Comprehensive Bufferization

Avoid Buffer Copies at all Costs

Summary

Comprehensive Bufferization is a bufferization pass for HPC-style codegen in Linalg. It operates based on heuristics and tries to bufferize in-place whenever possible.

Authors: Matthias Springer, Nicolas Vasilache, Tobias Gysi
{springerm, ntv, gysit}@google.com

Status: Draft
Created: 2021-09-27
Updated: 2022-01-10

Summary

Bufferization is the process of allocating/assigning `memref` buffers to `tensor` values and rewriting tensor-based operations to memref-based operations. Bufferization is necessary to make IR with tensor types executable. Comprehensive Bufferization is a new tensor bufferization pass designed for IR in destination-passing style, and with aggressive in-place bufferization.

Comprehensive Bufferization is:

- **Monolithic**: A single MLIR pass does the entire work, whereas the existing bufferization in MLIR is split across multiple passes residing in different dialects.
- A **whole-function at a time analysis**. In-place bufferization decisions are made by **analyzing SSA use-def chains on tensors**. In contrast, other bufferization solutions introduce copies at first, then try to remove them again based on a `memref` alias analysis.
- Extensible via an op **interface**: All ops that implement this interface can be bufferized.
- **Greedy**: Operations are analyzed one-by-one and it is decided on the spot whether a tensor OpOperand must be copied or not. **Heuristics** determine the order of analysis.
- **2-Pass**: Bufferization is internally broken down into 2 steps: First, *analyze* the entire IR and make bufferization decisions. Then, *bufferize* (rewrite) the IR. The analysis has access to exact SSA use-def information. It incrementally builds alias and equivalence sets and does not rely on a posteriori-alias analysis from preallocated memory.
- To reduce complexity, designed to be **run after other transformations**. Many transformations are easier in tensor land. E.g., tile/fuse/... on tensors first, then bufferize the remaining IR.

From an architecture perspective, Comprehensive Bufferize consists of `BufferizableOpInterface` (and its implementations) and an analysis of tensor SSA values that decides if a buffer can be used directly or must be copied. The `bufferize` method of the op interface inspects analysis results and rewrites tensor ops into memref ops. Comprehensive Bufferize is **modular**: The analysis could be replaced with a different one. Comprehensive Bufferize could even run without an analysis, in which case it behaves like the existing core bufferization (copy every buffer that is written into).

Comprehensive Bufferization is currently used when lowering Linalg programs. The existing MLIR "core" bufferization passes conservatively copy a buffer every time it is modified and rely on a copy removal pass to remove unneeded allocations and copies again (not currently implemented). In its current state, core bufferization is unacceptable for HPC codegen; even more so for sparse tensor code, where introducing a buffer copy can change the asymptotic complexity of a program.

Notation and Definitions

Comprehensive Bufferization looks for ops with tensor operands and/or results that have a memref-based equivalent. All other ops are ignored. A tensor result is typically (but not necessarily) *tied* to one (or multiple tensor operands), meaning that the two can share the same buffer in the absence of *conflicts*.

As an example, consider `vector.transfer_write` with a tensor operand and a tensor result.

The memref-based operation is also `vector.transfer_write` and the op's second operand is *tied* to its first (and only) result. I.e., `%0` and `%A` can share the same memref buffer in the absence of conflicts such as a subsequent read of `%A`.

Tensor-typed OpOperands have two important properties: They may or may not *bufferize to a memory read* or *bufferize to a memory write*. If the content of an OpOperand is read by an op, it is said to bufferize to a memory read. This is the case for most ops; however, some ops such as `linalg.generic` or `linalg.fill` can have *output only* operands who's contents are never read. An OpOperand is said to bufferize to a memory write if the op writes to its buffer after bufferization. E.g., a TransferReadOp's tensor operand bufferizes to a memory read but not to a memory write. A TransferWriteOp tensor operand (e.g., `%A` in the example) bufferizes to both a memory read and a memory write.

Phase 1: Analysis

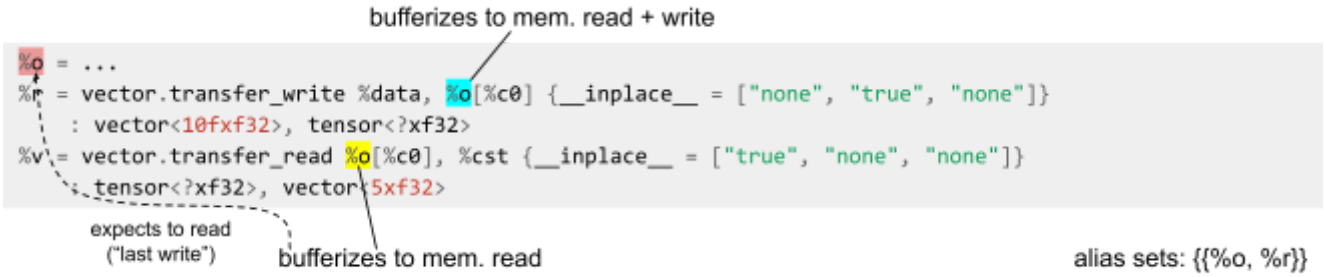
During the analysis phase, Comprehensive Bufferization walks over all ops (inside a given function) with a tensor operand. For each such operand, it decides whether the OpOperand's memory buffer can be used directly or if a new copy of the buffer should be used. In the former case, the OpOperand is said to *bufferize in-place*, indicated by an `__inplace__ = ["true"]` attribute[1]. In the latter case, the OpOperand is said to *bufferize out-of-place*.

The analysis also keeps track of tensor-typed SSA values that alias[2]. In the beginning, every SSA value has its own alias set. When bufferizing an OpOperand `o` in-place, if `o` has a tied OpResult `r`, the alias sets of `o` and `r` are merged.

An OpOperand `o` can bufferize in-place if this decision does not introduce a RaW (read after write) conflict. A RaW conflict emerges if, due to in-place bufferization decisions, an OpOperand reads data different from the one that would be expected when following its reverse SSA use-def chain (i.e., the place where the data of OpOperand is written).

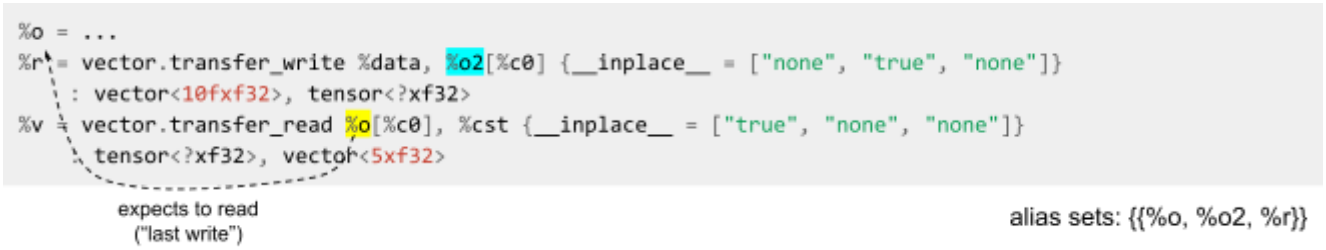
[1] The array attribute has one boolean value for each OpOperand. The value of non-tensor operands is "none".
[2] `llvm::EquivalenceClasses<Value>` is used as a data structure.

Example



In the above example, the in-place bufferization of the TransferWriteOp's %o operand constitutes a conflict because %o is written-to in-place[3] and the old value of %o is read afterwards via %o.

However, conflicts are not always that easy to spot. If the TransferWriteOp were to write in-place to another tensor %o2, there would still be a conflict if %o and %o2 alias (as per the alias sets):



One important point here is that the write/creation of %o happens before the conflicting write to %o2, so the conflicting write overwrites/modifies the original write.

RaW Conflict Detection Algorithm

The conflict detection algorithm iterates over all tensor OpOperands and decides on the spot whether the operand should bufferize in-place or not. For each OpOperand, the algorithm simulates in-place bufferization and checks if a conflict can be found under this assumption. If so, the OpOperand bufferizes out-of-place, otherwise in-place.

```
def bufferizable_in_place_analysis(op_operand):
  op_result = tied_result(op_operand) # Note: There may not be an op_result.
  uses_read = ...
  uses_inplace_write = ...

  for u_read in uses_read:
    last_write = find_last_write(u_read)
    for u_conflicting_write in uses_inplace_write:
      if is_conflict(u_read, last_write, u_conflicting_write):
        set_inplace_attr(op_operand, false)
        return
  set_inplace_attr(op_operand, true)
  alias_sets.merge(op_result, op_operand)
```

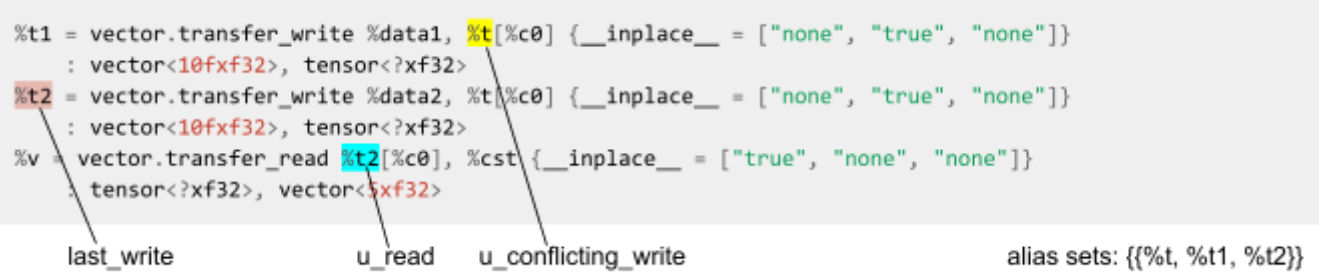
[3] Whenever we say "write to a tensor" or "read from a tensor", we mean "write to the corresponding buffer of the tensor after bufferization" or "read from the corresponding buffer of the tensor after bufferization".

`uses_read` is the set of all aliasing reads of `op_operand` and `op_result` (if any). I.e., take a look at all uses of all aliases of `op_operand` and `op_result`. If such a use bufferizes to a memory read, it is included in `uses_read`. If it does not bufferize to a memory read, but another use that is reachable by following this use's SSA use-def chain (always following the tied OpResult) bufferizes to a memory read, that other use is also included. Intuitively, we would like to capture all possible memory reads of the buffer of `op_operand` and `op_result` (which is simulated to be the same buffer).

`uses_inplace_write` is the set of all aliasing writes of `op_operand` and `op_result`. Similar to `uses_read`, a use is included if it *bufferizes to a memory write* and bufferizes in-place as per the `__inplace__` attribute. If no in-place bufferization was made for an OpOperand yet, it is assumed to bufferize out-of-place for the moment. In addition, if `op_operand` itself bufferizes to a memory write, it is also included in `uses_inplace_write`.

`last_write` is the SSA Value (block argument or OpResult) of the last write when tracing back `u_read` with SSA use-def chains[4]. This is usually `u_read`, but some ops such as `tensor.extract_slice` introduce aliases without writing data. More about that later.

A tuple (`last_write`, `u_conflicting_write`, `u_read`) constitutes a conflict, unless the analysis can prove it does not. Intuitively, there is no conflict if `u_conflicting_write` happens before `last_write` or after `u_read` as per op dominance[5].

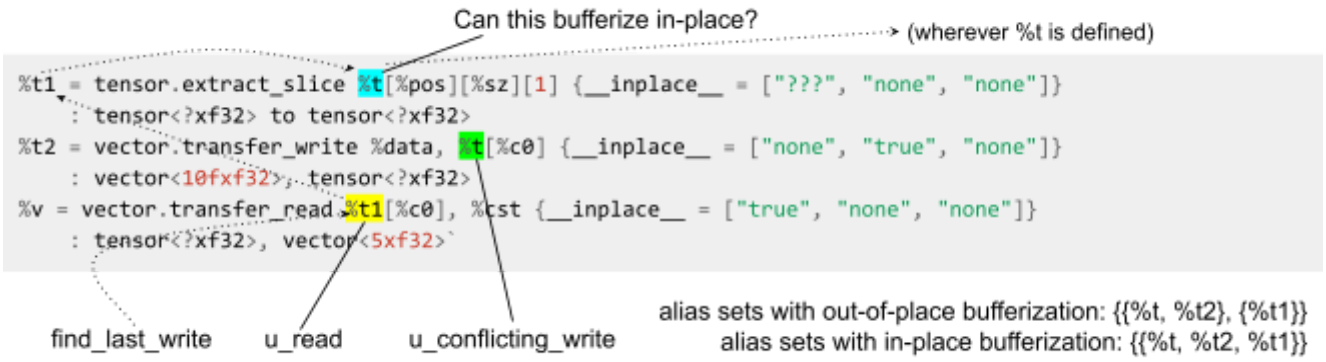


In the above example, TransferReadOp's read of `%t2`, the last write (`%t2`) and the potentially conflicting write of `%t` are not a conflict, because the conflicting write happens before the last write.
Additional rules are necessary to handle certain ops (such as `scf::IfOp`) correctly. These are mere extensions to the described RaW conflict detection algorithm and do not change it in a major way. These rules are described in the appendix.

Analysis of `tensor.extract_slice`

`tensor.extract_slice` is an interesting op: It neither bufferizes to a memory read nor to a memory write, but it introduces an alias if it bufferizes in-place (as any other op that has a tied OpOperand/OpResult pair). Due to the greedy nature of the analysis, the source OpOperand of a `tensor.extract_slice` op (same as any other op) may have to bufferize out-of-place in the context of existing bufferization decisions. In this case, out-of-place bufferization is not triggered by a new memory write but by extending an alias set. E.g., consider the following example, where all ops apart from the `tensor.extract_slice` have already been analyzed.

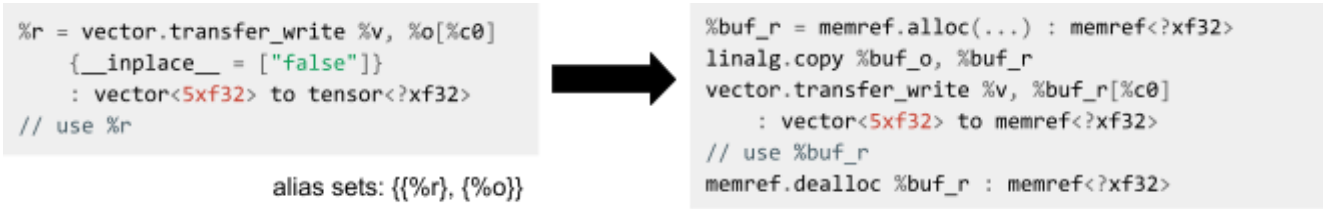
[4] To be precise, always select the tied OpOperand when tracing back. Stop when reaching an OpOperand that bufferizes to a memory write or when there is no tied OpOperand for a Value.
[5] If `u_conflicting_write` is the same op as `u_read` but a different OpOperand, there is conflict. If they are the same OpOperand, no conflict. If `u_conflicting_write`'s tied OpResult is `last_write`, there is no conflict.



Before the analysis of the `tensor.extract_slice`, the values `%t` and `%t2` are already known to alias due to the in-place bufferization of the `vector.transfer_write`. If the `tensor.extract_slice`'s `%t` were to bufferize in-place, the alias set would grow to {`%t`, `%t1`, `%t2`}, which would introduce a conflict with the `vector.transfer_read`'s `%t1` as `u_read`, `%t` as `last_write` (the `tensor.extract_slice` op is skipped in `find_last_write` because it does not write) and the `vector.transfer_write`'s `%t` as `u_conflicting_write`. Therefore, the `tensor.extract_slice` op has to bufferize out-of-place.

Phase 2: Bufferization

Based on the results from the analysis, ops are one-by-one rewritten to `memref`-based variants. Whenever an OpOperand was decided to bufferize out-of-place, a new buffer is allocated, data from the original buffer is copied over and the OpResult tensor is mapped to the new buffer.



Order of Analysis (Heuristic)

The order in which ops are analyzed can affect the quality of the bufferization decisions. The current implementation follows a greedy analysis: Comprehensive Bufferizations analyzes the ops in a function bottom-up. This strategy is based on the observation that blocks in Linalg programs often end with a `tensor.insert_slice`. It is crucial that these ops bufferize in-place for two reasons. First, copying the `dest` buffer is inefficient and it is likely better to copy some other (potentially smaller) buffer instead (if needed). Second, returning the result of an out-of-place bufferized op is not supported because new allocations may not escape function boundaries (*destination-passing style*).

Note that different orders of analysis may result in better bufferization decisions. We plan to give users a way to control the order of analysis in the future. Such a feature is not currently exposed to users, but Comprehensive Bufferize's implementation is already set up to handle arbitrary op traversal orders correctly.

Destination-Passing Style

Comprehensive Bufferize was designed for IR that is in destination-passing style. This imposes certain restrictions regarding what IR can be bufferized, but it also simplifies bufferization significantly.

In essence, ops that are in destination-passing style have an output tensor[6]. The future buffer of this output tensor is the buffer that will be written into (if an OpOperand bufferizes to a memory write). E.g., `tensor.insert`'s destination tensor is an output tensor. In the absence of a conflict, the buffer of the tensor will be written to. In the presence of a conflict, a copy will be made and the copy will be written to.

Certain MLIR ops are not in destination-passing style. E.g, the matrix multiplication ops from the TOSA dialect is not in destination-passing style.

```
%r = "tosa.matmul"(%a, %b) {__inplace__ = ["true", "true"]}
    : (tensor<?x?xf32>, tensor<?x?xf32>) -> tensor<?x?xf32>
```

Ops with an operand that bufferize to a memory write but have no tied OpResult are not in destination-passing style. To bufferize such an op, Comprehensive Bufferize must allocate a new memref buffer for the result. Neither `%a` nor `%b` in the example above could be used; they don't even have the correct runtime type. In the future, Comprehensive Bufferize could be extended to bufferize such ops more efficiently by considering buffers apart from tied OpOperands when. Any buffer that is in scope would do, but a new analysis is needed because choosing such a buffer can introduce new RaW conflicts.

Buffer Deallocation

Another problem of bufferization in general is buffer deallocation: When should a buffer be deallocated? In Comprehensive Bufferize, buffers are always deallocated in the same block in which they were allocated. All `memref.alloc`s are essentially `memref.alloca`s. Therefore, reference counting or similar techniques are not needed. However, this means that newly allocated buffers can never escape function boundaries or `scf::ForOp` / `scf::IfOp` blocks; input IR that would bufferize to such IR would fail bufferization.

If a tensor value should be returned from a function (or any operation), a tied tensor should always be provided as an OpOperand (destination-passing style). The OpResult can then bufferize in-place with its tied OpOperand. This is straightforward for ops such as `vector.transfer_write`, which have a tensor input. Ops that do not have input tensors but create tensor values "out of thin air" do not fit into this scheme. They may fail to bufferize. E.g., returning the result of a `linalg.init_tensor` or the above-mentioned TOSA matmul op is currently not possible. Similarly, returning the result of an op that was bufferized out-of-place also fails bufferization.

[6] In C/C++, this is similar to passing output args via a non-const reference instead of returning values from a function.

To be able to bufferize arbitrary IR efficiently (without sacrificing aggressive in-place bufferization of Comprehensive Bufferize, future versions of Comprehensive Bufferize will generate `memref.alloca` ops for all buffers that do not escape block boundaries. The remaining `memref.alloc` ops can be handled by the `buffer-deallocation` pass, which is part of core bufferization.

Unifying Core Bufferization and Comprehensive Bufferize

There are plans to unify core bufferization and Comprehensive Bufferize into a single bufferization framework in the bufferization dialect. The code base of Comprehensive Bufferize is already mostly set up for this. Both bufferizations implement the same contract at the boundary when encountering an op that is not bufferizable.

Bufferizing at the Boundary

Whenever a tensor is passed into an op as an operand, the tensor value is wrapped in a `bufferization.to_memref` op. Tensor results are replaced with the `memref` result of the new op[7], wrapped in a `bufferiation.to_tensor` op. `to_memref(to_tensor(x))` automatically folds to `x`. A `BlockAndValueMapping` could be used in Comprehensive Bufferize instead of materializing `to_tensor`/`to_memref` ops around every bufferized op. This would be more efficient. However, by generating these ops at the boundary, the IR always remains in a valid state and bufferization can be stopped at any time (e.g., when encountering an op that is not yet bufferizable).

```
%0 = "some_op"(%t)
  : (tensor<?xf32>) -> (tensor<?xf32>)
%r = tensor.insert %f into %0[%c0]
  {__inplace__ = ["none", "true", "none"]}
  : f32 into tensor<?xf32>
"another_op"(%r) : (tensor<?xf32>) -> ()
```

alias sets: {{%r, %0}, {%t}}

```
%0 = "some_op"(%t)
  : (tensor<?xf32>) -> (tensor<?xf32>)
%0_m = bufferization.to_memref %0 : memref<?xf32>
memref.store %f, %0_m : memref<?xf32>
%0_t = bufferization.to_tensor %0_m
  : memref<?xf32>
"another_op"(%0_t) : (tensor<?xf32>) -> ()
```

Note that `to_memref`/`to_tensor` ops must be dealth with care. They are in essence special named versions of `unrealized_conversion_cast` and impose certain restrictions on their operands/results [8]. E.g., mutating the result of a `to_memref` op is undefined behavior. Comprehensive Bufferize should be used for full program bufferization, in which case the resulting IR does not have any `to_tensor`/`to_memref` ops anymore. Only when bufferizing a program partially, these ops leak across pass boundaries. This is meant as a transitory feature to be compatible with legacy code that still uses the existing core bufferization. Ideally, these should all be migrated to the unified bufferization in the future.

From Core Bufferization to Comprehensive Bufferize

It is our vision that the new unified bufferization framework will in the future replace all existing uses of core bufferization. This will give users better bufferization results with fewer copies, thanks to Comprehensive Bufferize's analysis. But it will require users to switch from partial bufferization to full program (*one-shot*) bufferization. As a consequence, `to_tensor`/`to_memref` ops will no longer survive bufferization and the aforementioned potentially dangerous restrictions around these ops are no longer an issue.

---

[7] Some ops such as `vector.transfer_write` do not have memref results. In this particular case, all uses of the tensor result of the op are replaced with `to_memref(source)`, where source is the tensor operand of the op.
[8] For more details, check the documentation of these ops.

To ease the transition process, we made sure that Comprehensive Bufferize is compatible with core bufferization. Both bufferizations can be used at the same time. The only restriction is that Comprehensive Bufferize must run before any partial bufferization pass[9]. Comprehensive Bufferize can be configured to ignore certain ops (`dialect-filter`) and allow partial bufferization (`allow-unknown-ops`), meaning that `to_tensor`/`to_memref` ops may survive bufferization. After Comprehensive Bufferize, one or multiple existing partial bufferization pass can run and bufferize the remaining tensor ops.

Partial Bufferization via `BufferizableOpInterface`

Core bufferization passes themselves can also gradually be implemented in terms of `BufferizableOpInterface`. This is what we mean by "unifying" the two bufferizations. There are two different cases that we have to consider.

First, many ops/dialects are already supported in Comprehensive Bufferize. For these dialects, we can directly run Comprehensive Bufferize. E.g., `tensor-bufferize`'s implementation can be replaced by a Comprehensive Bufferize run with `dialect-filter=tensor`, `allow-unknown-ops` and `skip-analysis`. By skipping the analysis, all OpOperands that bufferize to a memory write are copied. This is exactly how the current core bufferization operates. Switching over the implementation of core bufferization passes should be mostly NFC from a user's perspective. Finally, once all users of `tensor-bufferize` are switched over to full program bufferization, i.e., Comprehensive Bufferize (which we can rename to *One-Shot Bufferization* or simply *Bufferization*) without a dialect filter, `tensor-bufferize` could be deleted[10].

Second, ops that are not supported in Comprehensive Bufferize yet, should implement `BufferizableOpInterface`, so that they can be handled by the analysis. Until this happens, the existing partial bufferization passes can simply be run after Comprehensive Bufferize. Once all needed ops implement the interface and all users of the existing partial bufferization passes are switched over, the partial bufferization passes could be deleted.

Note that Comprehensive Bufferize already supports many basic ops (e.g., `tensor.extract` and `vector.transfer_read`) that are not currently supported by their respective core bufferization passes. By reimplementing core bufferization passes in terms of Comprehensive Bufferize, existing users of core bufferization can get support for these ops for free.

Appendix

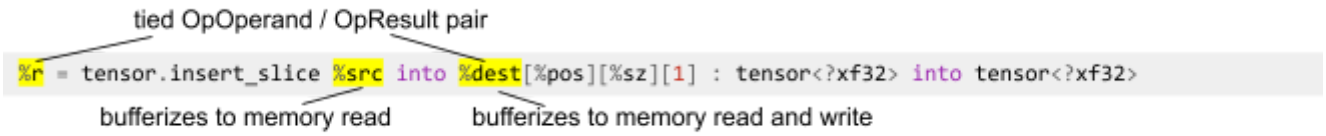
Special Handling of `tensor.extract_slice`/`tensor.insert_slice` Pairs

`linalg.insert_slice` ops have the following bufferization properties: The use of the source tensor bufferizes to a memory read. The use of the dest tensor bufferizes to a read and write.

---

[9] This is because Comprehensive Bufferize cannot analyze tensors that are generated by `to_tensor` ops or passed into `to_memref` ops.

[10] We would like to emphasize that Comprehensive Bufferize is not a complete "reimplementation" of core bufferization, but a mere extension. It is in essence "core bufferization with an analysis". Fundamental design properties such as pattern-based rewriting (the RewritePattern now calls into `BufferizableOpInterface`), copying buffers on every write (for ops that cannot be analyzed) or generating memory deallocations based on a separate analysis in a separate pass (for allocations that escape block boundaries) continue to apply.



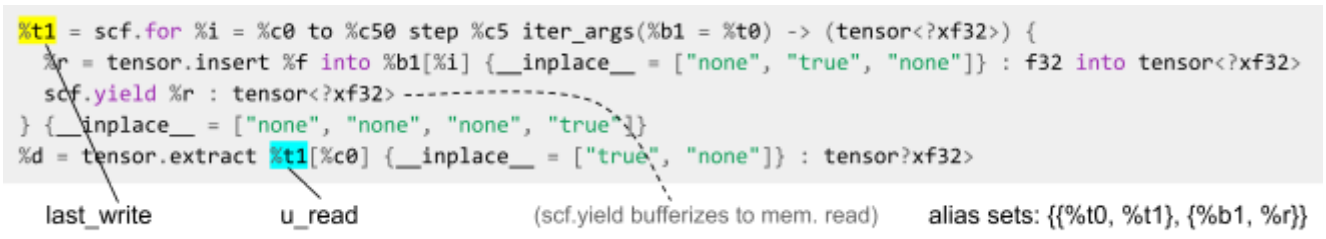
Tiling of Linalg ops produces matching `tensor.extract_slice`/`tensor.insert_slice` op pairs, i.e., they extract from/insert into the same tensor at the same offset and with the same size. This would usually be considered a RaW conflict. There are in fact two RaW conflicts.

In the first case, the read of the dest tensor (`%t0`) of a `tensor.insert_slice` op conflicts with a write to the source tensor (`%t1`). The key insight is that a `tensor.insert_slice` op's dest OpOperand does not read the entire destination tensor, but only the part that is not overwritten. With this in mind, we can formulate an additional rule to prevent such OpOperands from being considered a conflict.

In the second case, the read of the source tensor of a `tensor.insert_slice` op (`%t1`) conflicts with the write of the same op's dest tensor (`%t0`). The key insight here is that if the same data is read and written at the same offset, there is no conflict. Note that the FillOp in the example below already modified the buffer in-place, so there is nothing more to do for the `tensor.insert_slice` op.

Special Handling of `scf.for` Loops

Comprehensive Bufferize assumes that the i-th iter_arg OpOperand of an `scf.for` op is tied to the i-th OpResult. This implies that the i-th yielded value of inside the loop must alias with the i-th block argument of the loop. If this is not the case, bufferization will fail[11].



To simplify RaW detection, loop-related operations are treated as follows during the analysis.

[11] There are plans to relax this restriction by analyzing ops blocks before their parent ops.

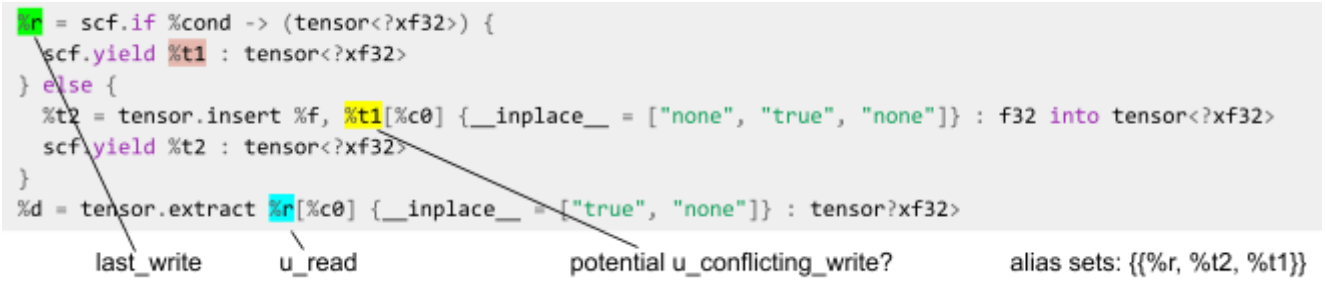
- `scf.for` iter_arg operand: Always bufferizes to a memory write, regardless of the loop body[12]. Note: If an OpOperand is not written to inside the loop, it usually does not make sense to have it as an iter_arg in the first place.
- `scf.yield` operand: Always bufferizes to a memory read.
- The block arguments of a loop do not alias with their corresponding iter_args OpOperands.

These properties make it possible to analyze loop bodies separately from the rest of the function and there is no need to reason about op nesting or the fact that a loop has multiple iterations. E.g., we never even consider conflicts between the write of the `tensor.insert` op and the read of the `tensor.extract` op in the above example, because these OpOperands do not alias. Furthermore, since `scf.yield` is considered a read, there is no need to consider conflicts between loop body ops of different iterations; we analyze the body as if there were only a single iteration.

Special Handling of `scf.if` Branches

`scf.if` ops do not have OpOperands mapping to BBArgs, so they must be treated differently from `scf.for` ops. Since the analysis iterates over OpOperands in the two loops of the RaW algorithm, `scf.if` ops are effectively ignored.

Instead, the i-th result of an `scf.if` is merged with the corresponding yield values of each branch in the alias set data structure before starting the analysis. This is because `scf.yield` is marked as `mustBufferizeInplace` in `BufferizableOpInterface`. Since `scf.if` ops have no tensor operands, they do not have `__inplace__` attributes.



To simplify RaW detection, if-related operations are treated as follows during the analysis.

- `scf.if`: Treated like a memory write during `find_last_write`. I.e., the reverse SSA use-def chain traversal stops when reaching an `scf.if` op.
- `scf.yield`: Always bufferizes to a memory read.

Since `scf.if` ops are treated like a memory write, `find_last_write` can always return a single SSA value and we do not have to deal with sets of SSA values.

A few additional rules are needed in the RaW detection algorithm to prevent false positives.

- Ops of different branches of the same `scf.if` op are not conflicting. E.g., in the above example, the read of `%t1` of the `scf.yield` in the then-branch does not conflict with the write of `%t1` in the `tensor.insert` in the else-branch.
- "opA happens before/after opB" can no longer be defined just through op dominance. E.g., consider two sequential `scf.if` ops. There is no op dominance relationship between an op `a`

[12] From a correctness perspective, it is safe to consider an op as "bufferizes to memory write" even if it does not actually write. However, this could introduce unnecessary out-of-place bufferization decisions.

inside a branch of the first `scf.if` op and an op `b` inside a branch of the second `scf.if` op; nevertheless, `a` always happens before `b` (if they happen at all).
- Writes inside a branch do not conflict with the `scf.if` op result as a last write. Therefore, the tuple shown in the example above is not a conflict.

Op Bufferization Properties ("Bufferization Interface")

The bufferization op interface (`BufferizableOpInterface`) is a way of specifying how ops should be bufferized. Each dialect should implement this interface for all of its ops that take or produce tensors values. In addition to the properties shown in the table below, the interface also provides a function that creates the new memref-based operation and encodes a few additional properties.

The most important interface methods are:

- `bufferizesToMemoryRead(OpOperand &)`: Specifies whether the given OpOperand bufferizes to a memory read.
- `bufferizesToMemoryWrite(OpOperand &)`: Specifies whether the given OpOperand bufferizes to a memory write.
- `getAliasingOpResult(OpOperand &)`: Returns the OpResult that is tied to the given OpOperand (if any).
- `getAliasingOpOperand(OpResult)`: This is the inverse of `getAliasingOpResult` and has a default implementation that queries `getAliasingOpResult`. Note: This function can return multiple OpOperands. This is useful for ops where an OpResult may at runtime be tied to one of multiple possible OpOperands without knowing statically which one it is. E.g., the result of an `scf.if` may be either one of the two corresponding yielded values of both branches.

OpOperand	**Tied OpResult**	**read**	**write**
tensor::CastOp source	result	no	no
tensor::ExtractOp source		yes	no
tensor::ExtractSliceOp source	result	no	no
tensor::InsertOp dest	result	yes	yes
tensor::InsertSliceOp source		yes	no
tensor::InsertSliceOp dest	result	yes	yes
scf::ForOp, i-th iter_arg OpOperand	i-th result	if corresp. BBarg is reading	yes
scf::YieldOp, i-th operand	i-th result of scf::IfOp, none if inside scf::ForOp	yes	no
ReturnOp operand		yes	no
vector::TransferReadOp source		yes	no
vector::TransferWriteOp source	result	yes	yes
linalg::TiledLoopOp input operand		if corresp. BBarg is reading	no
linalg::TiledLoopOp i-th output operand	i-th result	if corresp. BBarg is reading	yes
linalg::LinalgOp input operand		yes	no
linalg::LinalgOp i-th output operand	i-th result	depends on op	yes
CallOpInterface operand	maybe	maybe	maybe
unknown op's operand (analysis only)		yes	yes

`CallOp`s, `FuncOp`s and `ReturnOp`s are outside of the scope of this document. They are not handled by Comprehensive Bufferize and treated like other unknown ops. *Module Bufferization* is an extension of Comprehensive Bufferize that performs additional analyses and is able to bufferize modules with non-circular function call graphs. At the time of this writing, it was still under development and not feature-complete yet.