Sanitizing MLIR Programs with Runtime Operation Verification

Matthias Springer (NVIDIA)

AsiaLLVM 2025 – Technical Talk – June 10, 2025

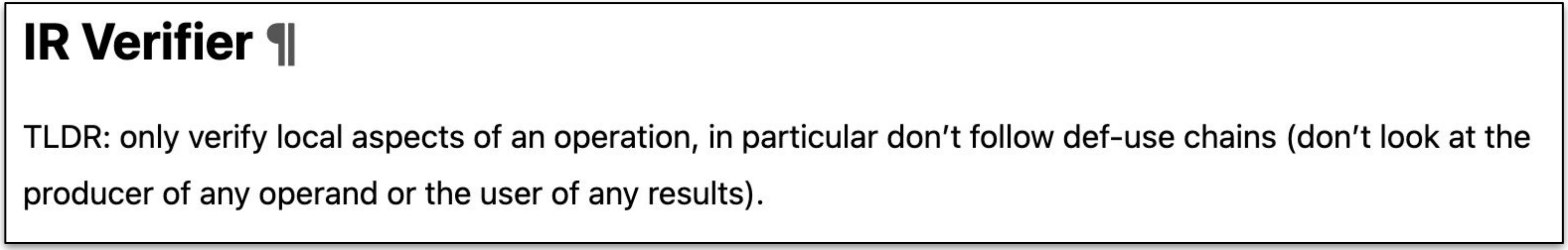
Outline

1. Compile-time Operation Verification
2. Runtime Operation Verification with `RuntimeVerifiableOpInterface`
3. Memory Leak Sanitizer
4. Limitations and Future Work

Operation Verification

Operation Verifier

- Runs between passes or manually.
- A callback defined on operations. Partly auto-generated, partly hand-written.
  - Auto-generated: Operand/result type checking, operation traits (e.g., `AllTypesMatch`)
  - Hand-written: `MyOp::verify()` function
- Should verify only local properties of an operation.



IR Verifier ¶

TLDR: only verify local aspects of an operation, in particular don't follow def-use chains (don't look at the producer of any operand or the user of any results).

- Purpose: Building a **robust compiler**. Also a useful tool for MLIR beginners.

Operation Verifier – Valid Operation

```
%0 = tensor.extract_slice %t [2] [5] [1] : tensor<10xf32> to tensor<5xf32>
```

Operation Verifier – Invalid Operation

```
%0 = tensor.extract_slice %t [8] [5] [1] : tensor<10xf32> to tensor<5xf32>
```

```
test.mlir:2:6: error: slice along dimension 0 runs out-of-bounds: 12 >= 10
%0 = tensor.extract_slice %t [8] [5] [1] : tensor<10xf32> to tensor<5xf32>
     ^
test.mlir:2:6: note: see current operation: %0 = "tensor.extract_slice"(%arg0)
<{operandSegmentSizes = array<i32: 1, 0, 0, 0>, static_offsets = array<i64: 8>,
static_sizes = array<i64: 5>, static_strides = array<i64: 1>}> :
(tensor<10xf32>) -> tensor<5xf32>
```

Operation Verifier – Valid Operation

```
%0 = tensor.extract_slice %t [%offset] [5] [1] : tensor<10xf32> to tensor<5xf32>
```

Operation Verifier – Valid Operation

```
%offset = arith.constant 8 : index
%0 = tensor.extract_slice %t [%offset] [5] [1] : tensor<10xf32> to tensor<5xf32>
```

Operation Verifier – Invalid at Runtime

```
// RUN: mlir-opt %s -generate-runtime-verification -one-shot-bufferize \
// RUN:     -test-cf-assert -convert-to-llvm | mlir-runner

%offset = arith.constant 8 : index
%0 = tensor.extract_slice %t [%offset] [5] [1] : tensor<10xf32> to tensor<5xf32>
```

```
ERROR: Runtime op verification failed

%13 = "tensor.extract_slice"(%arg0, %arg1) <{operandSegmentSizes = array<i32: 1, 1, 0, 0>,
static_offsets = array<i64: -9223372036854775808>, static_sizes = array<i64: 5>,
static_strides = array<i64: 1>}> : (tensor<10xf32>, index) -> tensor<5xf32>

^ extract_slice runs out-of-bounds along dimension 0

Location: loc("test.mlir":22:5)
```

Runtime Operation Verification

Runtime Operation Verification

- `RuntimeVerifiableOpInterface`: An op interface that inserts `cf.assert` operations to check invariants at runtime.
- `GenerateRuntimeVerificationPass`: A pass that triggers runtime verification code generation for each pre-existing op that implements the interface.

Example: `tensor.dim %t, %dim : tensor<*xf32>`

```
struct DimOpInterface
    : public RuntimeVerifiableOpInterface::ExternalModel<DimOpInterface, DimOp> {
  void generateRuntimeVerification(Operation *op, OpBuilder &builder, Location loc) const {
    auto dimOp = cast<DimOp>(op);
    Value rank = builder.create<RankOp>(loc, dimOp.getSource());
    Value zero = builder.create<arith::ConstantIndexOp>(loc, 0);
    Value inBounds1 = builder.createOrFold<arith::CmpIOp>(
      loc, arith::CmpIPredicate::sge, value, zero);
    Value inBounds2 = builder.createOrFold<arith::CmpIOp>(
      loc, arith::CmpIPredicate::slt, value, rank);
    Value inBounds =
      builder.createOrFold<arith::AndIOp>(loc, inBounds1, inBounds2);
    builder.create<cf::AssertOp>(
        loc, inBounds,
        RuntimeVerifiableOpInterface::generateErrorMessage(op, "index is out of bounds"));
  }
};
```

Example: `tensor.dim %t, %dim : tensor<*xf32>`

```
struct DimOpInterface
    : public RuntimeVerifiableOpInterface::ExternalModel<DimOpInterface, DimOp> {
  void generateRuntimeVerification(Operation *op, OpBuilder &builder, Location loc) const {
    auto dimOp = cast<DimOp>(op);
    Value rank = builder.create<RankOp>(loc, dimOp.getSource());
    Value zero = builder.create<arith::ConstantIndexOp>(loc, 0);
    Value inBounds1 = builder.createOrFold<arith::CmpIOp>(
      loc, arith::CmpIPredicate::sge, value, zero);
    Value inBounds2 = builder.createOrFold<arith::CmpIOp>(
      loc, arith::CmpIPredicate::slt, value, rank);
    Value inBounds =
      builder.createOrFold<arith::AndIOp>(loc, inBounds1, inBounds2);
    builder.create<cf::AssertOp>(
        loc, inBounds,
        RuntimeVerifiableOpInterface::generateErrorMessage(op, "index is out of bounds"));
  }
};
```

%dim >= 0

%dim < rank

error message, operation as string, location as string

Lowering of `cf.assert`

- `-convert-cf-to-llvm` / `-convert-to-llvm`
  - Prints error message and calls `abort()` function.
- `-test-cf-assert`
  - For runtime verification integration tests.
  - Does not crash the program, just prints the error message: a single integration test can test multiple invariants.

What to Verify?

Do Verify:

- Conditions that are “undefined behavior” according to the op documentation.
- In practice: Op documentation is often incomplete. Verify properties that the static op verifier would detect if it had enough static information.

Do Not Verify:

- Conditions that lead to “poison” (deferred undefined behavior).
  E.g.: `arith.addi <nsw, nuw>` overflow
- For efficiency reasons: Invariants that are already checked by the static op verifier.

Supported Operations in MLIR

- `memref.assume_alignment`
- `memref.atomic_rmw`: OOB access
- `memref.cast`: shape/offset/stride mismatch
- `memref.copy`: shape mismatch
- `memref.dim`: dimension OOB
- `memref.expand_shape`: invalid result shape
- `memref.generic_atomic_rmw`: OOB access
- `memref.load`: OOB access
- `memref.store`: OOB access
- `memref.subview`: OOB view

- `tensor.cast`: shape mismatch
- `tensor.dim`: dimension OOB
- `tensor.extract`: OOB access
- `tensor.insert`: OOB access
- `tensor.extract_slice`: OOB slice

- linalg structured op: OOB indices computed by `indexing_maps`

Supported Operations in MLIR

- `memref.assume_alignment`
- `memref.atomic_rmw`: OOB access
- `memref.cast`: shape/offset/stride mismatch
- `memref.copy`: shape mismatch
- `memref.dim`: dimension OOB
- `memref.expand_shape`: invalid result shape
- `memref.generic_atomic_rmw`: OOB access
- `memref.load`: OOB access
- `memref.store`: OOB access
- `memref.subview`: OOB view

- `tensor.cast`: shape mismatch
- `tensor.dim`: dimension OOB
- `tensor.extract`: OOB access
- `tensor.insert`: OOB access
- `tensor.extract_slice`: OOB slice

- linalg structured op: OOB indices computed by `indexing_maps`

```
linalg.generic {
    indexing_maps = [#identity1D, #identity1D],
    iterator_types = ["parallel"]}
  ins(%a : tensor<?xf32>) outs(%b : tensor<?xf32>) {
  ^bb0(%arg0: f32, %arg1: f32) :
    /* ... */
} -> tensor<?xf32>
```

with `%arg0 = tensor<5xf32>`, `%arg1 = tensor<4xf32>`

How to add Verification for another Dialect

- Implement `RuntimeVerifiableOpInterface` as an external model.
- Load the external model after loading dialects (see `InitAllDialects.h`).
- Declare the `RuntimeVerifiableOpInterface` as “promised”.
- Add one integration test file per operation. Integration tests run through `mlir-runner`. Add positive + negative tests (like roundtrip and invalid tests). (File naming scheme: `dim-runtime-verification.mlir`)
- Example: https://github.com/llvm/llvm-project/pull/141332

When to Trigger Verification?

- **Early:** Verify before high level information is lost.
- **Multiple times:** Can help finding buggy transformations.
- **Debug build only:** Runtime verification has a significant runtime overhead.

Memory Leak Sanitizer

Memory Leak Sanitizer

- What to detect?
  - **Memory Leak:** `memref.alloc` without matching `memref.dealloc`.
  - **Double Free:** Duplicate `memref.dealloc` for `memref.alloc`.
    Or: `memref.dealloc` for memref that was never allocated.
- Basic idea
  - MLIR runtime library (`RunnerUtils.cpp`) keeps track of allocated (but not yet deallocated) buffers. Stores error messages in `std::unordered_map<void *, std::string>`. If non-empty, static destructor prints all error messages and abort the program with error code.
  - `memref.alloc` runtime verification: call into runtime library to register an allocation.
  - `memref.dealloc` runtime verification: call into runtime library to remove an allocation.
- Prototype on Github:
  https://github.com/llvm/llvm-project/commits/users/matthias-springer/leak_san_mlir/

Test Case

```
// RUN: mlir-opt %s -generate-runtime-verification \
// RUN:     -test-cf-assert \
// RUN:     -convert-to-llvm | \
// RUN: mlir-runner -e main -entry-point-result=void \
// RUN:     -shared-libs=%mlir_runner_utils

func.func @main() {
  %alloc = memref.alloc() : memref<1xf32>
  return
}
```

Output with -DLLVM_USE_SANITIZER="Address"

```
ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x5d3caea610ff in malloc /tmp/final/llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:68:3
    #1 0x724fc8ee7094  (<unknown module>)
    #2 0x5d3cb0ab3f18 in compileAndExecuteVoidFunction((anonymous namespace)::Options&, mlir::Operation*,
llvm::StringRef, (anonymous namespace)::CompileAndExecuteConfig, std::unique_ptr<llvm::TargetMachine,
std::default_delete<llvm::TargetMachine>>)
/home/mspringer/llvm-project/mlir/lib/ExecutionEngine/JitRunner.cpp:239:10
    #3 0x5d3cb0ab058e in mlir::JitRunnerMain(int, char**, mlir::DialectRegistry const&, mlir::JitRunnerConfig)
/home/mspringer/llvm-project/mlir/lib/ExecutionEngine/JitRunner.cpp:397:23
    #4 0x5d3caeaa0a7c in main /home/mspringer/llvm-project/mlir/tools/mlir-runner/mlir-runner.cpp:93:10
    #5 0x724fc8c29d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f) (BuildId:
cd410b710f0f094c6832edd95931006d883af48e)
```

Output with Runtime Operation Verification

```
ERROR: Runtime op verification failed
%1 = "memref.alloc"() <{operandSegmentSizes = array<i32: 0, 0>}> : () -> memref<1xf32>
^ memory leak detected
Location: loc("alloc-runtime-verification.mlir":13:12)
```

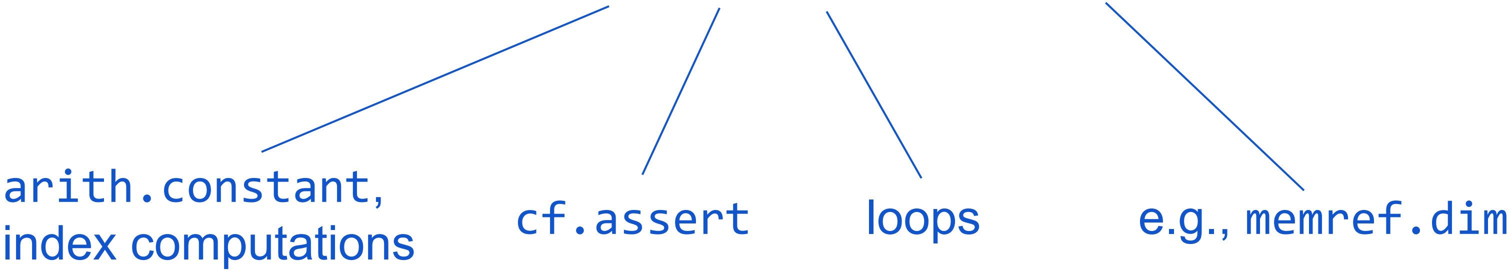
Limitations / Future Work / Open Questions

More Customization Options

- Generate runtime verification only for specific ops/dialects.
- Turn on/off specific runtime checks
  - Different sanitizers: e.g., leak sanitizer, UB sanitizer, …
  - Different kinds of UB.
    E.g., see https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

What kind of IR may the Interface Implementation Emit?

- Problem: `RuntimeVerifiableOpInterface` implementation may build operations that are not lowered by the compilation pipeline.
- Action item: Define a clear contract that describes which operations/dialects may be emitted.
- Dialects being emitted today: `arith`, `cf`, `scf`, the dialect being verified



`arith.constant`,
index computations

`cf.assert`

loops

e.g., `memref.dim`

Poison Semantics

- *Poison semantics:* “Invalid” op does not immediately trigger UB, but poisons the result value. Using a poisoned value either propagates the poison or triggers UB. (E.g., branching based on a poisoned value is UB.)
- Poison semantics is incompatible with runtime op verification. **There is nothing to verify for an op that produces a poisoned result.** Finding out whether a value is poisoned at a later point of time requires a more elaborate lowering strategy. E.g., lower to `!llvm.struct<orig_type, i1>`.
- MLIR seems to be moving towards more poison semantics and less immediate undefined behavior. How useful is runtime op verification?

Better Error Messages

Current error message:

```
ERROR: Runtime op verification failed
^ extract_slice runs out-of-bounds along dimension 0
```

Better error message:

```
ERROR: Runtime op verification failed
^ extract_slice runs out-of-bounds along dimension 0: 12 >= 10
```

Action item: Add format string + SSA values support to `cf.assert` or add a dedicated operation for runtime op verification.

Questions?

`RuntimeVerifiableOpInterface`

`GenerateRuntimeVerificationPass`

Static Operation Verifier

Runtime Operation Verification

Poison Semantics

Undefined Behavior

Local Property of an Operation

Memory Leak

Double Free

Use After Free

ASAN

UBSAN

External Model / Promised Interface

Round-trip Test / Invalid Test

Integration Test

`mlir-runner`

MLIR Runtime Library

Verify Early and Multiple Times

Debug vs Release Build

`cf.assert`

See also:
https://discourse.llvm.org/t/rfc-runtime-op-verification/66776

Appendix

What else can be verified?

- `arith.divsi/divui/remf/remsi/remui`: Check for division by zero.
- `llvm.extractelement/insertelement`: Check for OOB access.
- `vector.extract/insert/gather/scatter`: Check of OOB access.
- `scf.for`: Check for negative step value.
- `vector/affine` load/store ops: Check of OOB access.
- `ptr.from_ptr`: Check for nullptr.
- …