A Faster and Simpler Dialect Conversion Driver without Pattern Rollback

Matthias Springer (NVIDIA), Markus Böck (NVIDIA / ETH Zürich)

LLVM Dev Meeting 2025 – Technical Talk – October 28, 2025

No-rollback (“One-Shot”) Dialect Conversion Driver

- Development started in May 2024 (RFC). Merged in August 2025.
- A dialect conversion driver that **cannot rollback patterns** / foldings.
  - Materializes all IR changes immediately instead of in a delayed fashion.
  - A few breaking API changes compared to the rollback driver.
- **Faster compilation time** (up to -50%), lower memory usage (up to -90%).
- **Easier to debug**: No hidden C++ state, everything is materialized in IR.
- Better support for listeners and context-aware type conversions.
- Enable with `ConversionConfig::allowPatternRollback = false`.
- Compatible with most existing upstream patterns (61 tests still failing today).

Timeline

Listener / Rewriter API Improvements

Feb 2023 ⟷ March 2024

Internal Refactorings, Bug Fixes, Cleanups (`IRRewrite`, etc.)

Feb 2024 ⟷ Aug 2024

1:N Support

Sept 2024 ⟷ Jan 2025

Context-Aware Type Converter

May 2025 ⟷ Aug 2025

One-Shot Dialect Conversion

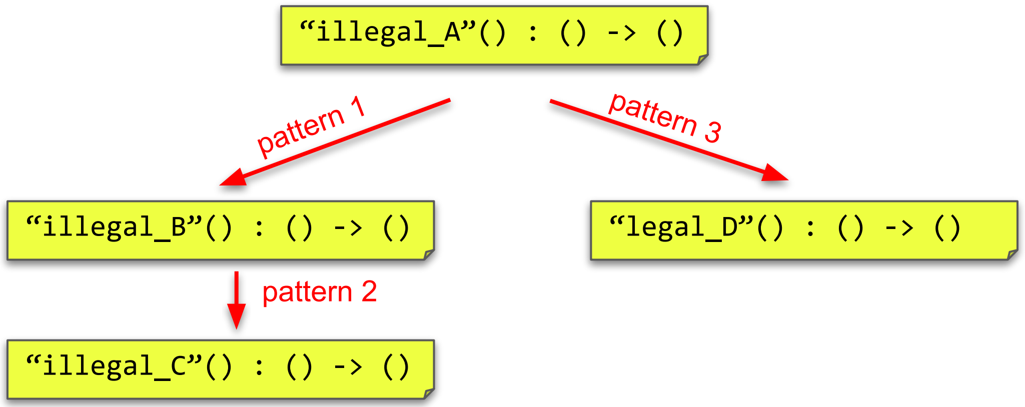
Apr 2025 ⟷ Oct 2025

MemRef 1:N

API Differences

Rollback vs. No Rollback

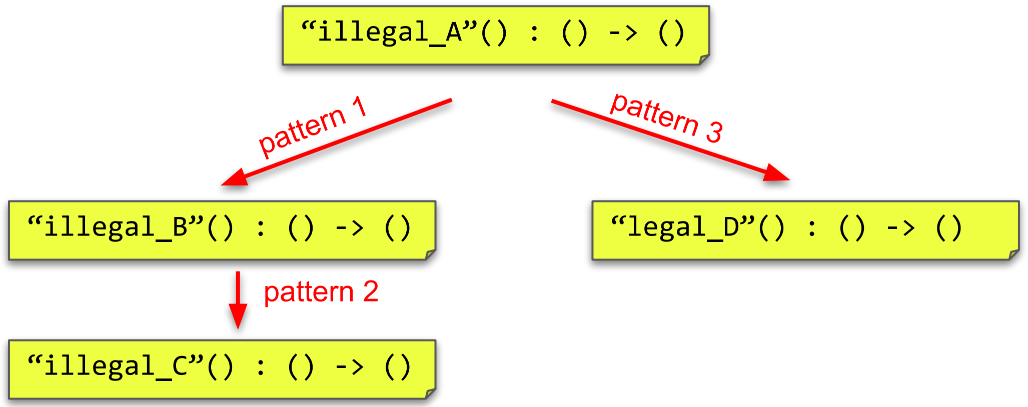
- Old driver (“rollback driver”) can rollback patterns / foldings when hitting a dead end during the lowering process (backtracking).
- One-Shot Dialect Conversion driver (“no-rollback driver”) cannot rollback.



Rollback vs. No Rollback

- Old driver ("rollback driver") can rollback patterns / foldings when hitting a dead end during the lowering process (backtracking).
- One-Shot Dialect Conversion driver ("no-rollback driver") cannot rollback.

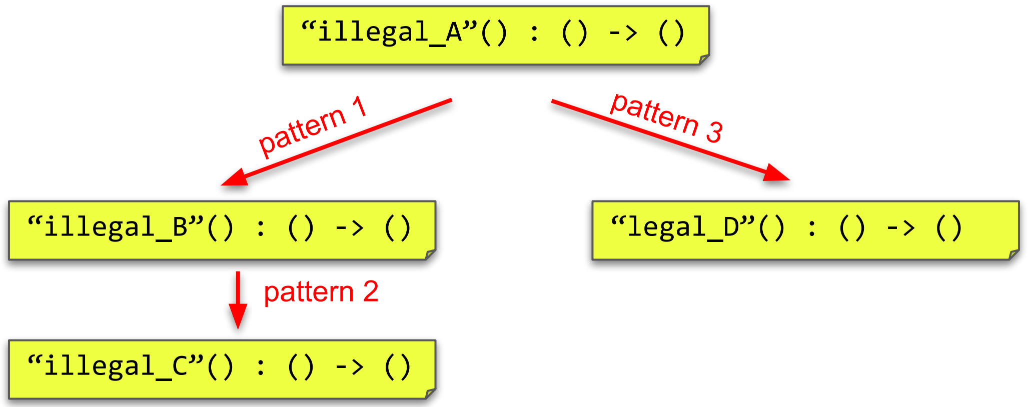
rollback driver:
apply pattern 1
apply pattern 2
rollback pattern 2
rollback pattern 1
apply pattern 3
conversion successful



Rollback vs. No Rollback

- Old driver ("rollback driver") can rollback patterns / foldings when hitting a dead end during the lowering process (backtracking).
- One-Shot Dialect Conversion driver ("no-rollback driver") cannot rollback.

<u>no-rollback driver:</u>
apply pattern 1
apply pattern 2
conversion failed



Delayed vs. Immediate Materialization

- Rollback driver: The driver delays certain IR modifications and maintains a transcript of all IR changes.
  - IR cloning would be simpler, but is too expensive. (Which IR to clone exactly?)
  - Example: Operation erasure is delayed. If it were immediate, the `Operation*` pointer would change on rollback. Furthermore, rollback is easier when the original operation is still around.
  - Example: Operation insertion is immediate. If it were delayed, follow-up pattern could not match it.
  - Old IR more or less stays side-by-side with new IR (and remains accessible to patterns).
- No-rollback driver: Like a normal `PatternRewriter`. All modifications are materialized immediately. Patterns always see only the most recent IR.

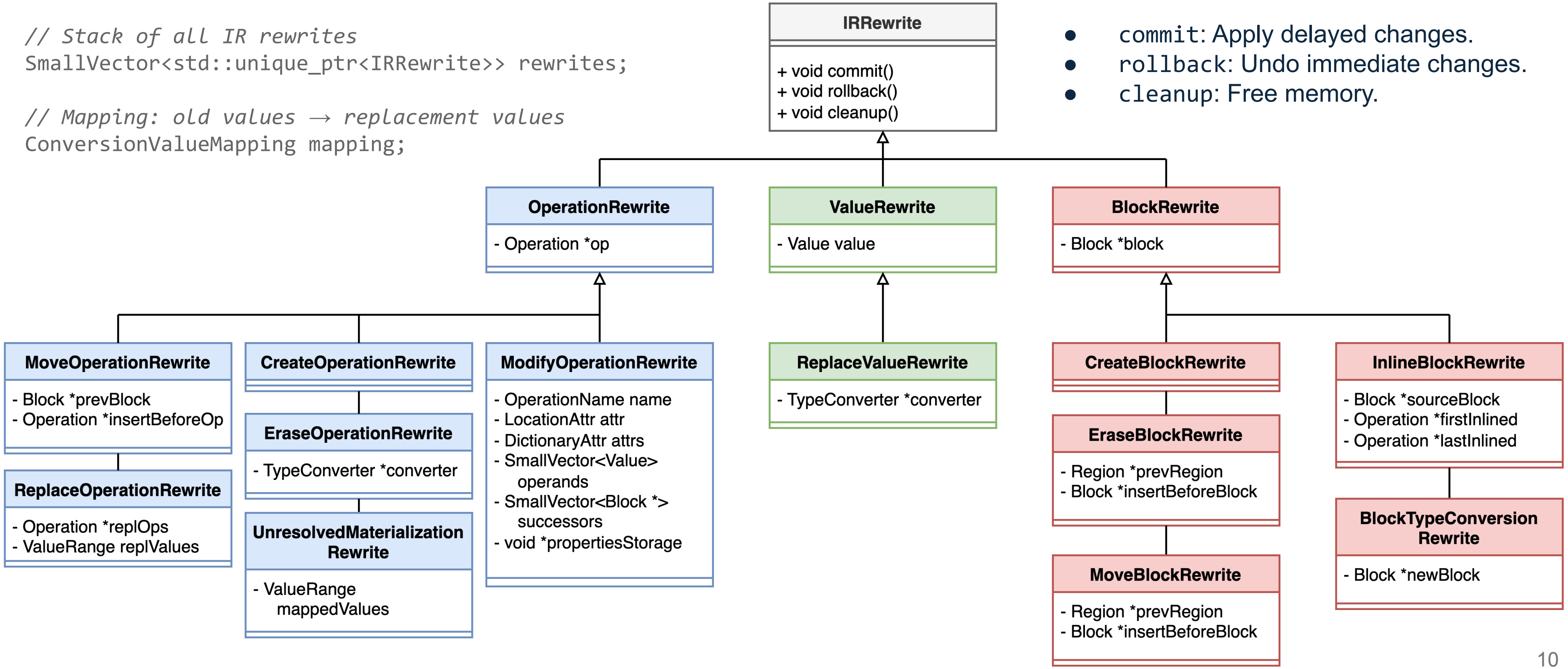
Delayed vs. Immediate Materialization

	Rollback Driver	**No-Rollback Driver**
Op Insertion / Movement	Immediate	Immediate
Op Replacement	Delayed	Immediate
Op Erasure	Delayed	Immediate
Op Modification	Immediate	Immediate
Value Replacement	Delayed, partly supported	Immediate, partly supported
Block Insertion / Movement	Immediate	Immediate
Block Replacement	Not directly supported	Not directly supported
Block Erasure	Partly delayed	Immediate
Block Signature Conversion = Block Insertion + Block Replacement + Op Insertion + Value Replacement + Block Erasure	Partly delayed	Immediate
Region / Block Inlining = Block / Op Insertion (+ Value Replacement)	Partly delayed	Immediate

Rollback Driver: Transcript of IR Modifications

```
// Stack of all IR rewrites
SmallVector<std::unique_ptr<IRRewrite>> rewrites;

// Mapping: old values → replacement values
ConversionValueMapping mapping;
```

- `commit`: Apply delayed changes.
- `rollback`: Undo immediate changes.
- `cleanup`: Free memory.



API Differences: `replaceOp` / `eraseOp`

- No-rollback driver: Operation is immediately erased.
- Rollback driver: Operation is marked for erasure, but stays around until the end of the conversion process.
  - Additional uses of op results can be created, even though the op is marked for erasure. All existing uses are replaced / “dropped” at the end of the conversion process.
  - Patterns that rely on this feature are not compatible with the no-rollback driver.

API Differences: Immediate Erasure

```
rewriter.replaceOp(op, alloc);
rewriter.create<memref::CopyOp>(loc, op.getInput(), alloc);
return success();
```

```
rewriter.create<memref::CopyOp>(loc, op.getInput(), alloc);
rewriter.replaceOp(op, alloc);
return success();
```

Tip: ASAN can detects accesses of erased operations / blocks.

API Differences: `replaceAllUsesWith`

- No-rollback driver: All uses that exist at call time are immediately replaced.
- Rollback driver: Value is replaced at the end of the conversion process.
  - Newly-created uses (after `replaceAllUsesWith`) will also be replaced.
  - Calling `replaceAllUsesWith` on the same value multiple times is not allowed. (Triggers assertion.)
- `replaceAllUsesExcept` / `replaceUsesWithIf` / `replaceOpUsesWithinBlock` are not supported in either driver. (Support will be added to the no-rollback driver eventually.)

API Differences: Failed Patterns Must Not Modify IR

```
auto setupOp = rewriter.create<ConstantOp>(op.getLoc(), 0);
...
if (!precondition(op))
  return failure();
// do rewrite
return success();
```

"matchBeforeRewrite"

```
if (!precondition(op))
  return failure();
auto setupOp = rewriter.create<ConstantOp>(op.getLoc(), 0);
// do rewrite
return success();
```

Tip: Caught by `MLIR_ENABLE_EXPENSIVE_PATTERN_API_CHECKS` assertion.

API Differences: IR Traversal Is Now Safe

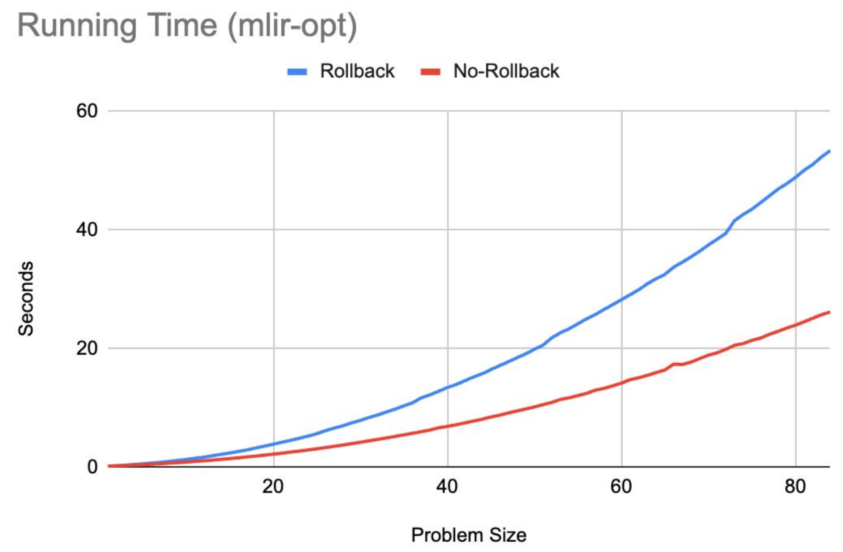
```
if (auto blockArgument = dyn_cast<BlockArgument>(value)
 if (blockArgument.getOwner()->isEntryBlock())
   // Optimized lowering.

// Fallback lowering.
```

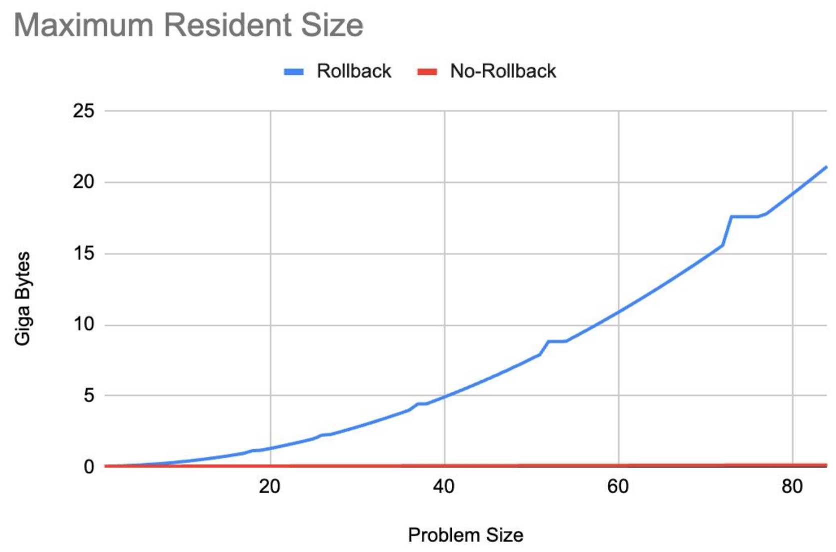
You always see the most recent IR.

https://github.com/llvm/llvm-project/issues/89770

Performance

Performance Comparison

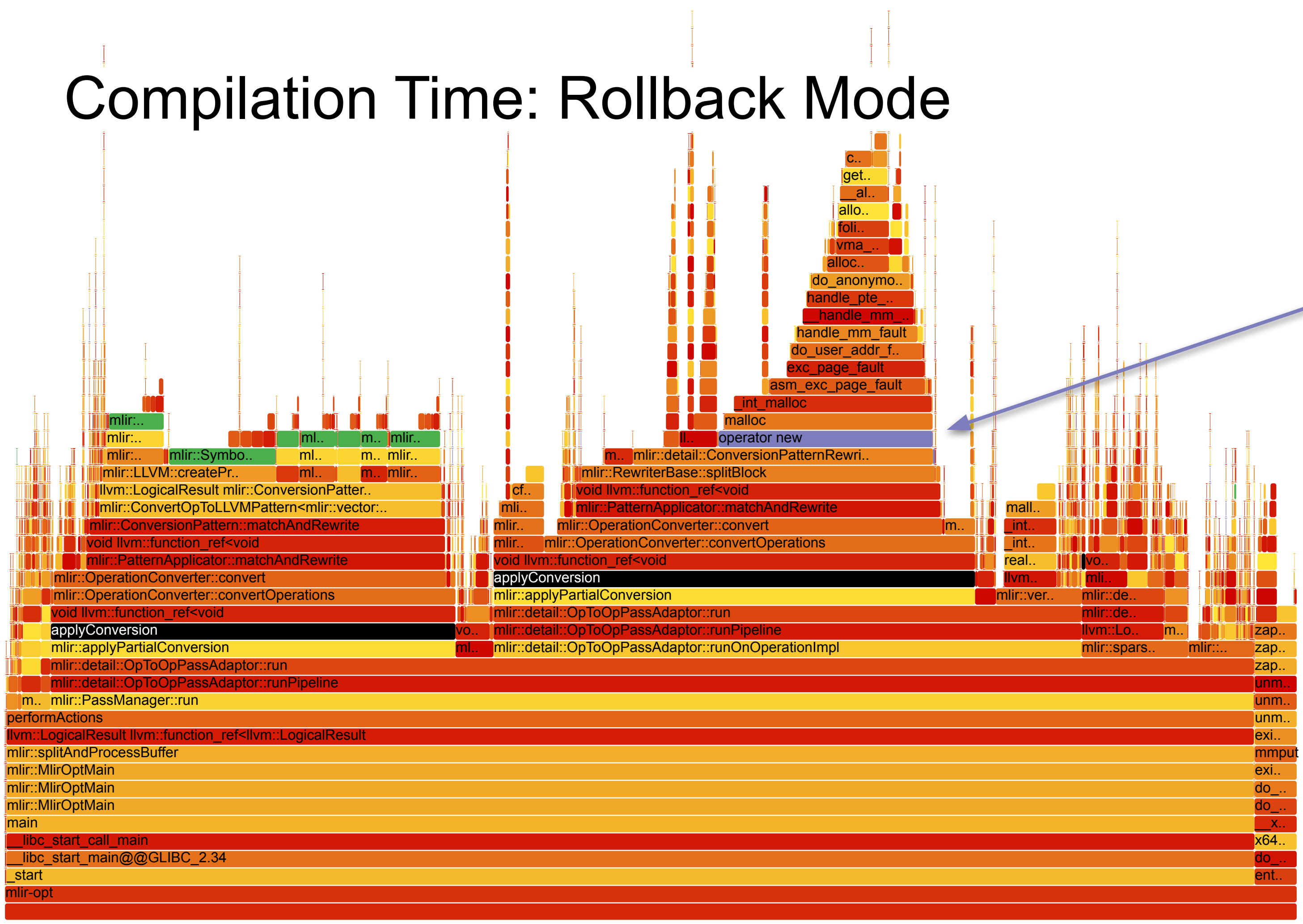




```
mlir-opt mlir/test/Integration/Dialect/SparseTensor/CPU/sparse_conversion.mlir
    --sparsifier="enable-runtime-library=false" -mlir-disable-threading -o /dev/null
```

Build options: `Release`, no assertions

Compilation Time: Rollback Mode



Allocation of `IRRewrite` objects (op insertion), triggered by `splitBlock`. E.g., during SCF → CF.

Build options:
`RelWithDebInfo`, no assertions, `-O2 -g -fno-omit-frame-pointer`

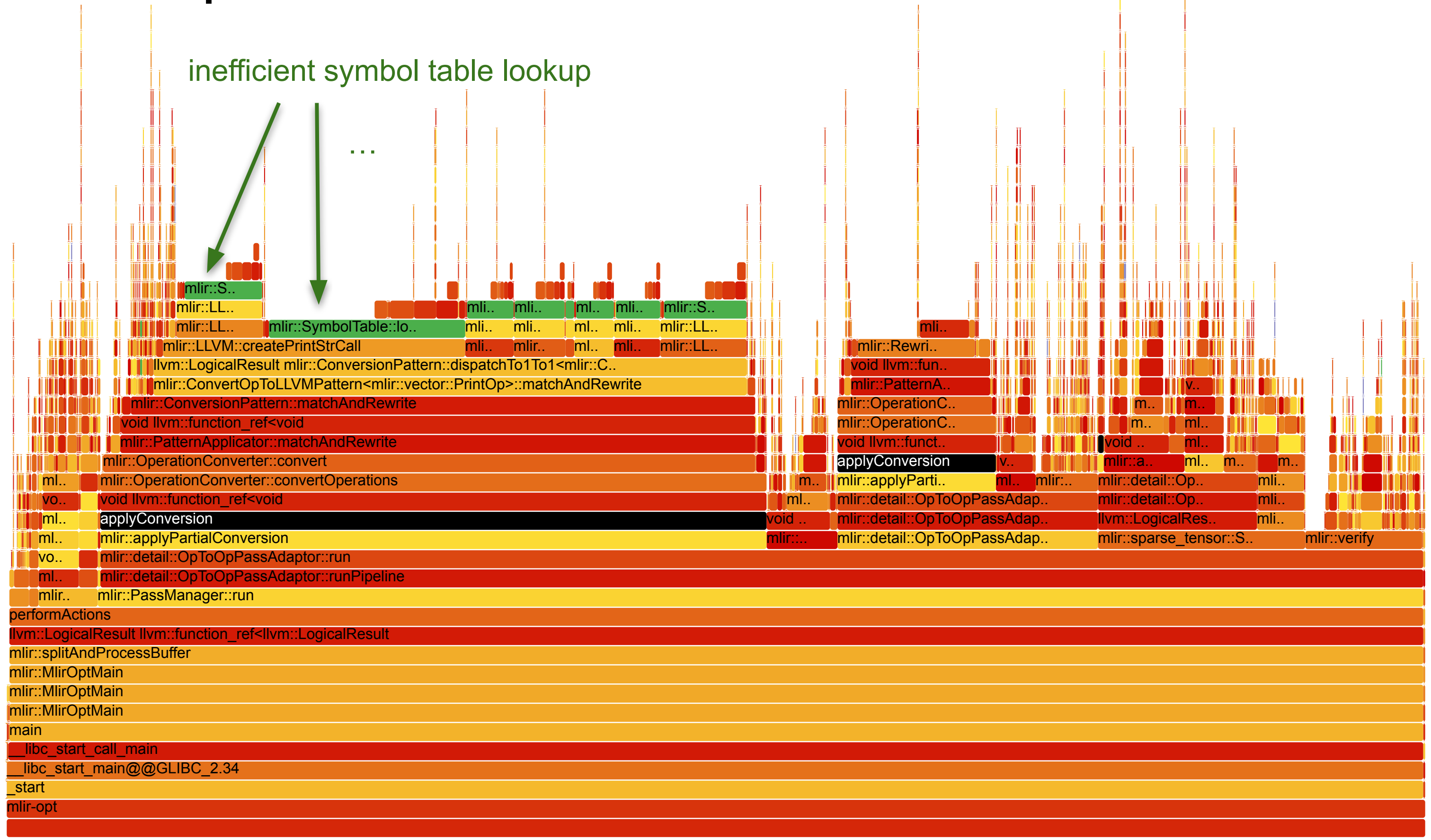
Profiling options:
`perf record -F 500 -g`

Problem size: 24

Compilation Time: No-Rollback Mode

inefficient symbol table lookup

…



19

Migration / Debugging Guide

Example: `--test-emulate-narrow-int`

```
%buf = memref.alloc(%a, %b) : memref<?x?xi4>




// …


%vec = vector.load %buf[%c, %d] : memref<?x?xi4>, vector<128xi4>
```

Example: `--test-emulate-narrow-int`

```
~~%buf = memref.alloc(%a, %b) : memref<?x?xi4>~~
%new_buf = memref.alloc(...) : memref<?xi8>
%buf = builtin.unrealized_conversion_cast %new_buf : memref<?xi8> to memref<?x?xi4>


// …


%vec = vector.load %buf[%c, %d] : memref<?x?xi4>, vector<128xi4>
```

https://gist.github.com/matthias-springer/c7e754d1ec90bf4c35251bc2593624d7

Example: --test-emulate-narrow-int

```
~~%buf = memref.alloc(%a, %b) : memref<?x?xi4>~~
%new_buf = memref.alloc(...) : memref<?xi8>
%buf = builtin.unrealized_conversion_cast %new_buf : memref<?xi8> to memref<?x?xi4>

// …

~~%vec = vector.load %buf[%c, %d] : memref<?x?xi4>, vector<128xi4>~~
%base, %off, %sz0, %sz1, %str0, %str1 = memref.extract_strided_metadata %buf
%new_pos = arith.apply affine_map<()[s0, s1, s2] -> ((s2 + s0 * s1) floordiv 2)>()[%c, %b, %d]
%vec8 = vector.load %new_buf[%new_pos] : memref<?xi8>, vector<64xi8>
%vec = vector.bitcast %vec8 : vector<64xi8> to vector<128xi4>
```

https://gist.github.com/matthias-springer/c7e754d1ec90bf4c35251bc2593624d7

Example: `--test-emulate-narrow-int`

```
~~%buf = memref.alloc(%a, %b) : memref<?x?xi4>~~
%new_buf = memref.alloc(...) : memref<?xi8>
%buf = builtin.unrealized_conversion_cast %new_buf : memref<?xi8> to memref<?x?xi4>

// ...

~~%vec = vector.load %buf[%c, %d] : memref<?x?xi4>, vector<128xi4>~~
%base, %off, %sz0, %sz1, %str0, %str1 = memref.extract_strided_metadata %buf
%new_pos = arith.apply affine_map<()[s0, s1, s2] -> ((s2 + s0 * s1) floordiv 2)>()[%c, %b, %d]
%vec8 = vector.load %new_buf[%new_pos] : memref<?xi8>, vector<64xi8>
%vec = vector.bitcast %vec8 : vector<64xi8> to vector<128xi4>
```

LLVM ERROR: pattern '(anonymous namespace)::**ConvertVectorLoad' produced IR that could not be legalized**. new ops: {**memref.extract_strided_metadata**, affine.apply, affine.max, affine.apply, vector.load, vector.bitcast}, modified ops: {}, inserted block into ops: {}

`ExtractStridedMetadataOpAllocFolder<memref::AllocOp>` no longer matches in no-rollback mode!

https://gist.github.com/matthias-springer/c7e754d1ec90bf4c35251bc2593624d7

Debugging Experience with -debug

```
// RUN: mlir-opt %s -convert-to-llvm

func.func @get_dim_size(%m: memref<?xf32>, %dim: index) -> index {
  %0 = memref.dim %m, %dim : memref<?xf32>
  return %0 : index
}
```

Debugging Experience with -debug: Rollback Mode

```
Trying to match "(anonymous namespace)::FuncOpConversion" -> SUCCESS : pattern applied successfully
// *** IR Dump After Pattern Application ***

"builtin.module"() ({

  "llvm.func"() <{function_type = !llvm.func<i64 (ptr, ptr, i64, i64, i64, i64)>, sym_name = "get_dim_size"}> ({
  ^bb0(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: i64, %arg3: i64, %arg4: i64, %arg5: i64):
    %0 = "memref.dim"(<<UNKNOWN SSA VALUE>>, <<UNKNOWN SSA VALUE>>) : (memref<?xf32>, index) -> index
    "func.return"(%0) : (index) -> ()
  }) : () -> ()

  "func.func"() <{function_type = (memref<?xf32>, index) -> index, sym_name = "get_dim_size"}> ({ }) : () -> ()

})
```

Debugging Experience with -debug: Rollback Mode

```
Trying to match "(anonymous namespace)::FuncOpConversion" -> SUCCESS : pattern applied successfully
// *** IR Dump After Pattern Application ***

"builtin.module"() ({

  "llvm.func"() <{function_type = !llvm.func<i64 (ptr, ptr, i64, i64, i64, i64)>, sym_name = "get_dim_size"}> ({
  ^bb0(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: i64, %arg3: i64, %arg4: i64, %arg5: i64):
    %0 = "memref.dim"(<<UNKNOWN SSA VALUE>>, <<UNKNOWN SSA VALUE>>) : (memref<?xf32>, index) -> index
    "func.return"(%0) : (index) -> ()
  }) : () -> ()

  "func.func"() <{function_type = (memref<?xf32>, index) -> index, sym_name = "get_dim_size"}> ({ }) : () -> ()

})
```

values are not immediately replaced

operations are not immediately erased

Debugging Experience with `-debug`: No-Rollback Mode

```
Trying to match "(anonymous namespace)::FuncOpConversion" -> SUCCESS : pattern applied successfully
// *** IR Dump After Pattern Application ***

"builtin.module"() ({

  "llvm.func"() <{function_type = !llvm.func<i64 (ptr, ptr, i64, i64, i64, i64)>, sym_name = "get_dim_size"}> ({
  ^bb0(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: i64, %arg3: i64, %arg4: i64, %arg5: i64):
    %0 = "builtin.unrealized_conversion_cast"(%arg5) {__pure_type_conversion__} : (i64) -> index
    %1 = "builtin.unrealized_conversion_cast"(%arg0, %arg1, %arg2, %arg3, %arg4) {__pure_type_conversion__}
        : (!llvm.ptr, !llvm.ptr, i64, i64, i64) -> memref<?xf32>
    %2 = "memref.dim"(%1, %0) : (memref<?xf32>, index) -> index
    "func.return"(%2) : (index) -> ()
 }) : () -> ()

}) : () -> ()
```

More dump friendly!

Conclusion

Lessons Learned: Getting Large Refactorings Merged

- Be prepared: It's not just about the refactored component, but also its **up/downstream uses**.
- Post RFC on Discourse: **keep it short**, show before/after IR/API (+migration guide), highlight what downstream users can gain from your feature, ideally **attach a prototype PR**.
- **Send many smalls PRs**. The smaller the better. Nobody wants to review 150+ LoC changes.
  - Split out NFC changes. Small code improvements can already get merged, while the larger design is still being reviewed.
  - For breaking API changes: Add guide for LLVM integration to commit message. "Prepare" upstream call sites beforehand.
  - 146 commits to date for the dialect conversion driver. Probably same number of PRs that update / prepare call sites.
- **Merge + request reviews slowly**: Leave a week between two consecutive large/API-changing PRs. Give downstream users time to integrate changes piece-by-piece and report problems.
- For code with insufficient upstream test coverage and/or unclear API:
  - **Reach out to downstream users for a dry-run.** (Time is precious: choose wisely which PRs to send.)
  - Use assertions eagerly. Assertion/documentation-only PRs can be useful in preparation of breaking API changes.
  - Be ready to roll back commits and change your design.
- Find motivated reviewers. Typically folks who are interested in your feature/refactoring.

Conclusion

- Dialect conversion driver can operate in two modes: rollback / no-rollback.
- Use the no-rollback version when possible.
  - No-rollback driver is **faster, easier to debug and uses less memory.**
  - No-rollback driver is **better suited for context-aware type conversions** because there are no uses (operands) of block arguments of unlinked blocks at any point during the conversion.
  - No-rollback driver provides more accurate and **immediate listener notifications**.
  - We are trying to eventually **deprecate + remove the rollback driver** to simplify the codebase. (This would probably reduce the driver code by around 50%.)
- Potential future work
  - More performance optimizations. E.g., pooling `unrealized_conversion_cast` ops.
  - Align even more with `RewritePattern` API. E.g., disallow erasure of ops that still have uses.

Questions?

`ConversionConfig::allowPatternRollback`
Pattern Rollback
Folder Rollback
Backtracking
`RewritePattern` / `ConversionPattern`
`ConversionPatternRewriter`
Op / Block Insertion
Op / Replacement
Op / Block Erasure
Block Signature Conversion
Value / Block Replacement
Block / Region Inlining
`IRRewrite`

Compilation Time
Memory Usage
IR Traversal
Old IR Side-by-side
Immediate / Delayed Materialization
Match Before Rewrite
Expensive Pattern Checks
Migration to No-rollback Mode
`-debug` / Dumping IR
`builtin.unrealized_conversion_cast`
`<<UNKNOWN SSA VALUE>>`
Context-aware Type Conversion
Deprecation of Rollback Driver

Appendix

Folder Rollback Error

Attempted pattern rollback:

```
LLVM ERROR: pattern '(anonymous namespace)::ConvertVectorLoad' produced IR that could
not be legalized. new ops: {memref.extract_strided_metadata, affine.apply,
affine.max, affine.apply, vector.load, vector.bitcast}, modified ops: {}, inserted
block into ops: {}
```

Attempted folder rollback (when a materialized constant cannot be legalized):

```
LLVM ERROR: op 'arith.fptosi' folder rollback of IR modifications requested
```