize() is the only mandatory
interface method for destination
style ops

```
604  /// Bufferization of tensor.insert. Replace with memref.store.
605  ///
606  /// Note: DstBufferizableOpInterfaceExternalModel provides many default method
607  /// implementations for DestinationStyle ops.
608  struct InsertOpInterface
609  : public DstBufferizableOpInterfaceExternalModel<InsertOpInterface,
610  tensor::InsertOp> {
611  LogicalResult bufferize(Operation *op, RewriterBase &rewriter,
612  const BufferizationOptions &options) const {
613  auto insertOp = cast<tensor::InsertOp>(op);
614  FailureOr<Value> destMemref =
615  getBuffer(rewriter, insertOp.getDest(), options);
616  if (failed(destMemref))
617  return failure();
618  rewriter.create<memref::StoreOp>(insertOp.getLoc(), insertOp.getScalar(),
619  *destMemref, insertOp.getIndices());
620  replaceOpWithBufferizedValues(rewriter, op, *destMemref);
621  return success();
622  }
623  };
```

llvm-project/mlir/lib/Dialect/Tensor/Transforms/
BufferizableOpInterfaceImpl.cpp

BufferizableOpInterface

llvm-project/mlir/lib/Dialect/Tensor/Transforms/
BufferizableOpInterfaceImpl.cpp

Not a destination style op because
operand and result are not tied

Returns a brand new buffer

Memory read side effect on
bufferized source tensor

No aliasing relationship between
bufferized operand and result

Predict the bufferized result type.
Needed for compute the type of loop
iter_args etc.

```
706 // Bufferization of tensor.pad. Replace with bufferization.alloc_tensor +
707 // linalg.map + insert_slice.
708 // For best performance, vectorize before bufferization (better performance in
709 // case of padding with a constant).
710 struct PadOpInterface
711     : public BufferizableOpInterface::ExternalModel<PadOpInterface,
712           tensor::PadOp> {
713     bool bufferizesToAllocation(Operation *op, Value value) const { return true; }
714     bool bufferizesToMemoryRead(Operation *op, OpOperand &opOperand,
715           const AnalysisState &state) const {
716         return true;
717     }
718     bool bufferizesToMemoryWrite(Operation *op, OpOperand &opOperand,
719           const AnalysisState &state) const {
720         return false;
721     }
722     AliasingValueList getAliasingValues(Operation *op, OpOperand &opOperand,
723           const AnalysisState &state) const {
724         return {};
725     }
726     FailureOr<BaseMemRefType>
727     getBufferType(Operation *op, Value value, const BufferizationOptions &options,
728           SmallVector<Value> &invocationStack) const {
729         // Infer memory space from the source tensor.
730         auto padOp = cast<tensor::PadOp>(op);
731         auto maybeSrcBufferType = bufferization::getBufferType(
732           padOp.getSource(), options, invocationStack);
733         if (failed(maybeSrcBufferType))
734             return failure();
735         MemRefLayoutAttrInterface layout;
```


Common Pitfalls

- BufferizableOpInterface external models not registered: ops are not getting bufferized
- Function boundary bufferization not enabled: function bbArgs are read-only
- getBufferType() not implemented: mismatch between loop inits and iter_arg types
- getBufferType() / bufferize() ignores the memory space of MemRef types
- Input IR has to_tensor without restrict
- Assumptions that a computation materializes in a certain buffer without making it explicit (e.g., tensor ops may canonicalize/fold away)
- Loop op yields value that is not equivalent to corresponding iter_arg: inefficient due to current implementation details

```
140 // Register all external models.
141 affine::registerValueBoundsOpInterfaceExternalModels(registry);
142 arith::registerBufferDeallocationOpInterfaceExternalModels(registry);
143 arith::registerBufferizableOpInterfaceExternalModels(registry);
144 arith::registerValueBoundsOpInterfaceExternalModels(registry);
145 bufferization::func_ext::registerBufferizableOpInterfaceExternalModels(
146     registry);
147 builtin::registerCastOpInterfaceExternalModels(registry);
148 cf::registerBufferizableOpInterfaceExternalModels(registry);
149 cf::registerBufferDeallocationOpInterfaceExternalModels(registry);
150 gpu::registerBufferDeallocationOpInterfaceExternalModels(registry);
151 linalg::registerBufferizableOpInterfaceExternalModels(registry);
152 linalg::registerSubsetInsertionOpInterfaceExternalModels(registry);
153 linalg::registerTilingInterfaceExternalModels(registry);
154 linalg::registerValueBoundsOpInterfaceExternalModels(registry);
155 memref::registerAllocationOpInterfaceExternalModels(registry);
156 memref::registerBufferizableOpInterfaceExternalModels(registry);
mlir/include/mlir/InitAllDialects.h
```

bufferize-function-boundaries

bufferization


.materialize_in_destination

```
%a, %b = scf.for ... iter_args(%arg0 = %c, %arg1 = %d) ... {
  scf.yield %arg1, %arg0 : tensor<5xf32>, tensor<5xf32>
}
```

Read-after-Write Conflict Detection

- **Definition:** A tensor SSA value that defines the contents of a tensor.
- **Conflicting Write:** A use (OpOperand) that scrambles/overwrites a part of the definition.
- **Read:** A use (OpOperand) that expects to read a part of the definition.

in this order



```
// Definition
%0 = tensor.from_elements %f0, %f1, %f2
    : tensor<3xf32>

// Conflicting Write
%1 = tensor.insert %f3 into %0[%idx]
    : tensor<3xf32>

// Read
%r = tensor.extract %0[%idx2]
    : tensor<3xf32>
```

Read-after-Write Conflict Detection

- **Definition:** A tensor SSA value that defines the contents of a tensor.
- **Conflicting Write:** A use (OpOperand) that scrambles/overwrites a part of the definition.
- **Read:** A use (OpOperand) that expects to read a part of the definition.

in this order

```
%m = memref.alloc() : memref<3xf32>
memref.store %f0, %m[%c0] : memref<3xf32>
memref.store %f1, %m[%c1] : memref<3xf32>
memref.store %f2, %m[%c2] : memref<3xf32>
%r = bufferization.to_tensor %0
      : memref<3xf32>
```

```
%m2 = memref.alloc() : memref<3xf32>
memref.copy %m, %m2 : memref<3xf32>
memref.store %f3, %m2[%idx] : memref<3xf32>
```

```
%r = memref.load %m[%idx2] : memref<3xf32>
```

Read-after-Write Conflict Detection

- **Definition:** A tensor SSA value that defines the definition.
- **Conflicting write:** A use (OpOperand) that scrambles/overwrites a part of the definition.
- **Read:** A use (OpOperand) that expects to read a part of the definition.

Also takes into account aliasing

RaW can be avoided by inserting copy in one of two places

```
// Definition
%0 = tensor.from_elements %f0, %f1, %f2
    : tensor<3xf32>
%0_alias = tensor.extract_slice %0[1][2][1]
           : tensor<3xf32> to tensor<2xf32>

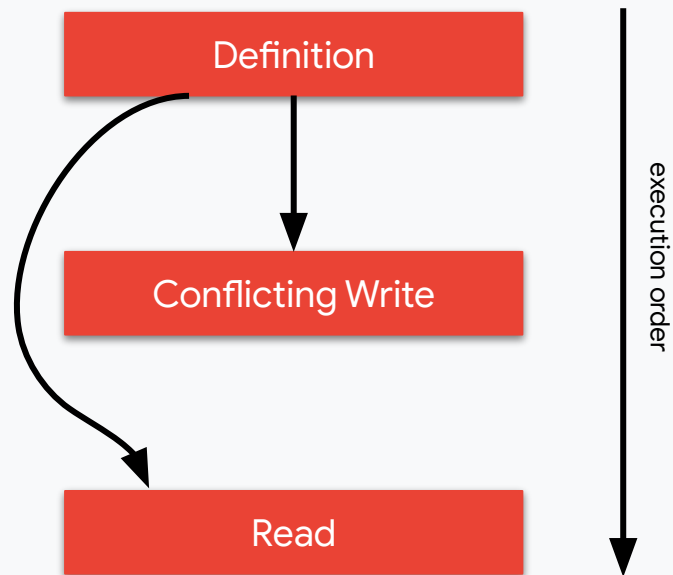
// Conflicting Write
%1 = tensor.insert %f3 into %0[%idx]
    : tensor<3xf32>

// Read
%r = tensor.extract %0_alias[%idx2]
    : tensor<3xf32>
```

Conflict Detection Rules

Definition → Conflicting Write → Read
according to op dominance

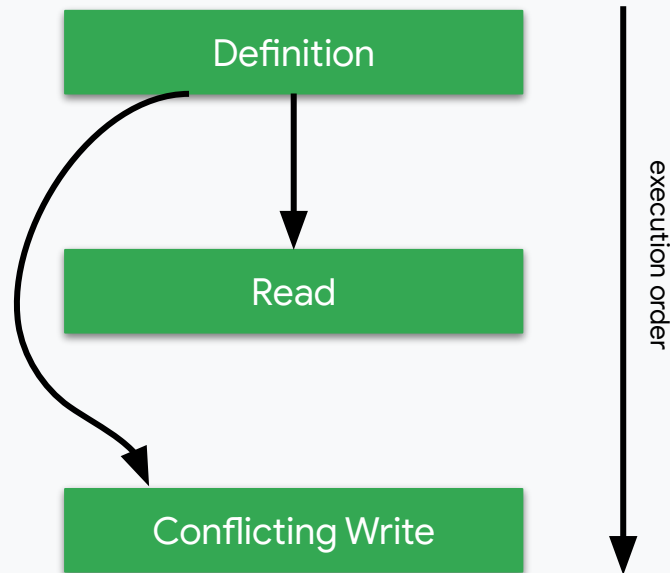
CONFLICT



Conflict Detection Rules

No conflict if Read happens before
Conflicting Write

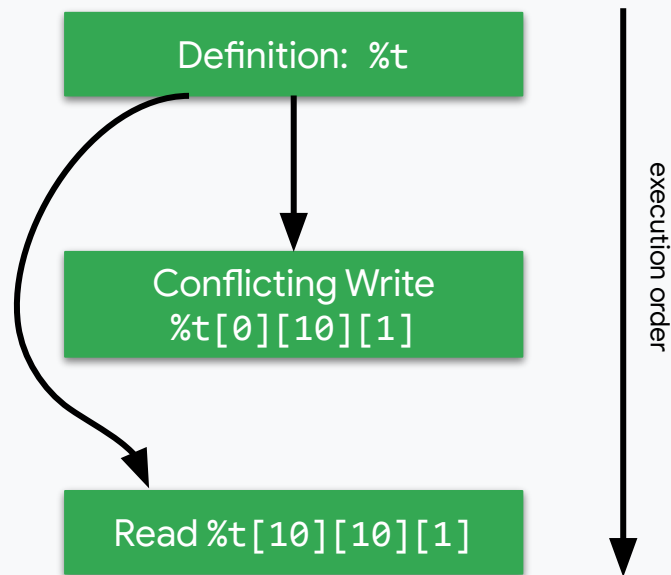
NOT A CONFLICT



Conflict Detection Rules

No conflict if Read and Conflicting Write operate on disjoint subsets (as per `SubsetInsertionOpInterface`)

NOT A CONFLICT

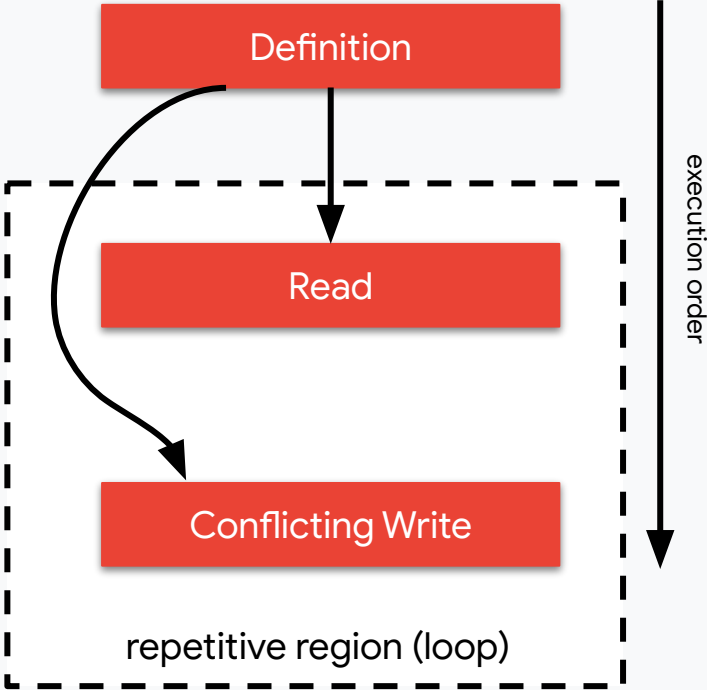
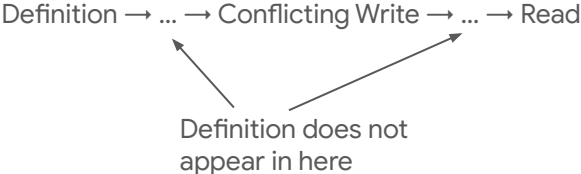


Important for efficient bufferization of `tensor.extract_slice/insert_slice` pairs.

Google Research

Conflict Detection Rules

There is a conflict if there this is a possible execution order (as per RegionBranchOpInterface):



DEMO: Debugging Spurious Copies: Mini Example

- `test_analysis_only`: Annotes the IR with the results of the analysis.
- `print_conflicts`: Print additional information about RaW conflicts.
- `dump_alias_sets`: Print alias sets.

<https://gist.github.com/matthias-springer/81748fe1e530974dd5ff6b3ad57e3eeb>

<http://tiny.cc/3wxbvz>

DEMO: Debugging Spurious Copies: Matmul, Tiled

- `test_analysis_only`: Annotes the IR with the results of the analysis.
- `print_conflicts`: Print additional information about RaW conflicts.
- `dump_alias_sets`: Print alias sets.

<https://gist.github.com/matthias-springer/372162baa30e79c49180bb3ace216995>

<https://gist.github.com/matthias-springer/b664feb23be0159f72726025923bb9ca>

<http://tiny.cc/3wxbvz>

02

Empty Tensor Elimination

tensor.empty

- A tensor with unspecified contents
- Generated by various transformation, e.g., tosa-to-linalg

```
%r = tensor.empty() : tensor<5xf32>
```



```
%0 = memref.alloc() : memref<5xf32>  
%r = bufferization.to_tensor %0  
      : memref<5xf32>
```

Example: tosa-to-linalg

Bufferizes to allocation

Destination passing style

Preprocessing

`rewrite_in_destination_passing_style`

```
%r = tosa.add %A, %B  
      : (tensor<5xf32>, tensor<5xf32>) -> tensor<5xf32>
```



```
%0 = tensor.empty() : tensor<5xf32>  
  
%r = linalg.generic ...  
      ins(%A, %B : tensor<5xf32>, tensor<5xf32>)  
      outs(%0 : tensor<5xf32>) {  
^bb0(%in: f32, %in_0: f32, %out: f32):  
      %1 = arith.addf %in, %in_0 : f32  
      linalg.yield %1 : f32  
} -> tensor<5xf32>
```

Example: tosa-to-linalg

bufferizes to memref.copy

Preprocessing

rewrite_in_destination_passing_style

could be a function “out” argument

```
%r = tosa.add %A, %B
      : (tensor<5xf32>, tensor<5xf32>) -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      restrict writable : (tensor<5xf32>, memref<5xf32>) -> ()
```



```
%0 = tensor.empty() : tensor<5xf32>

%r = linalg.generic ...
      ins(%A, %B : tensor<5xf32>, tensor<5xf32>)
      outs(%0 : tensor<5xf32>) {
^bb0(%in: f32, %in_0: f32, %out: f32):
      %1 = arith.addf %in, %in_0 : f32
      linalg.yield %1 : f32
} -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      restrict writable : (tensor<5xf32>, memref<5xf32>) -> ()
```

Example: tosa-to-linalg

Empty tensor elimination: Instead of computing something in a temporary buffer (`tensor.empty`) and then copying the result into another buffer, perform the computation directly in that buffer.

Preprocessing

`rewrite_in_destination_passing_style`

```
%r = tosa.add %A, %B
      : (tensor<5xf32>, tensor<5xf32>) -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      restrict writable : (tensor<5xf32>, memref<5xf32>) -> ()
```



```
%0 = tensor.empty() : tensor<5xf32>
%r = linalg.generic ...
      ins(%A, %B : tensor<5xf32>, tensor<5xf32>)
      outs(%0 : tensor<5xf32>) {
^bb0(%in: f32, %in_0: f32, %out: f32):
      %1 = arith.addf %in, %in_0 : f32
      linalg.yield %1 : f32
} -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      restrict writable : (tensor<5xf32>, memref<5xf32>) -> ()
```

Example: tosa-to-linalg

Empty tensor elimination: Instead of computing something in a temporary buffer (`tensor.empty()`) and then copying the result into another buffer, perform the computation directly in that buffer.

still bufferizes to `memref.copy`,
but from `%buffer` to `%buffer`

Preprocessing

`rewrite_in_destination_passing_style`

`-eliminate-empty-tensors`

```
%r = tosa.add %A, %B
      : (tensor<5xf32>, tensor<5xf32>) -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      restrict writable : (tensor<5xf32>, memref<5xf32>) -> ()
```



```
%0 = tensor.empty() : tensor<5xf32>
%0 = bufferization.to_tensor %buffer restrict writable : tensor<5xf32>
%r = linalg.generic ...
      ins(%A, %B : tensor<5xf32>, tensor<5xf32>)
      outs(%0 : tensor<5xf32>) {
^bb0(%in: f32, %in_0: f32, %out: f32):
  %1 = arith.addf %in, %in_0 : f32
  linalg.yield %1 : f32
} -> tensor<5xf32>
bufferization.materialize_in_destination %r in %buffer
      writable : (tensor<5xf32>, memref<5xf32>) -> ()
```


DEMO: Empty Tensor Elimination

Materialize in buffer destination:

<https://gist.github.com/matthias-springer/b3f40d1667c977c29a76cc7a469cc1a0>

Materialize in tensor destination:

<https://gist.github.com/matthias-springer/e531580242d27f14e0a239e0b6fe80ae>

Rewrite other ops in destination passing style:

<https://gist.github.com/matthias-springer/35e54346cb6374bf417e7224259dc77e>

<http://tiny.cc/3wxbvz>

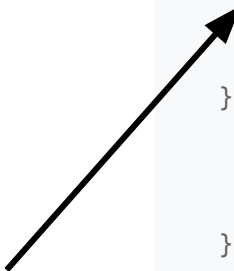
02

Buffer Deallocation

Buffer Deallocation

Should run at the very end: certain optimization passes (e.g., hoisting) do not support dealloc ops

conditional alloc without dealloc: **memory leak**



-buffer-deallocation-pipeline

Deallocation

~~-buffer-deallocation~~

```
func.func @dealloc_test(  
  %c: i1, %m: memref<5xf32>, %idx: index, %f: f32) -> f32 {  
  %0 = scf.if %c -> memref<5xf32> {  
    %1 = memref.alloc() : memref<5xf32>  
    linalg.fill ins(%f: f32) outs(%1 : memref<5xf32>)  
    scf.yield %1 : memref<5xf32>  
  } else {  
  
    scf.yield %m : memref<5xf32>  
  }  
  
  %r = memref.load %0[%idx] : memref<5xf32>  
  
  return %r : f32  
}
```

Old Buffer Deallocation

buffer-deallocation should not be used anymore:

- has bugs (memory leaks, etc.)
- has assumptions that are incompatible with One-Shot Bufferize
- inserts additional copies (expensive)

unconditional alloc allows
for unconditional dealloc:
inefficient!

-buffer-deallocation-pipeline

Deallocation

~~-buffer-deallocation~~

```
// RUN: mlir-opt -buffer-deallocation -canonicalize

func.func @dealloc_test(
  %c: i1, %m: memref<5xf32>, %idx: index, %f: f32) -> f32 {
  %0 = scf.if %c -> memref<5xf32> {
    %1 = memref.alloc() : memref<5xf32>
    linalg.fill ins(%f: f32) outs(%1 : memref<5xf32>)
    scf.yield %1 : memref<5xf32>
  } else {
    %2 = memref.alloc() : memref<5xf32>
    memref.copy %m, %2 : memref<5xf32> to memref<5xf32>
    scf.yield %2 : memref<5xf32>
  }

  %r = memref.load %0[%idx] : memref<5xf32>
  memref.dealloc %0 : memref<5xf32>
  return %r : f32
}
```

Ownership- Based Buffer Deallocation

conditional alloc and
conditional dealloc



-buffer-deallocation-pipeline

Deallocation

~~-buffer-deallocation~~

```
// RUN: mlir-opt -buffer-deallocation-pipeline
```

```
func.func @dealloc_test(  
  %c: i1, %m: memref<5xf32>, %idx: index, %f: f32) -> f32 {  
  %0 = scf.if %c -> memref<5xf32> {  
    %1 = memref.alloc() : memref<5xf32>  
    linalg.fill ins(%f: f32) outs(%1 : memref<5xf32>)  
    scf.yield %1 : memref<5xf32>  
  } else {  
    scf.yield %m : memref<5xf32>  
  }  
  %r = memref.load %0[%idx] : memref<5xf32>  
  
  %base_buffer, %offset, %sizes, %strides =  
    memref.extract_strided_metadata %0  
    : memref<5xf32> -> memref<f32>, index, index, index  
  scf.if %c {  
    memref.dealloc %base_buffer : memref<f32>  
  }  
  return %r : f32  
}
```

Google Research

Ownership-Based Buffer Deallocation

-buffer-deallocation-pipeline

Deallocation

~~-buffer-deallocation~~

Ops with region must implement
RegionBranchOpInterface

Terminators must implement
BranchOpInterface or
RegionBranchTerminatorOpInterface

(Or implement BufferDeallocationOpInterface.)

```
// RUN: mlir-opt -buffer-deallocation-pipeline

func.func @dealloc_test(
  %c: i1, %m: memref<5xf32>, %idx: index, %f: f32) -> f32 {
  %0 = scf.if %c -> memref<5xf32> {
    %1 = memref.alloc() : memref<5xf32>
    linalg.fill ins(%f: f32) outs(%1 : memref<5xf32>)
    scf.yield %1 : memref<5xf32>
  } else {
    scf.yield %m : memref<5xf32>
  }
  %r = memref.load %0[%idx] : memref<5xf32>

  %base_buffer, %offset, %sizes, %strides =
    memref.extract_strided_metadata %0
    : memref<5xf32> -> memref<f32>, index, index, index
  scf.if %c {
    memref.dealloc %base_buffer : memref<f32>
  }
  return %r : f32
}
```

Google Research

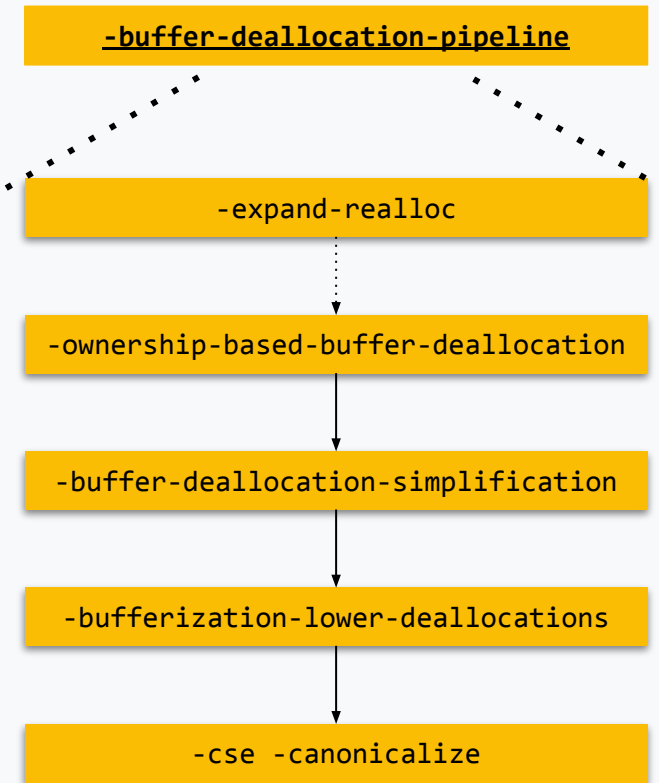
Feature Overview

	Old Deallocation	Ownership-based
Buffer writes must not dominate all reads	✗	✓
Unstructured CF loops	✗	✓
Function Boundary ABI	✗	✓
Existing deallocation ops allowed	✓ (some limitations)	✗
Refinable & extensible (via interface)	✗	✓

Disadvantage: input IR must adhere to it

Ownership- Based Buffer Deallocation

Deallocation



Ownership- Based Buffer Deallocation

replace memref.realloc
with conditional alloc +
copy, no dealloc (leak)

Deallocation

-buffer-deallocation-pipeline

-expand-realloc

-ownership-based-buffer-deallocation

-buffer-deallocation-simplification

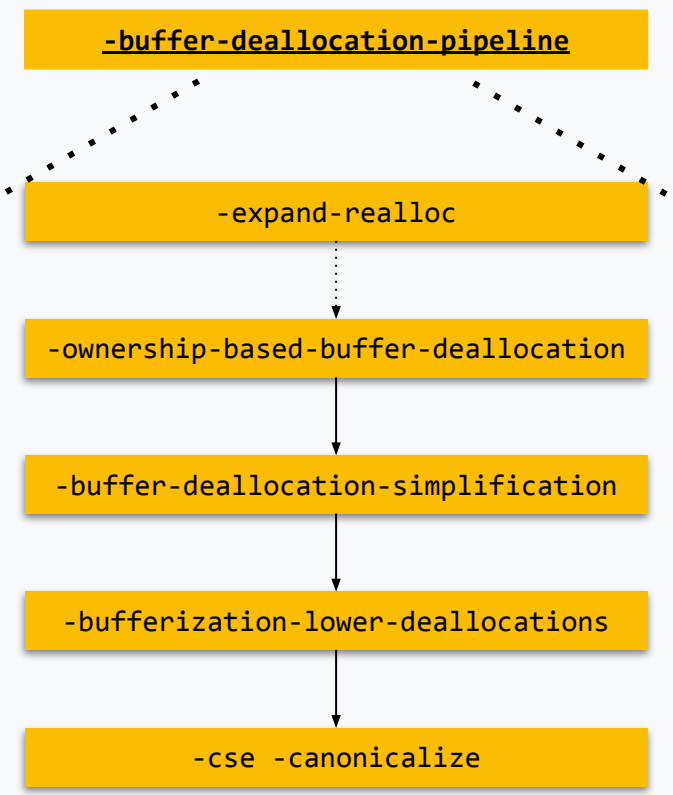
-bufferization-lower-deallocations

-cse -canonicalize

Ownership-Based Buffer Deallocation

insert
bufferization.dealloc
at the end of basic blocks

Deallocation



Ownership- Based Buffer Deallocation

simplify/canonicalize
bufferization.dealloc

Deallocation

-buffer-deallocation-pipeline

-expand-realloc

-ownership-based-buffer-deallocation

-buffer-deallocation-simplification

-bufferization-lower-deallocations

-cse -canonicalize

Ownership-Based Buffer Deallocation

Deallocation

-buffer-deallocation-pipeline

-expand-realloc

-ownership-based-buffer-deallocation

-buffer-deallocation-simplification

-bufferization-lower-deallocations

-cse -canonicalize

lower
bufferization.dealloc
to memref.dealloc

New `bufferization.dealloc` operation

- New deallocation pass is based on the concept of **buffer ownership**.
 - Ownership is a **property of the base allocation** (e.g., result of `memref.alloc`). (There is no separate ownership for aliases.)
 - The owner of an allocation is always a **basic block**.
- `bufferization.dealloc` models a **conditional deallocation** and is always inserted at the end of a basic block. Whether we need to deallocate a buffer depends on:
 - **Ownership**: Does the block own the buffer, i.e., is it responsible for deallocating a given buffer?
 - **Aliasing Information**: Is the buffer (or an alias thereof) used at a later point?
 - **Liveness**: are there uses in a successor block?
- Capturing this in a dedicated operation allows for
 - A simpler deallocation pass
 - Subsequent optimizations
 - Specialized lowerings

New `bufferization.dealloc` operation

dealloc list: these are all buffers that may have to be deallocated

```
bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)
```

```
  if (%own1, %own2)
```

```
    retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

ownership indicators: represented as `i1`

retain list: these buffers are still needed

same # of operands

arbitrary # of MemRefs,
Memref types do not have
to match

Intuitively: Deallocate `%m1` if `%own1` and `%m2` if `%own2`. But only if that would not invalidate `%r1`, `%r2` or `%r3`.

New `bufferization.dealloc` operation

Should `%m1` be deallocated?

```
bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)
```

```
if (%own1, %own2)
```

1. Is `%own1` 'true'?

```
retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

2. Does `%m1` originate from the same allocation as any of the retained values? **No use-after-free!**

Intuitively: Deallocate `%m1` if `%own1` and `%m2` if `%own2`. But only if that would not invalidate `%r1`, `%r2` or `%r3`.

New `bufferization.dealloc` operation

1. Does it originate from the same allocation as `%m1` and was `%m1` deallocated?
No double deallocs!

Should `%m2` be deallocated?

```
bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)
```

```
if (%own1, %own2)
```

2. Is `%own2` 'true'?

```
retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

2. Does `%m1` originate from the same allocation as any of the retained values? **No use-after-free!**

Intuitively: Deallocate `%m1` if `%own1` and `%m2` if `%own2`. But only if that would not invalidate `%r1`, `%r2` or `%r3`.

New `bufferization.dealloc` operation

Ownership of %r1, %r2, and %r3

By construction, must contain all buffers that the enclosing basic block may own.

```
%0:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
                                     if (%own1, %own2)  
                                     retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

Same # of results and retained operands

Intuitively: Given that the `dealloc` list contains all buffers that the basic block may own, return the ownership of each retained value.

Note: Ownership is a **per-allocation property**. If a block owns a `memref`, it also owns all of its aliases (including the base allocation).

New `bufferization.dealloc` operation

How to compute `%O#0`?

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
        retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

0. Start with `%O#0 := %false`

1. `%O#0 := %false` or `(isSameBuffer(%m1, %r1) and %own1)`

Evaluates to `%true` (by assumption)

Note: Results can be conveniently used as the additional forwarded operands.

New `bufferization.dealloc` operation

How to compute `%O#0`?

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
        retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

0. Start with `%O#0 := %false`

1. `%O#0 := %false` or `(%true and %own1)`

Note: Results can be conveniently used as the additional forwarded operands.

New `bufferization.dealloc` operation

How to compute `%O#0`?

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
        retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

0. Start with `%O#0 := %false`

1. `%O#0 := %own1`

Note: Results can be conveniently used as the additional forwarded operands.

New `bufferization.dealloc` operation

How to compute `%O#0`?

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
      retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

2. Does `%r1` originate from the same allocation as `%m2`?
Let's assume that's NOT the case.

0. Start with `%O#0 := %false`

1. `%O#0 := %own1`

2. `%O#0 := %own1` or `(isSameBuffer(%m2, %r1) and %own2)`

Evaluates to `%false` (by assumption)

Note: Results can be conveniently used as the additional forwarded operands.

New `bufferization.dealloc` operation

How to compute `%O#0`?

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
      retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

2. Does `%r1` originate from the same allocation as `%m2`?
Let's assume that's NOT the case.

0. Start with `%O#0 := %false`

1. `%O#0 := %own1`

2. `%O#0 := %own1` or (`%false` and `%own2`)

Note: Results can be conveniently used as the additional forwarded operands.

New `bufferization.dealloc` operation

How to compute `%O#O`?

```
%O:3 = bufferization.dealloc (%m1, %m2 : memref<?xf32>, memref<3xf32>)  
      if (%own1, %own2)  
      retain (%r1, %r2, %r3 : memref<5xf64>, memref<?xf32>, memref<f32>)
```

1. Does `%r1` originate from the same allocation as `%m1`?
Let's assume that's the case.

2. Does `%r1` originate from the same allocation as `%m2`?
Let's assume that's NOT the case.

0. Start with `%O#O := %false`

1. `%O#O := %own1`

2. `%O#O := %own1`

Note: Results can be conveniently used as the additional forwarded operands.

bufferization.dealloc Lowering

- One **default lowering**
 - Runtime in $O(|deallocs|^2 + |deallocs| * |retained|)$
 - Space in $O(|deallocs| + |retained|)$
 - Inserts private func . func library function → **ModuleOp** pass
- **Optimized lowerings** for frequent special cases
 - One MemRef to dealloc, none retained:
dealloc (%m : ...) if (%cond)
 - One MemRef to dealloc, any number of retains:
dealloc (%m : ...) if (%cond) retain (...)
 - **Can be run on functions** (if default lowering is not needed)

DEMO: Buffer Deallocation Step-by-Step

Ownership-based Buffer Deallocation Pass:

<https://gist.github.com/maerhart/e8d29fb3d483aa98ab511aefcfb7fd9c>

Simplifying `bufferization.dealloc` operations:

<https://gist.github.com/maerhart/c608792add1ca6bce012d9734e2ee4d3>

Lowering `bufferization.dealloc` operations:

<https://gist.github.com/maerhart/532fa2f6801f49663dfb3762af190130>

<http://tiny.cc/3wxbvz>

- BufferizableOpInterface
- BufferDeallocationOpInterface
- DestinationStyleOpInterface
- SubsetInsertionOpInterface
- bufferization.dealloc
- bufferization.materialize in destination
- bufferization.to memref
- bufferization.to tensor
- tensor.empty
- your_dialect.alloc
- your_dialect.dealloc
- Conditional Dealloc vs. Buffer Cloning
- Function Boundary ABI
- Parallel Region
- Read-after-Write (RaW) Conflict Detection
- Repetitive Region / Cycling Basic Block Graph
- Memory Space
- MemRef Layout Map
- Transform Dialect Integration
- Unstructured Control Flow

<http://tiny.cc/3wxbvz>

Preprocessing

Bufferization

Buffer-Level
Optimizations

Deallocation

rewrite_in_destination_passing_style

-eliminate-empty-tensors

-one-shot-bufferize

-buffer-hoisting

-buffer-loop-hoisting

-buffer-results-to-out-params

-drop-equivalent-buffer-results

-promote-buffers-to-stack

-buffer-deallocation-pipeline

~~-buffer-deallocation~~

Backup Slides

Pass Options (4)

```
mlir-opt %s
--one-shot-bufferize=
  "bufferize-function-boundaries
  function-boundary-type-conversion=identity-layout-map"
--drop-equivalent-buffer-results
```

Bufferization

`-one-shot-bufferize`

↓
Buffer-Level
Optimizations

`-drop-equivalent-buffer-results`

```
func.func @test(%t: tensor<8xf32>, %idx: index)
-> tensor<8xf32> {
  %f = arith.constant 5.000000e+00 : f32
  %0 = tensor.insert %f into %t[%idx] : tensor<8xf32>
  return %0 : tensor<8xf32>
}
```



```
func.func @test(%t: memref<8xf32>, %idx: index) {
  %f = arith.constant 5.000000e+00 : f32
  memref.store %f, %t[%idx] : memref<8xf32>
  return
}
```

Destination Passing Style (DPS)

- Ops specify the (tensor) **destination** of a computation.
- DestinationStyleOpInterface
- One-Shot Bufferize tries to perform the computation **in-place** in the future buffer of the destination. If not possible: new allocation.
- Can be seen as **memory SSA**.
- Non-DPS ops: bufferize to new allocations.

Example: DPS op

tied OpOperand/OpResult pair destination

```
%r = tensor.insert_slice %t into %t2[1][5][1]  
: tensor<5xf32> into tensor<10xf32>
```



```
%0 = bufferization.to_memref %t  
: memref<5xf32>  
%1 = bufferization.to_memref %t2  
: memref<10xf32>  
%1_view = memref.subview %1[1][5][1]  
: memref<10xf32> to memref<5xf32, ...>  
memref.copy %0, %m1 : ...  
%r = bufferization.to_tensor %0  
: memref<5xf32>
```

BufferizableOpInterface

llvm-project/mlir/lib/Dialect/Tensor/Transforms/
BufferizableOpInterfaceImpl.cpp

Not a destination style op because
operand and result type do not match

No memory read/write side effects
on any operand

The only tensor operand will alias with the
result of the op (if bufferized in-place)

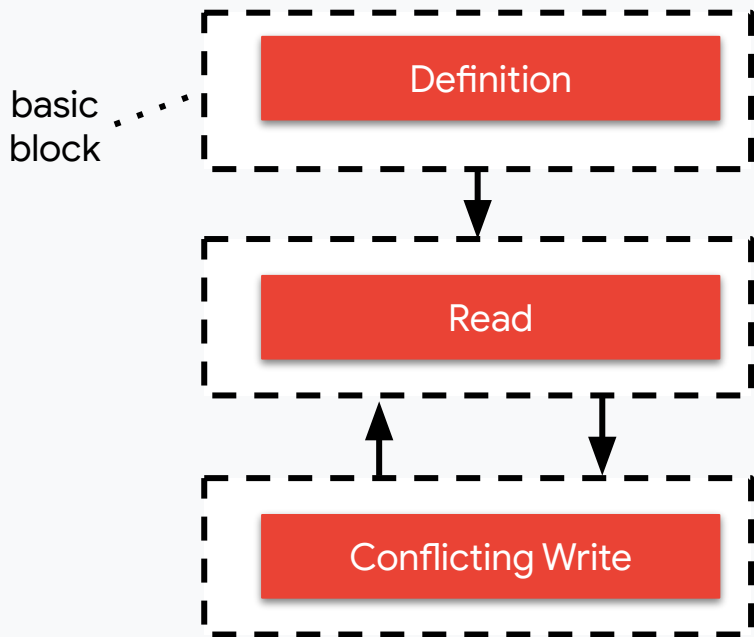
Predict the bufferized result type.
Needed for compute the type of loop
iter_args etc.

```
286  /// Bufferization of tensor.expand_shape. Replace with memref.expand_shape.
287  struct ExpandShapeOpInterface
288  : public BufferizableOpInterface::ExternalModel<ExpandShapeOpInterface,
289  tensor::ExpandShapeOp> {
290  bool bufferizesToMemoryRead(Operation *op, OpOperand &opOperand,
291  const AnalysisState &state) const {
292  return false;
293  }
294
295  bool bufferizesToMemoryWrite(Operation *op, OpOperand &opOperand,
296  const AnalysisState &state) const {
297  return false;
298  }
299
300  AliasingValueList getAliasingValues(Operation *op, OpOperand &opOperand,
301  const AnalysisState &state) const {
302  return {{op->getOpResult(0), BufferRelation::Equivalent}};
303  }
304
305  FailureOr<BaseMemRefType>
306  getBufferType(Operation *op, Value value, const BufferizationOptions &options,
307  SmallVector<Value> &invocationStack) const {
308  auto expandShapeOp = cast<tensor::ExpandShapeOp>(op);
309  auto maybeSrcBufferType = bufferization::getBufferType(
310  expandShapeOp.getSrc(), options, invocationStack);
311  if (failed(maybeSrcBufferType))
312  return failure();
313  auto srcBufferType = llvm::cast<MemRefType>(*maybeSrcBufferType);
314  auto maybeResultType = memref::ExpandShapeOp::computeExpandedType(
315  srcBufferType, expandShapeOp.getResultType().getShape(),
316  expandShapeOp.getReassociationIndices());
317  if (failed(maybeResultType))
318  return failure();
```

Conflict Detection Rules

There is a conflict if there this is the following two paths in the basic block graph:

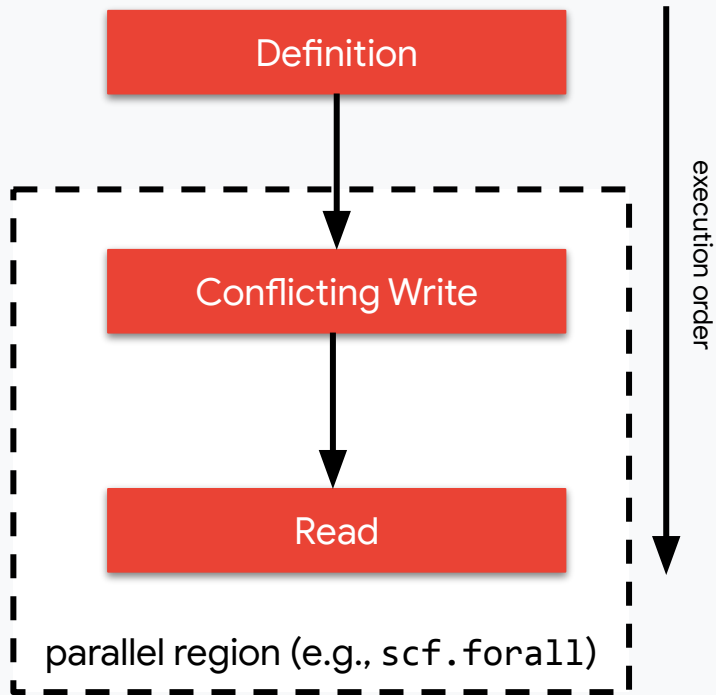
1. $\text{block}(\text{Read}) \rightarrow \text{block}(\text{Conflicting Write})$, without passing through block (Definition)
2. $\text{block}(\text{Conflicting Write}) \rightarrow \text{block}(\text{Read})$, without passing through block (Definition)



Conflict Detection Rules

There is a conflict if the Conflicting Write is a parallel region different from the Definition: buffer must be privatized

CONFLICT



Conflict Detection Rules

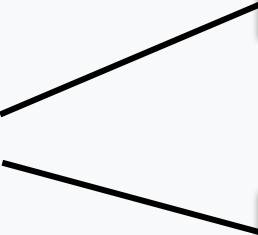
```
BufferizableOpInterface  
::isNotConflicting() returns true
```

specify custom rules for your ops

Definition

Conflicting Write

Read



DEMO: Debugging Spurious Copies: Element-wise, Tiled

- `test_analysis_only`: Annotes the IR with the results of the analysis.
- `print_conflicts`: Print additional information about RaW conflicts.
- `dump_alias_sets`: Print alias sets.

<https://gist.github.com/matthias-springer/50f5cc3a7b8ad85054c19b96770042dd>

<https://gist.github.com/matthias-springer/5cc5b29c1bd727a272a78d71f1e6e19a>

<http://tiny.cc/3wxbvz>

BufferDeallocationOpInterface

Error: All operations with attached regions need to implement RegionBranchOpInterface!

Op with region that doesn't implement FunctionOpInterface or RegionBranchOpInterface not supported by default implementation

```
28 func.func @reduce(%buffer: memref<100xf32>) {
29   %init = arith.constant 0.0 : f32
30   %c0 = arith.constant 0 : index
31   %c1 = arith.constant 1 : index
32   scf.parallel (%iv = (%c0) to (%c1) step (%c1) init (%init) -> f32 {
33     %elem_to_reduce = memref.load %buffer[%iv] : memref<100xf32>
34     scf.reduce(%elem_to_reduce) : f32 {
35       ^bb0(%lhs : f32, %rhs: f32):
36         %alloc = memref.alloc() : memref<2xf32>
37         memref.store %lhs, %alloc [%c0] : memref<2xf32>
38         memref.store %rhs, %alloc [%c1] : memref<2xf32>
39         %0 = memref.load %alloc[%c0] : memref<2xf32>
40         %1 = memref.load %alloc[%c1] : memref<2xf32>
41         %res = arith.addf %0, %1 : f32
42         scf.reduce.return %res : f32
43     }
44   }
45   func.return
46 }
```

Apply the default implementation, i.e., insert `bufferization.dealloc` right before the reduce

```
59 struct ReduceReturnOpInterface
60   : public BufferDeallocationOpInterface::ExternalModel<
61     ReduceReturnOpInterface, scf::ReduceReturnOp> {
62   FailureOr<Operation*> process(Operation *op, DeallocationState &state,
63     const DeallocationOptions &options) const {
64     auto reduceReturnOp = cast<scf::ReduceReturnOp>(op);
65     if (isa<BaseMemRefType>(reduceReturnOp.getOperand().getType()))
66       return op->emitError("only supported when operand is not a MemRef");
67     SmallVector<Value> updatedOperandOwnership;
68     return deallocation_impl::insertDeallocOpForReturnLike(
69       state, op, {}, updatedOperandOwnership);
70 }
```

Mixed Allocations

IR may contain different memref allocation operations with corresponding deallocation ops.

E.g.:

- memref.alloc + memref.dealloc
- my_dialect.alloc + my_dialect.dealloc

```
/// Options for BufferDeallocationOpInterface-based buffer deallocation.
struct DeallocationOptions {
    using DetectionFn = std::function<bool(Operation *)>;

    /// Given an allocation side-effect on the passed operation, determine whether
    /// this allocation operation is of relevance (i.e., should assign ownership
    /// to the allocated value). If it is determined to not be relevant,
    /// ownership will be set to 'false', i.e., it will be leaked. This is useful
    /// to support deallocation of multiple different kinds of allocation ops.
    DetectionFn isRelevantAllocOp = [](Operation *op) {
        return isa<memref::MemRefDialect, bufferization::BufferizationDialect>(
            op->getDialect());
    };

    /// Given a free side-effect on the passed operation, determine whether this
    /// deallocation operation is of relevance (i.e., should be removed if the
    /// `removeExistingDeallocations` option is enabled or otherwise an error
    /// should be emitted because existing deallocation operations are not
    /// supported without that flag). If it is determined to not be relevant,
    /// the operation will be ignored. This is useful to support deallocation of
    /// multiple different kinds of allocation ops where deallocations for some of
    /// them are already present in the IR.
    DetectionFn isRelevantDeallocOp = [](Operation *op) {
        return isa<memref::MemRefDialect, bufferization::BufferizationDialect>(
            op->getDialect());
    };
};
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {  
  %0 = memref.alloc() : memref<f64>  
  %1 = my_dialect.alloc() : memref<f64>  
  %2 = arith.select %cond, %0, %1  
  
  return %2  
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {  
  %0 = memref.alloc() : memref<f64>  
  %1 = my_dialect.alloc() : memref<f64>  
  %2 = arith.select %cond, %0, %1  
  bufferization.dealloc (%0, %1)  
    if (%true, %false)  
    retain (%2)  
  
  return %2  
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {  
    %0 = memref.alloc() : memref<f64>  
    %1 = my_dialect.alloc() : memref<f64>  
    %2 = arith.select %cond, %0, %1  
    bufferization.dealloc (%0)  
        if (%true)  
        retain (%2)  
  
    return %2  
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., `memref.alloc`) and %false for all other alloc op types (e.g., `my_dialect.alloc`)

```
func private @example() -> memref<f64> {  
    %0 = memref.alloc() : memref<f64>  
    %1 = my_dialect.alloc() : memref<f64>  
    %2 = arith.select %cond, %0, %1  
    bufferization.dealloc (%0)  
                                if (%not_cond)  
  
    return %2  
}
```


Mixed Allocations

Specify deallocation operation to be inserted in C++ pass options.

```
/// Options for the LowerDeallocation pass and rewrite patterns.  
struct LowerDeallocationOptions {  
    /// Given a MemRef value, build the operation(s) necessary to properly  
    /// deallocate the value.  
    std::function<void(OpBuilder &, Location, Value)> buildDeallocOp =  
        [](OpBuilder &builder, Location loc, Value memref) {  
            builder.create<memref::DeallocOp>(loc, memref);  
        };  
};
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {  
  %0 = memref.alloc() : memref<f64>  
  %1 = my_dialect.alloc() : memref<f64>  
  %2 = arith.select %cond, %0, %1  
  scf.if %not_cond {  
    memref.dealloc %0  
  }  
  
  return %2  
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {
  %0 = memref.alloc() : memref<f64>
  %1 = my_dialect.alloc() : memref<f64>
  %2 = arith.select %cond, %0, %1
  scf.if %not_cond {
    memref.dealloc %0
  }
  bufferization.dealloc (%0, %1)
                        if (%false, %true)
                        retain (%2)

  return %2
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {
  %0 = memref.alloc() : memref<f64>
  %1 = my_dialect.alloc() : memref<f64>
  %2 = arith.select %cond, %0, %1
  scf.if %not_cond {
    memref.dealloc %0
  }
  bufferization.dealloc (%1)
                        if (%true)
                        retain (%2)

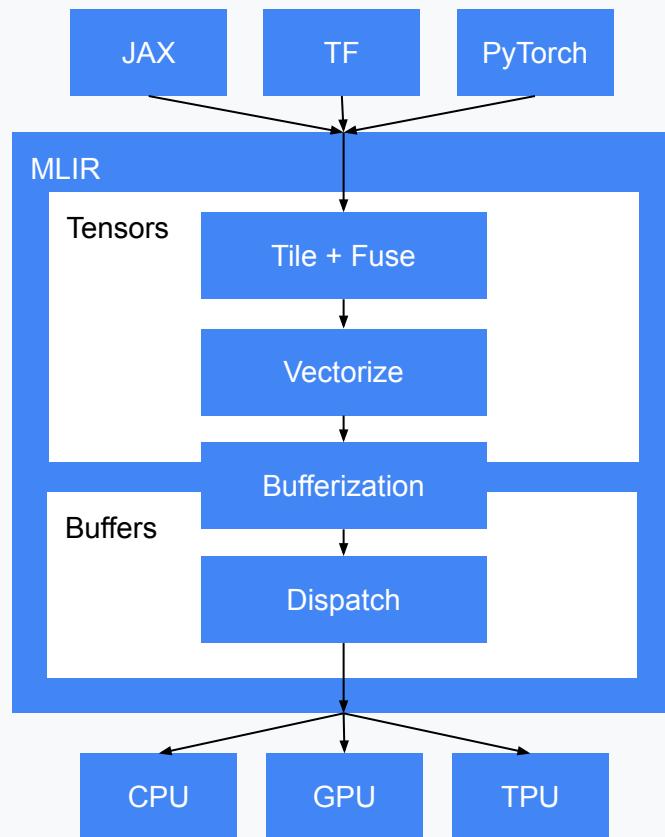
  return %2
}
```

Mixed Allocations

- Run pipeline once for each kind of alloc op
- Assign ownership of %true for specified alloc op type (e.g., memref.alloc) and %false for all other alloc op types (e.g., my_dialect.alloc)

```
func private @example() -> memref<f64> {
  %0 = memref.alloc() : memref<f64>
  %1 = my_dialect.alloc() : memref<f64>
  %2 = arith.select %cond, %0, %1
  scf.if %not_cond {
    memref.dealloc %0
  }
  scf.if %cond {
    my_dialect.dealloc %1
  }
  return %2
}
```

Compilation Pipeline



The Old Buffer Deallocation Pass

- No (documented) function boundary ABI
- Leaks memory
- Does not support unstructured control flow loops
- All buffer writes have to dominate all buffer reads (not guaranteed by One-Shot Bufferize)
- Potentially inserts a lot of copies

From CloneOp documentation:

Valid implementations of this operation may alias the input and output views or create an actual copy. Mutating the source or result of the clone operation after the clone operation thus leads to undefined behavior.

```
func.func @callee() -> memref<1xf64> {
  %1 = memref.alloc() : memref<1xf64>
  return %1 : memref<1xf64>
}
func.func @caller() {
  %0:2 = call @callee() : () -> memref<1xf64>
  // memory is leaked here
  return
}
```

```
func.func @many_clones(%cond : i1) -> memref<4xf32> {
  %1 = memref.alloc() : memref<4xf32>
  %2 = scf.if %cond -> (memref<4xf32>) {
    %3 = bufferization.clone %1 : memref<4xf32>
    scf.yield %3 : memref<4xf32>
  } else {
    %3 = memref.alloc() : memref<4xf32>
    scf.yield %3 : memref<4xf32>
  }
  memref.dealloc %2 : memref<4xf32>
  return %2 : memref<4xf32>
}
```

memref.realloc Lowering

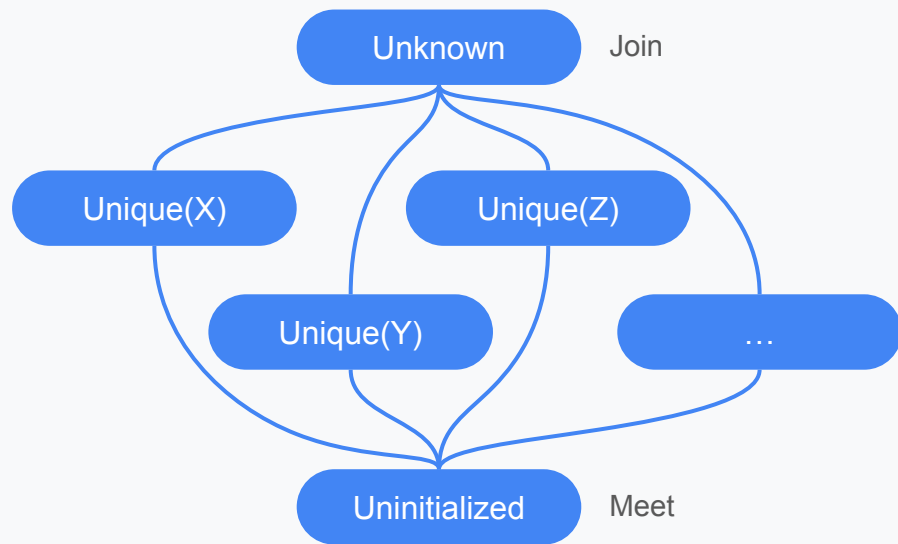
```
%realloc = memref.realloc %alloc (%size)  
: memref<?xf32> to memref<?xf32>
```



```
%c0 = arith.constant 0 : index  
%dim = memref.dim %alloc, %c0 : memref<?xf32>  
%is_old_smaller = arith.cmpi ult, %dim, %arg1  
%realloc = scf.if %is_old_smaller -> (memref<?xf32>) {  
  %new_alloc = memref.alloc(%size) : memref<?xf32>  
  %subview = memref.subview %new_alloc[0][%dim][1]  
  memref.copy %alloc, %subview  
  memref.dealloc %alloc  
  scf.yield %alloc_0 : memref<?xf32>  
} else {  
  %reinterpret_cast = memref.reinterpret_cast %alloc to  
    offset: [0], sizes: [%size], strides: [1]  
  scf.yield %reinterpret_cast : memref<?xf32>  
}
```


Ownership Lattice

- Each MemRef-typed SSA value is assigned an Ownership value
- *Unique* state can materialize as SSA value
- Pass inserts conditional deallocations
 - Old pass made copies instead
 - Decide at runtime whether deallocation should be performed

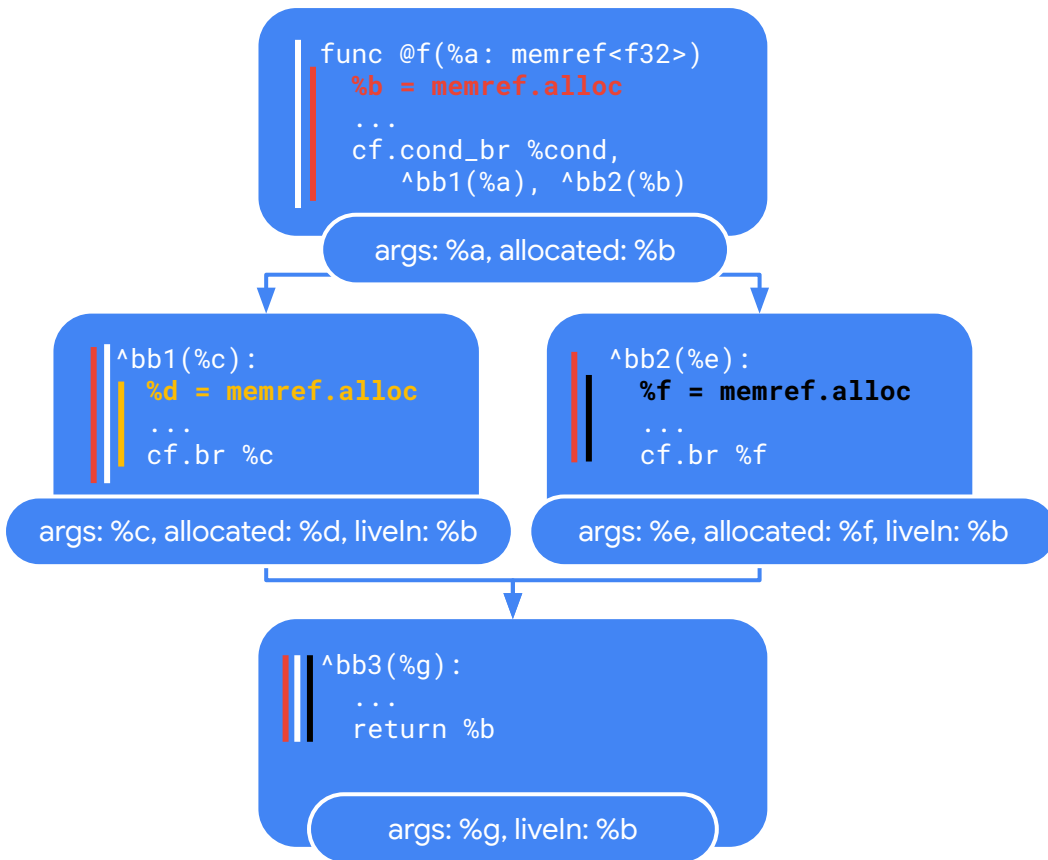


X, Y, Z are distinct SSA values of i1 type

Ownership-based Deallocation Pass

1. Collect MemRef values that potentially need to be deallocated per block
 - a. Uses Liveness Analysis

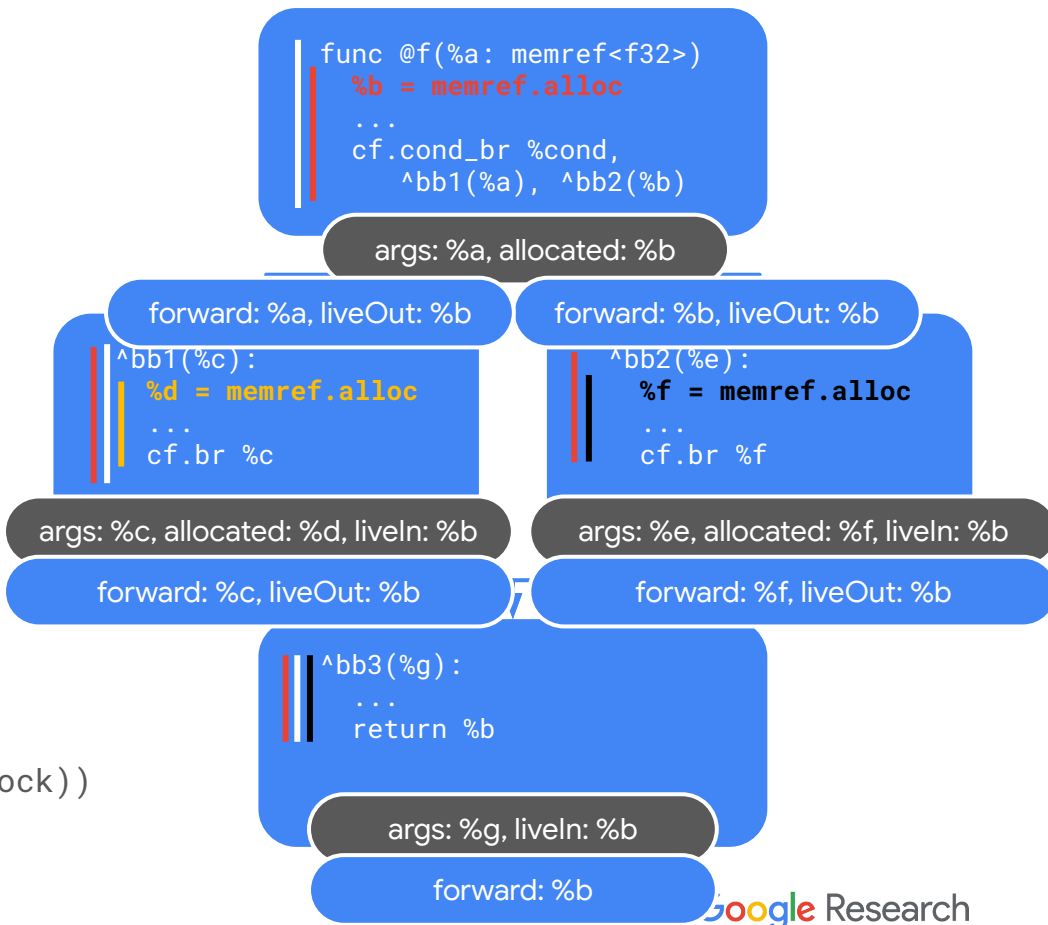
$\text{liveIn}(\text{block})$
 \cup
 $\text{allocated}(\text{block})$
 \cup
 $\text{arguments}(\text{block})$



Ownership-based Deallocation Pass

1. Collect MemRef values that potentially need to be deallocated per block
2. Collect MemRef values to retain per block
 - a. Uses Liveness Analysis

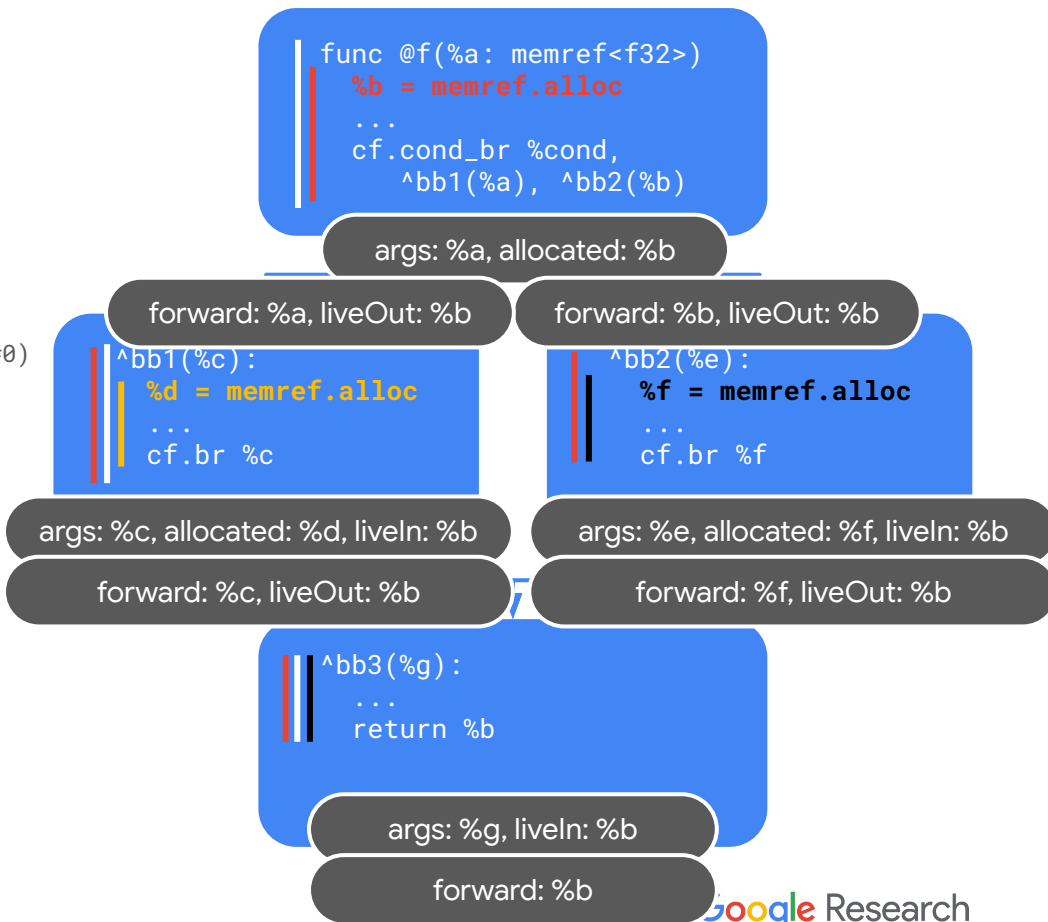
$$\text{forwardedOperands} \cup (\text{liveOut}(\text{fromBlock}) \cap \text{liveIn}(\text{toBlock}))$$



```

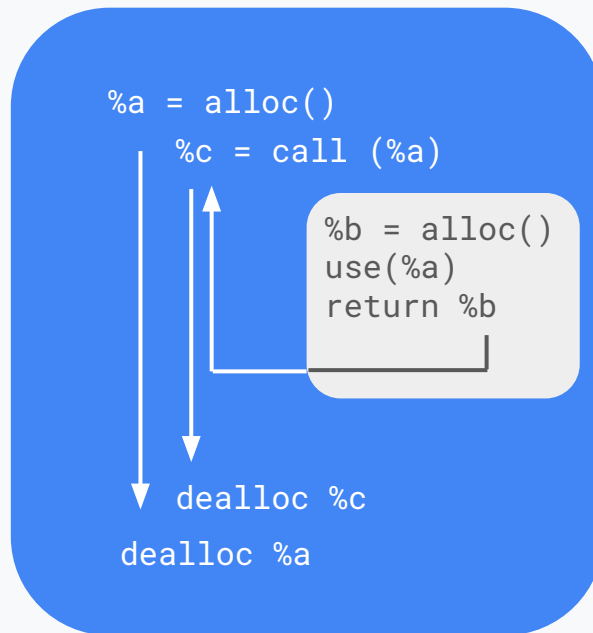
func @f(%a: memref<f32>) -> memref<f32> {
  %b = memref.alloc
  ...
  %b_then = arith.andi %cond, %true
  %not_cond = arith.xori %cond, %true
  %b_else = arith.andi %not_cond, %true
  %0:2 = bufferization.dealloc (%a, %b)
      if (%false, %b_then)
        retain (%a, %b)
  %1:2 = bufferization.dealloc (%a, %b)
      if (%false, %b_else)
        retain (%b, %b)
  %b_own = arith.select %cond, %0#1, %1#1
  cf.cond_br %cond, ^bb1(%a, %0#0), ^bb2(%b, %1#0)
^bb1(%c, %c_own):
  %d = memref.alloc
  ...
  %2:2 = bufferization.dealloc (%c, %d, %b)
      if (%c_own, %true, %b_own)
        retain (%c, %b)
  cf.br ^bb3(%c, %2#0)
^bb2(%e, %e_own):
  %f = memref.alloc
  ...
  %3:2 = bufferization.dealloc (%e, %f, %b)
      if (%e_own, %true, %b_own)
        retain (%f, %b)
  cf.br ^bb3(%f, %3#0)
^bb3(%g, %g_own):
  ...
  bufferization.dealloc (%g, %b)
      if (%g_own, %b_own)
        retain (%b)
  return %b
}

```



Public Function Boundary ABI

- Ownership is never acquired by callee
- Ownership of returned MemRef is always passed to caller
- Returned MemRefs must not alias with function arguments (it would then not be possible to return the MemRef with ownership)
- Returned MemRefs must not alias each other



Extending the ABI

- Private Functions: add ownership indicators as additional return values
- *Future Work*: Allow users to statically specify aliasing and ownership ABI of a function (e.g., as attributes)

```
func private @dyn_own(%cond: i1) -> memref<f64> {  
  %0 = memref.alloc() : memref<f64>  
  %1 = memref.get_global @global : memref<f64>  
  %2 = arith.select %cond, %0, %1 : memref<f64>  
  // instead of cloning here and deallocating %0,  
  // return an additional i1 result  
  return %2 : memref<f64>  
}
```

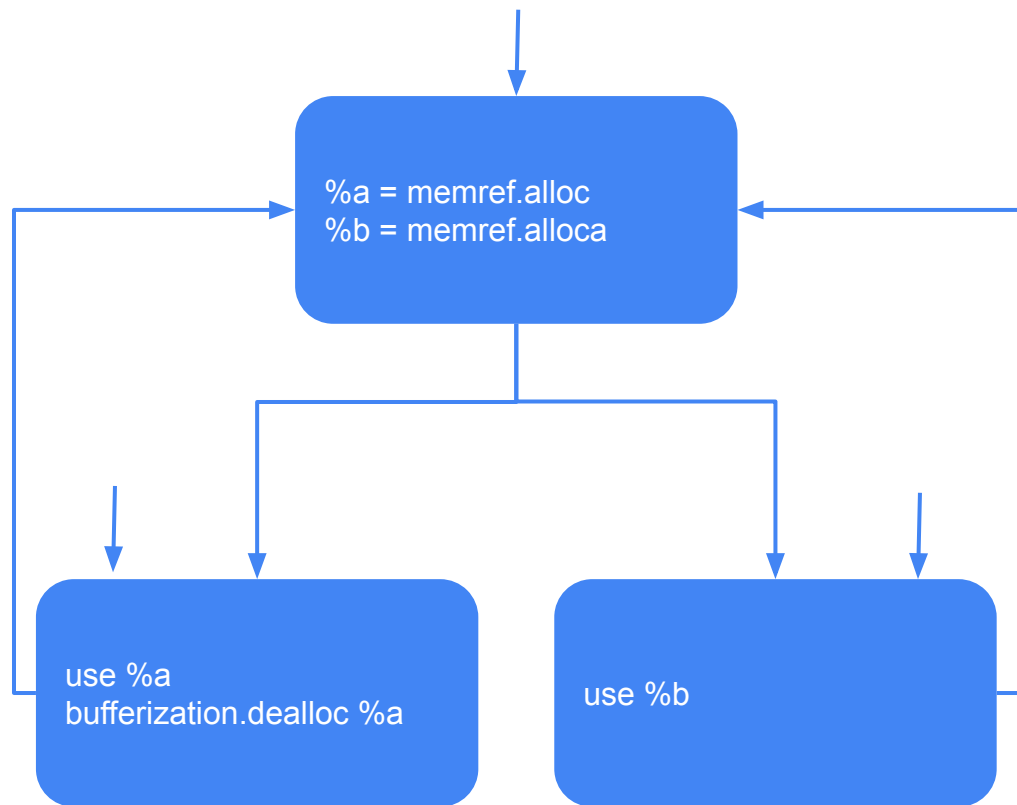
```
func @ret_ownership() -> memref<f64> {  
  %0 = memref.alloc() : memref<f64>  
  return %0 : memref<f64>  
}
```

```
func @ret_no_ownership() -> memref<f64> {  
  %0 = memref.get_global @global : memref<f64>  
  // would need to clone here  
  return %0 : memref<f64>  
}
```

Some Theoretical Thoughts

Could we implement the deallocation pass without the BufferDeallocationOpInterface?

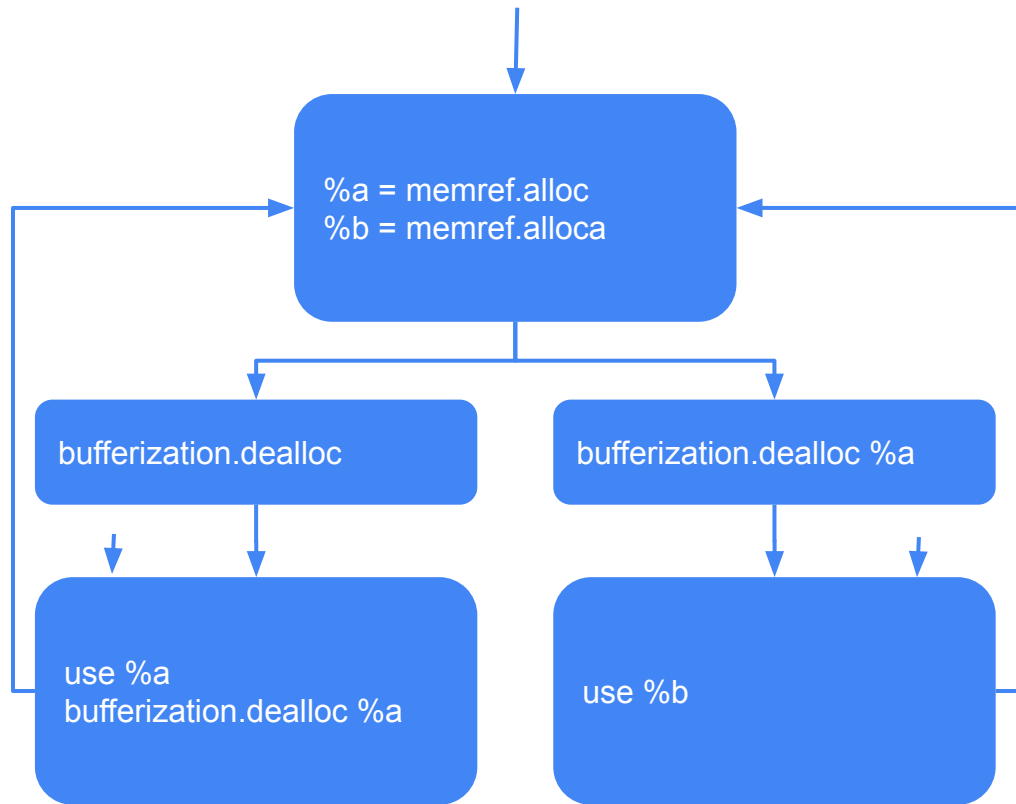
Yes, correctness can be maintained, but performance will suffer



Some Theoretical Thoughts

Could we implement the deallocation pass without the BufferDeallocationOpInterface?

Yes, correctness can be maintained, but performance will suffer



Some Theoretical Thoughts

Could we get a unique ownership value for `arith.select` without implementing the interface?

```
%s = arith.select %cond, %a1, %a2
%s_ptr = memref.extract_alloc_pointer_as_index %s
%a1_ptr = memref.extract_alloc_pointer_as_index %a1
%a2_ptr = memref.extract_alloc_pointer_as_index %a2
ownership(%s) =
Switch %s_ptr
Case %a1_ptr : ownership(%a1)
Case %a2_ptr : ownership(%a2)
```

Buffer Origin Analysis

- LocalAliasAnalysis
 - entirely different kind of analysis, using it would be incorrect
- BufferViewFlowAnalysis
 - Caching mechanism makes it hard to use with a rewriter
 - No *MUST* information, only *MAY*, and *NONE*
- What we actually need is a “is same base allocation” analysis, not an “aliasing” analysis

