CRASH REPRODUCERS IN MLIR

Artemiy Bulavin
artemiyb@graphcore.ai

GRAPHCORE

WHAT THIS TALK IS ABOUT

Compilation pipelines can be complex and nuanced.

Crashes that result from buggy transformations become hard to track down.

Reproducing these bugs becomes hard.

What does MLIR have to help with this?

REPRODUCING BUGS 101

C++ unit test

```
Value of: isEven(16)
  Actual: false
Expected: true

[  FAILED  ] MyIsEvenFunc.TestItWorks
[----------] 1 test from MyIsEvenFunc

[  FAILED  ] 1 test, listed below:
[  FAILED  ] MyIsEvenFunc.TestItWorks

 1 FAILED TEST
```

```
TEST(MyIsEvenFunc, TestItWorks) {
  ASSERT_TRUE(isEven(16));
}
```

```
$ ctest -R TestIsEven
```

1. An output that you get from the (buggy) program

2. The output you expect to see

3. A way to reliably show that:

what you're getting != what you're expecting

REPRODUCING BUGS 101

Lit test

```
tor<4xf32>, %arg1: vector<4xf32>, %arg2: f32) -> f32
xf32>
     X error: no match found
       with "F" equal to "0"
       with "C" equal to "arg2"
: vector<4xf32> into f32
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                          possible intended match
```

```
// CHECK-LABEL: func @extract_co
// CHECK-SAME: %[[A:.*0]]: vecto
// CHECK-SAME: %[[B:.*1]]: vecto
// CHECK-SAME: %[[C:.*2]]: f32
// CHECK:      %[[F:.*]] = arith
// CHECK:      %[[R:.*]] = vecto
// CHECK:      return %[[R]] : f
```

```
// RUN: mlir-opt %s \
//      --transform-interpreter \
//      | FileCheck %s
```

1. An output that you get from the (buggy) program

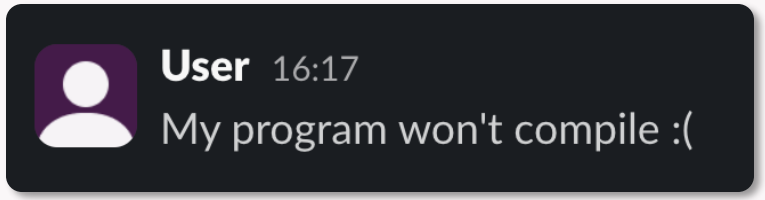
2. The output you expect to see

3. A way to reliably show that:

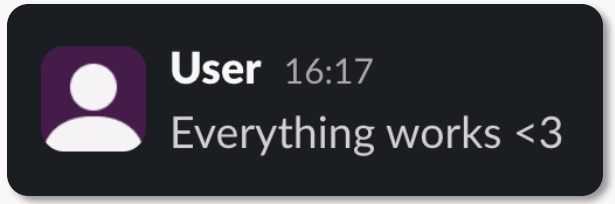
what you're getting != what you're expecting

REPRODUCING BUGS 101

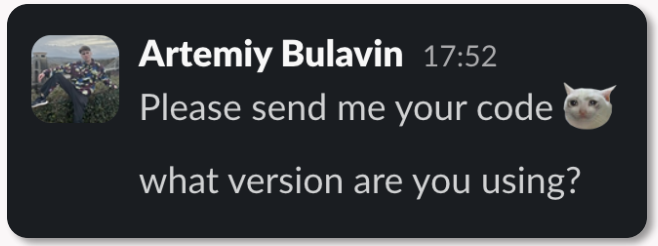
Full compilation pipeline



1. An output that you get from the (buggy) program



2. The output you expect to see



3. A way to ~~reliably~~ show that:

what you're getting != what you're expecting

EXAMPLE

Suppose we have a bug in the `ConvertVectorToLLVM` pass that won't lower `vector.contract` correctly:

```
mlir-opt --convert-vector-to-llvm
         vector-contract-to-dot-transforms.mlir
```

```
error: 'llvm.insertvalue' op Type mismatch: cannot insert 'f32' into '!llvm.array<2 x vector<2xf32>>'
  %res = vector.contract #matmat_trait %lhs, %rhs, %init
         ^
within split at vector-contract-to-dot-transforms.mlir:1 offset :201:10: note: see current operation:
%39 = "llvm.insertvalue"(%3, %38) <{position = array<i64: 0>}> : (!llvm.array<2 x vector<2xf32>>,
f32) -> !llvm.array<2 x vector<2xf32>>
```

EXAMPLE

Suppose we have a bug in the `ConvertVectorToLLVM` pass that won't lower `vector.contract` correctly:

We can tell `mlir-opt` to create a crash reproducer

```
mlir-opt --convert-vector-to-llvm
         --mlir-pass-pipeline-crash-reproducer=repro.mlir
         vector-contract-to-dot-transforms.mlir
```

```
error: 'llvm.insertvalue' op Type mismatch: cannot insert 'f32' into '!llvm.array<2 x vector<2xf32>>'
  %res = vector.contract #matmat_trait %lhs, %rhs, %init
         ^
within split at vector-contract-to-dot-transforms.mlir:1 offset :201:10: note: see current operation:
%39 = "llvm.insertvalue"(%3, %38) <{position = array<i64: 0>}> : (!llvm.array<2 x vector<2xf32>>,
f32) -> !llvm.array<2 x vector<2xf32>>
within split at vector-contract-to-dot-transforms.mlir:1 offset :0:0: error: Failures have been
detected while processing an MLIR pass pipeline
within split at vector-contract-to-dot-transforms.mlir:1 offset :0:0: note: Pipeline failed while
executing [`ConvertVectorToLLVMPass` on 'builtin.module' operation]: reproducer generated at
`repro.mlir`
```

repro.mlir U

mlir > test > Dialect > Vector > repro.mlir

```
#map = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
module {
  func.func @contract_to_dot_matmat(%arg0: vector<2x2xf32>,
                                    %arg1: vector<2x2xf32>,
                                    %arg2: vector<2x2xf32>) -> vector<2x2xf32> {
    %0 = vector.contract {
      indexing_maps = [#map, #map1, #map2],
      iterator_types = ["parallel", "parallel", "reduction"],
      kind = #vector.kind<add>} %arg0, %arg1, %arg2 : vector<2x2xf32>, vector<2x2xf32> into vector<2x2xf32>
    return %0 : vector<2x2xf32>
  }
}

{-#
  external_resources: {
    mlir_reproducer: {
      pipeline: "builtin.module(convert-vector-to-llvm{enable-amx=false enable-arm-i8mm=false enable-arm-neon=false enable-arm-sve=false er
      force-32bit-vector-indices=true reassociate-fp-reductions=false use-vector-alignment=false vector-contract-lowering=dot vector-transp
      ",
      disable_threading: false,
      verify_each: true
    }
  }
#-}
```

repro.mlir U

mlir > test > Dialect > Vector > repro.mlir

```
#map = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
module {
  func.func @contract_to_dot_matmat(%arg0: vector<2x2xf32>,
                                    %arg1: vector<2x2xf32>,
                                    %arg2: vector<2x2xf32>) -> vector<2x2xf32> {
    %0 = vector.contract {
      indexing_maps = [#map, #map1, #map2],
      iterator_types = ["parallel", "parallel", "reduction"],
      kind = #vector.kind<add>} %arg0, %arg1, %arg2 : vector<2x2xf32>, vector<2x2xf32> into vector<2x2xf32>
    return %0 : vector<2x2xf32>
  }
}

{-#
  external_resources: {
    mlir_reproducer: {
      pipeline: "builtin.module(convert-vector-to-llvm{enable-amx=false enable-arm-i8mm=false enable-arm-neon=false enable-arm-sve=false er
      force-32bit-vector-indices=true reassociate-fp-reductions=false use-vector-alignment=false vector-contract-lowering=dot vector-transp
      ",
      disable_threading: false,
      verify_each: true
    }
  }
#-}
```

The IR to reproduce

repro.mlir U

mlir > test > Dialect > Vector > repro.mlir

```
#map = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
module {
  func.func @contract_to_dot_matmat(%arg0: vector<2x2xf32>,
                                    %arg1: vector<2x2xf32>,
                                    %arg2: vector<2x2xf32>) -> vector<2x2xf32> {
    %0 = vector.contract {
      indexing_maps = [#map, #map1, #map2],
      iterator_types = ["parallel", "parallel", "reduction"],
      kind = #vector.kind<add>} %arg0, %arg1, %arg2 : vector<2x2xf32>, vector<2x2xf32> into vector<2x2xf32>
    return %0 : vector<2x2xf32>
  }
}

{-#
  external_resources: {
    mlir_reproducer: {
      pipeline: "builtin.module(convert-vector-to-llvm{enable-amx=false enable-arm-i8mm=false enable-arm-neon=false enable-arm-sve=false er
      force-32bit-vector-indices=true reassociate-fp-reductions=false use-vector-alignment=false vector-contract-lowering=dot vector-transp
      ",
      disable_threading: false,
      verify_each: true
    }
  }
#-}
```

The passes to reproduce

repro.mlir U

mlir > test > Dialect > Vector > repro.mlir

```
#map = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
module {
  func.func @contract_to_dot_matmat(%arg0: vector<2x2xf32>,
                                    %arg1: vector<2x2xf32>,
                                    %arg2: vector<2x2xf32>) -> vector<2x2xf32> {
    %0 = vector.contract {
      indexing_maps = [#map, #map1, #map2],
      iterator_types = ["parallel", "parallel", "reduction"],
      kind = #vector.kind<add>} %arg0, %arg1, %arg2 : vector<2x2xf32>, vector<2x2xf32> into vector<2x2xf32>
    return %0 : vector<2x2xf32>
  }
}

{-#
  external_resources: {
    mlir_reproducer: {
      pipeline: "builtin.module(convert-vector-to-llvm{enable-amx=false enable-arm-i8mm=false enable-arm-neon=false enable-arm-sve=false er
      force-32bit-vector-indices=true reassociate-fp-reductions=false use-vector-alignment=false vector-contract-lowering=dot vector-transp
      ",
      disable_threading: false,
      verify_each: true
    }
  }
#-}
```

PassManager and MLIRContext options

- Reproducers can be created using `mlir-opt` and the `PassManager`
- Reproducers are **external resources.**
- Handled by `AsmParser`

```
{-#
      dialect_resources: {
            builtin: {
                  blob1: "0x08000000010000000000000002000000000000",
            }
      },
      external_resources: {
            external: {
                  blob: "0x08000000010000000000000002000000000000",
            },
            other_stuff: {
                  bool: true
            }
      }
#-}
```

EXAMPLE

Once you have the reproducer, you ran run it:

```
mlir-opt --run-reproducer repro.mlir
```

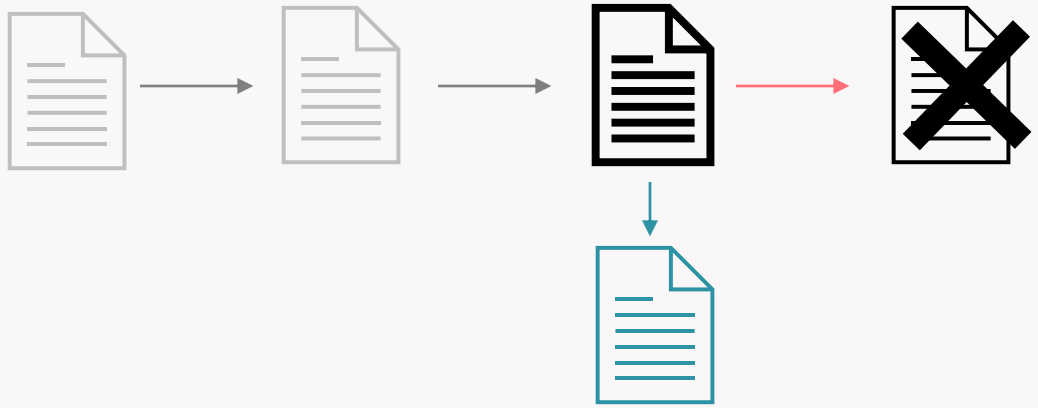
```
error: 'llvm.insertvalue' op Type mismatch: cannot insert 'f32' into '!llvm.array<2 x vector<2xf32>>'
  %res = vector.contract #matmat_trait %lhs, %rhs, %init
         ^
within split at vector-contract-to-dot-transforms.mlir:1 offset :201:10: note: see current operation:
%39 = "llvm.insertvalue"(%3, %38) <{position = array<i64: 0>}> : (!llvm.array<2 x vector<2xf32>>,
f32) -> !llvm.array<2 x vector<2xf32>>
```

LOCAL VS FULL REPRODUCERS

Reproducers contain the initial IR when you run the passes.

`repro.mlir`

Local reproducers contain the IR and passes *just before failure.*



`repro_local.mlir`

Our `ConvertVectorToLLVM` lowering contains a bug that affects `vector.contract`

What will **full** and **local** the reproducers for this IR look like?

(Demo)

```
#matmat_accesses = [
        // ...
]
#matmat_trait = {
        // ...
}
#map0 = // ...

func.func @contract_to_dot_matmat(
        %memref_lhs: memref<?x?xf32>, %rhs: vector<2x2xf32>,
        %init: vector<2x2xf32>) -> vector<2x2xf32> {
        %c0 = arith.constant 0 : index
        %cst = arith.constant 0.0 : f32

        %lhs = vector.transfer_read %memref_lhs[%c0, %c0], %cst { permutation_map=#map0 } :
        memref<?x?xf32>, vector<2x2xf32>

        %res = vector.contract #matmat_trait %lhs, %rhs, %init
        : vector<2x2xf32>, vector<2x2xf32> into vector<2x2xf32>
        return %res : vector<2x2xf32>
}
```

Use the PassManager:

- `PassManager::enableCrashReproducerGeneration(outputFile, generateLocal);`
  - Analogous to `--mlir-pass-pipeline-crash-reproducer` and `--mlir-pass-pipeline-local-reproducer`
- `mlir::makeReproducer(anchorName, passes, op, outputFile);`
  - Analogous to `--mlir-generate-reproducer`

Let’s look at an example out in the wild: **Triton**

EXAMPLES OUT IN THE WILD

```
auto reproducerPath =
        triton::tools::getStrEnv("TRITON_REPRODUCER_PATH");
if (!reproducerPath.empty()) {
        auto anchorName = self.getOpAnchorName();
        auto passes = self.getPasses();
        Operation *op = mod.getOperation();
        // Save a reproducer for the current pass manager invocation
        // immediately.
        makeReproducer(anchorName, passes, op, reproducerPath);
        // But if the pass manager crashes, attempt to generate a local
        // reproducer instead.
        context->disableMultithreading();
        self.enableCrashReproducerGeneration(reproducerPath,
                /*genLocalReproducer=*/true);
} else {
        self.enableCrashReproducerGeneration(makeConsoleReproducer());
}
// ... Later on, PassManager::run
```

https://github.com/triton-lang/triton/blob/main/python/src/ir.cc

BEST PRACTICES

Reduce the problem size:

- ☑ Remove passes that have no effect
- ☑ Remove ‘safe’ passes like canonicalisation, CSE
- ☑ Remove ops and values while still seeing the bug

If still unclear:

- ☑ Use a full reproducer, not local

USING REPRODUCERS WITH MLIR-REDUCE

- `mlir-reduce` is a tool for reducing the size of input IR that contains a bug.
- Iteratively applies transformations to the input to ‘reduce’ it, while checking the bug remains.
- You write an ‘interestingness script’, telling `mlir-reduce` whether the reduced IR contains the bug or not.
- `mlir-reduce` trims the input till it’s not ‘interesting’ anymore.

INTERESTINGNESS SCRIPT

Put the pipeline from the reproducer here

```
mlir-opt "$1" --pass-pipeline="..." 2>&1 | \
      grep -q "cannot insert 'f32' into '!llvm.array"


# Exit code 1 if grep finds the error, 0 otherwise.
if [ $? -eq 0 ]; then
      exit 1
else
      exit 0
fi
```

```
$ mlir-reduce --reduction-tree='traversal-mode=0 test=interesting.sh'
              reproducer.mlir
```

INTERESTINGNESS SCRIPT

```
mlir-opt "$1" –my-broken-pass-1 –my-broken-pass-2 ...

if [ $? -neq 0 ]; then
        exit 1
else
        exit 0
fi
```

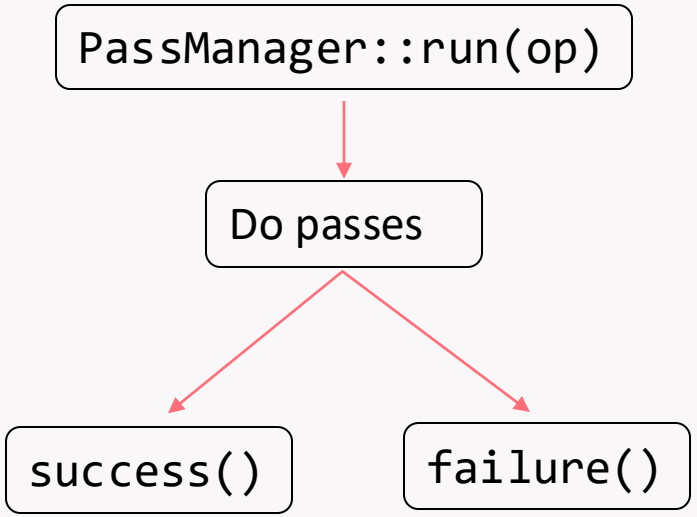
```
$ mlir-reduce --reduction-tree='traversal-mode=0 test=interesting.sh'
              reproducer.mlir
```

INTERESTINGNESS SCRIPT

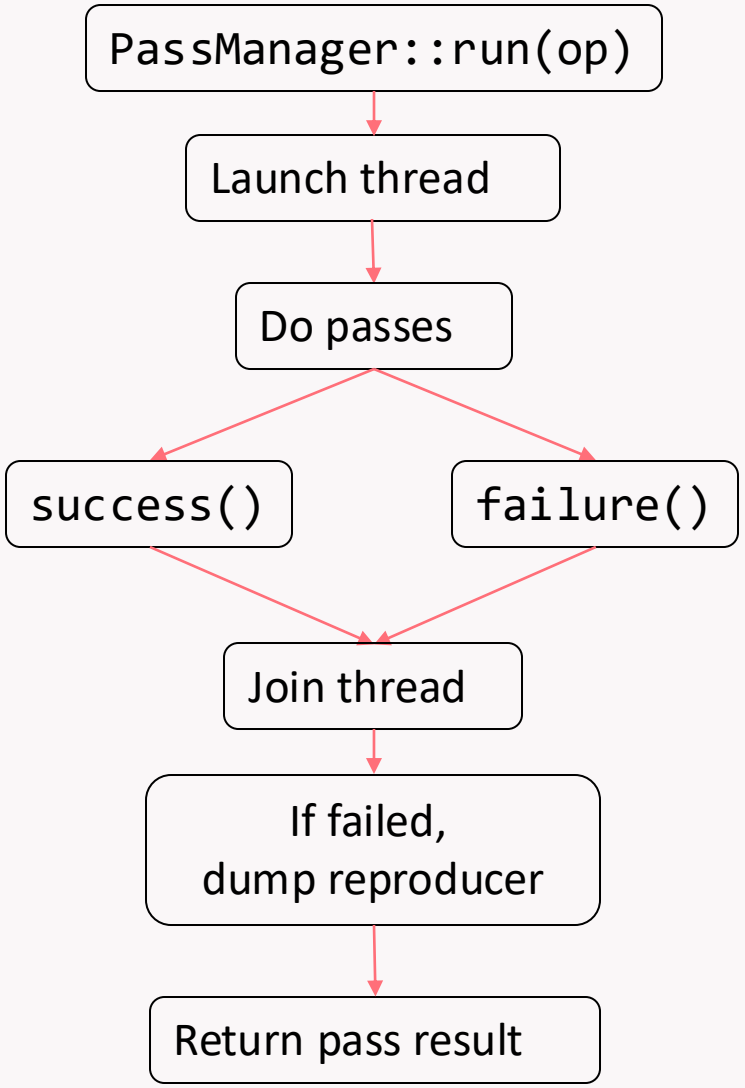
```
mlir-runner "$1" ... 2>&1 | grep ”error"

if [ $? -eq 0 ]; then
        exit 1
else
        exit 0
fi
```

```
$ mlir-reduce --reduction-tree='traversal-mode=0 test=interesting.sh'
              reproducer.mlir
```

HOW DOES IT WORK?



Normal Run



Crash reproduction enabled

1. Pass options need to be *printable*

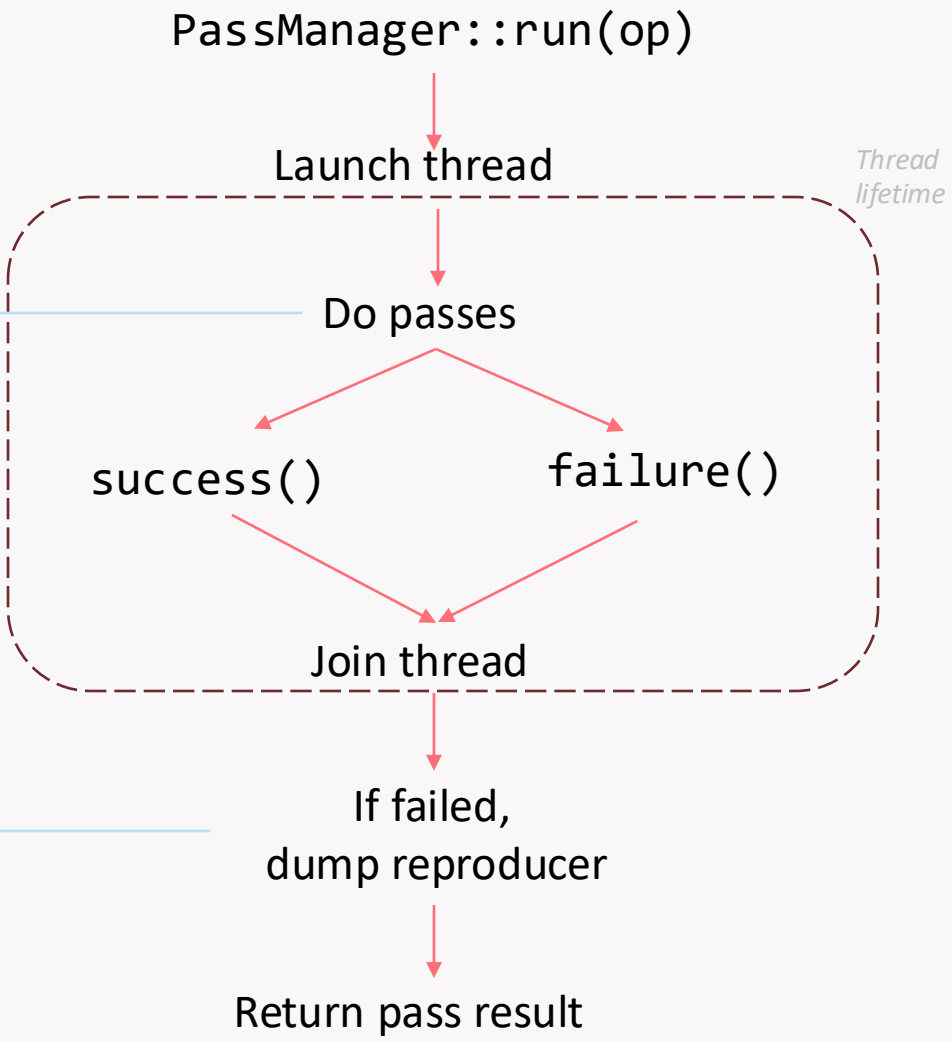
```
def ConvertVectorToLLVMPass : Pass<"convert-vector-to-llvm"> {
// ...
Option<"vectorTransformsOptions", "vector-transform-options",
"vector::VectorTransformsOptions",
"vector::VectorTransformsOptions()",
”...">,
}
```

1. Pass options need to be *printable*

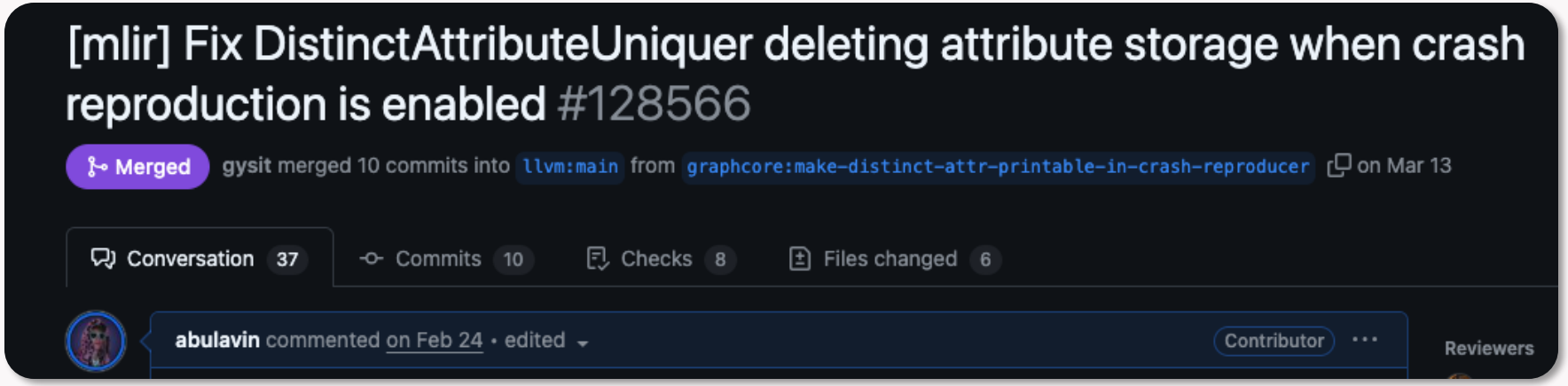
2. Be careful with thread-local state

😎 *thread_local SomeStorage x = ...;*

😭 *auto stored = x.get();*



2. Avoid thread-local state



SUMMARY

Creating reproducers

Using mlir-opt and the PassManager

Integration

Using it in a real compiler

Using reproducers

Best practices for streamlining debugging

Internals

Everything you need to know to use reproducers seamlessly

WHAT YOU CAN DO NEXT

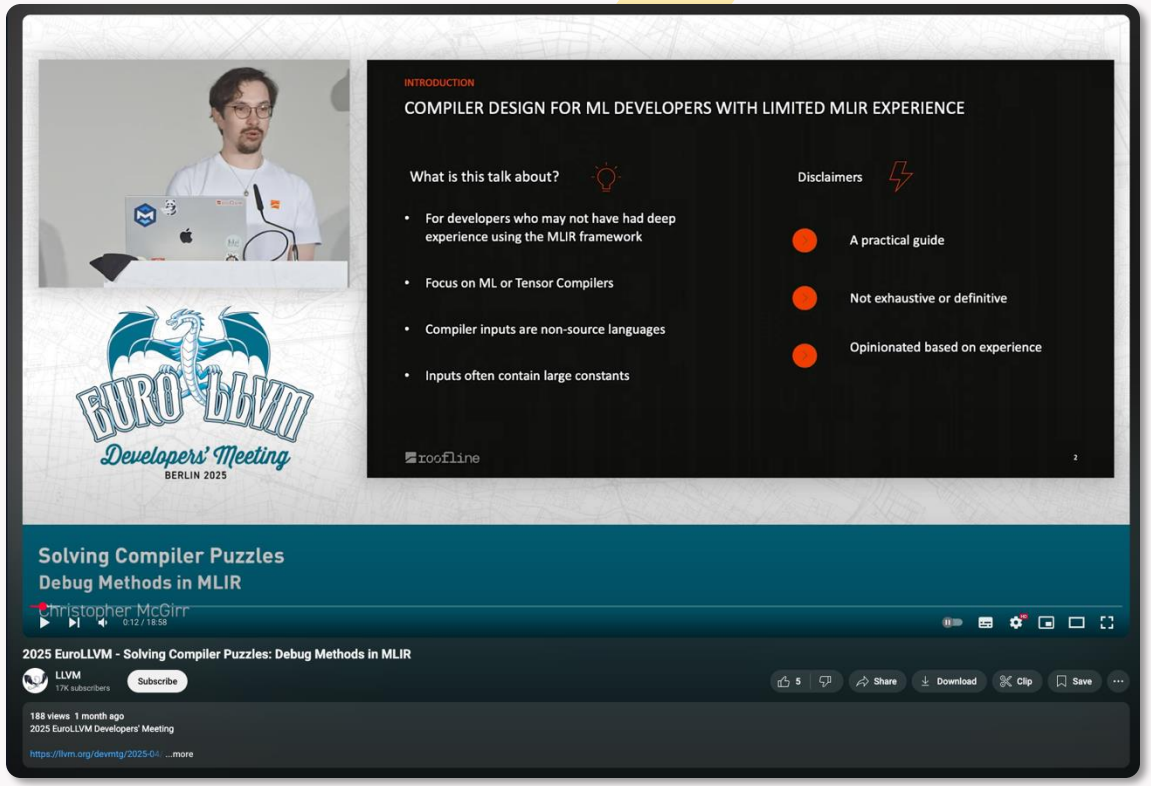
1. Try the reproducers!
   - Integrate them in your compilers
   - Encourage your users to include them in bug reports
2. Add lit tests that exercise reproducers
   - Both upstream and in your projects

MORE RESOURCES

For more tools and tips on debugging MLIR:

2025 EuroLLVM - Solving Compiler Puzzles: Debug Methods in MLIR

Christopher McGirr



THANK YOU

Artemiy Bulavin

artemiyb@graphcore.ai

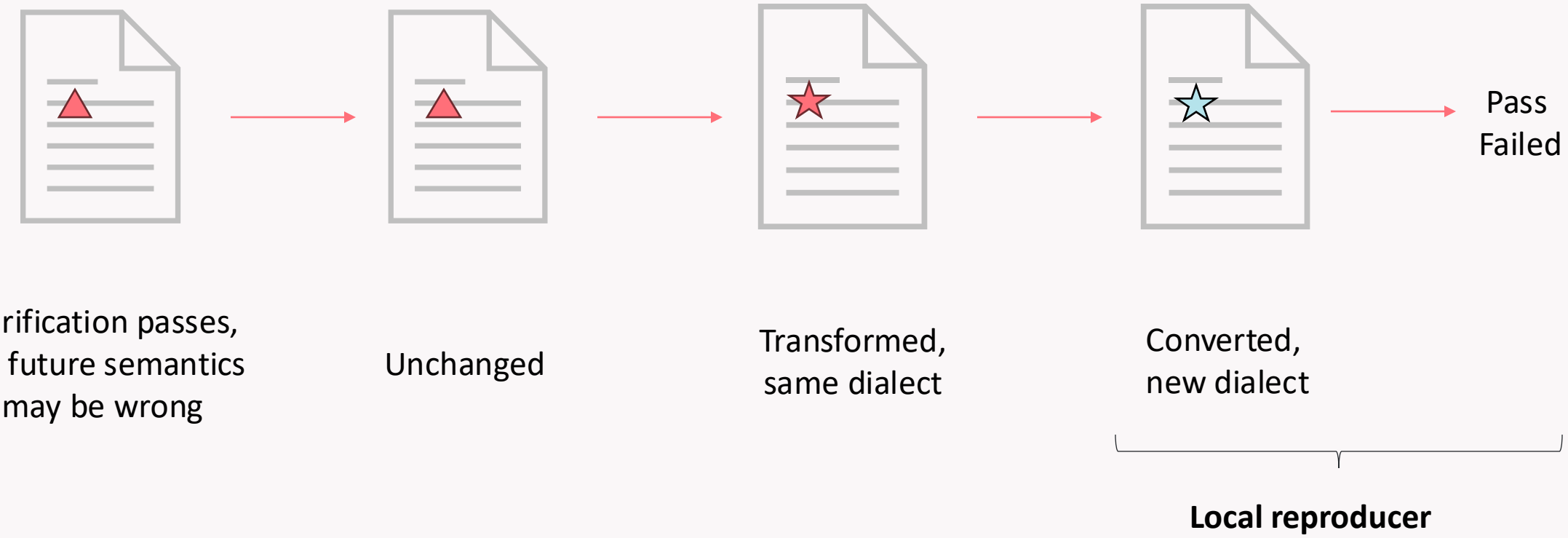
Github/Discourse: @abulavin

BONUS SLIDES

LOCAL VS FULL REPRODUCER

Local reproducers capture the state of the IR *just before failure…*

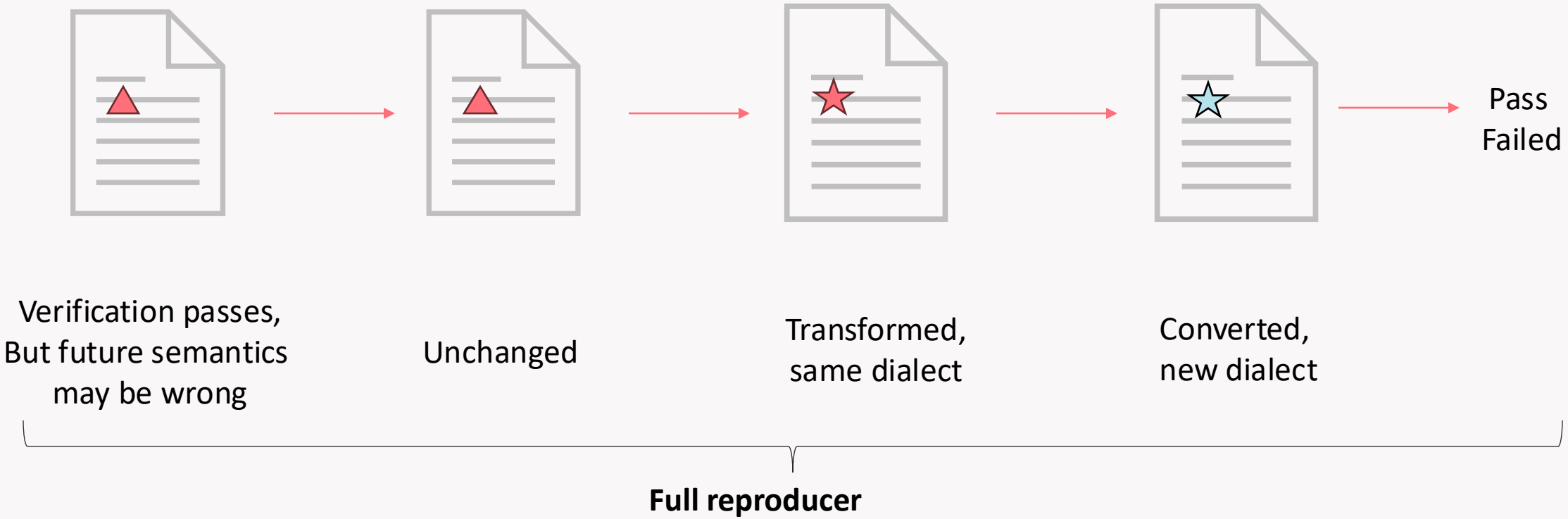
… but something bad may have happened *much earlier*



LOCAL VS FULL REPRODUCER

Local reproducers capture the state of the IR *just before failure…*

… but something bad may have happened *much earlier*



EXAMPLES OUT IN THE WILD

- **Triton**
- Tensorflow
- IREE