



Injecting SSA Value Facts to Fix Integer Range / Overflow Bugs

Matthias Springer (NVIDIA)

MLIR Meetup – October 29, 2026



Introduction

- MLIR canonicalizations / transformations / lowerings have a surprising amount of incorrect edge cases around integer overflow and poison.
 - [Example](#): Incorrect $(x + 100 \text{ nsw}) + 100 \text{ nsw}$ canonicalization.
 - [Example](#): Signedness not handled correctly during integer narrowing transformation.
 - I found around 20 issues (6 fixed now) with a coding agent as preparation for this talk.
Sometimes required targeted querying: “Can you find such issues in component XYZ?” Formal verification may be able to find many more issues.
- In the absence of integer range information, some transformations cannot be fixed without completely deactivating them.
- This talk: An overview of some bugs / missed optimizations and how to address them by utilizing integer range information.



Case 1: arith.remsi Lowering to llvm.srem

- `llvm.srem(INT_MIN, -1)` is undefined behavior.
- `arith.remsi(INT_MIN, -1)` is defined as 0.
- A correct lowering ([fix](#)) could produce `srem(select(y == -1, 0, x), y)`
- Comparison and select can be omitted if `y != -1` is statically known.
- Alternative: Change the semantics ([fix](#)) of `arith.remsi`.

Incorrect Lowering:

```
func.func @case1(%a : i8, %b : i8) -> i8 {  
    %0 = arith.remsi %a, %b : i8  
    return %0 : i8  
}
```



```
llvm.func @case1(%a : i8, %b : i8) -> i8 {  
    %0 = llvm.srem %a, %b : i8  
    return %0 : i8  
}
```



Case 2: `arith.shrsi` $\rightarrow$ `arith.shrui`

- Canonicalization pattern rewrites `trunci(shrsi(x, 8), nsw)` to `trunci(shrui(x, 8), nsw)` for `i16  $\rightarrow$  i8` truncation ([fix](#)).
- Correct only if `x > 0`. Otherwise, `shrsi` and `shrui` differ.
- Is this even a useful canonicalization?

Incorrect Canonicalization:

```
func.func @case2(%a: i16) -> i8 {  
  %c8 = arith.constant 8 : i16  
  %sh = arith.shrsi %a, %c8 : i16  
  %hi = arith.trunci %sh  
          overflow<nsw> : i16 to i8  
  return %hi : i8  
}
```



```
func.func @case2(%a: i16) -> i8 {  
  %c8 = arith.constant 8 : i16  
  %sh = arith.shrui %a, %c8 : i16  
  %hi = arith.trunci %sh  
          overflow<nsw> : i16 to i8  
  return %hi : i8  
}
```



Case 3: Folding arith.truncf into arith.sitofp

- Incorrect example: $x = 2^{54} + 2^{30} + 1$, round-to-nearest, ties-to-even.
 - $i64 \rightarrow f64 \rightarrow f32$: 0x5a800000
 - $i64 \rightarrow f32$: 0x5a800001
- Folding would be correct if $-2^{53} \leq x \leq 2^{53}$, so that every integer can be represented as f64.

Incorrect Canonicalization:

```
func.func @case3(%x: i64) -> f32 {  
    %wide = arith.sitofp %x : i64 to f64  
    %r = arith.truncf %wide : f64 to f32  
    return %r : f32  
}
```



```
func.func @case3(%x: i64) -> f32 {  
    %r = arith.sitofp %x : i64 to f32  
    return %r : f32  
}
```



Case 4: Missed Optimization in CMPF → CMPI

- CmpFIntToFPConst rewrites `cmpf(oeq, sitofp(x), fp_cst)` to `cmpi(eq, x, int_cst)` if integer width and FP semantics permit.
- Additional cases could be supported based on integer ranges.
- Example: `cmpf(oeq, sitofp(x, i32→f32), 16677216.0)` could be rewritten as `cmpi(eq, x, 16677216)` if $-16677216 \leq x \leq 16677216$.

Existing Conservative Canonicalization:

```
func.func @case4(%x: i32) -> i1 {  
  %c = arith.constant 16677216.0 : f32  
  %fp = arith.sitofp %x : i32 to f32  
  %eq = arith.cmpf oeq, %fp, %c : f32  
  return %eq : i1  
}
```



```
func.func @case4(%arg0: i32) -> i1 {  
  %c = arith.constant 16677216 : i32  
  %0 = arith.cmpi eq, %arg0, %c : i32  
  return %0 : i1  
}
```



Case 5: Inferring no-wrap Flags Based on Integer Range

- nsw/nuw flags can enable more efficient lowerings / transformations.
- Flags can sometimes be inferred.
- E.g. `addi(a, b)` for `i8` where $0 \leq a < 50$ and $0 \leq b < 50$ can be rewritten as `addi(a, b, nsw, nuw)`.

```
func.func @case5(%a: i8, %b: i32) -> i8 {  
  %r = arith.addi %a, %b : i8  
  return %r : i8  
}
```



```
func.func @case5(%a: i8, %b: i8) -> i8 {  
  %r = arith.addi %a, %b <nsw> : i8  
  return %r : i8  
}
```



Case 6: Overflow During ipowi Expansion

```
func.func @case6(%x: i8) -> i8 {  
  %minus_two = arith.constant -2 : i8  
  %result = math.ipowi %x, %minus_two : i8  
  return %result : i8  
}
```



```
func.func @case6(%x: i8) -> i8 {  
  %one = arith.constant 1 : i8  
  %square = arith.muli %x, %x : i8  
  %result = arith.divsi %one, %square : i8  
  return %result : i8  
}
```

- `ipowi(x, -2)` expands to `divsi(1, muli(x, x))`.
- For `x = 127`, the multiplication overflows to 1, so `result = 1` instead of 0.
- For `x = 16`, there is a division by zero.
- Range information could limit transformation to cases without overflow.
(Or directly produce 0 if exponent is negative and $\text{abs}(\text{base}) > 1$.)



Case 7: scf.while to scf.for Uplifting

```
func.func @f(%lb: i8, %ub: i8, %step: i8) {  
  %unused = scf.while (%iv = %lb)  
    : (i8) -> i8 {  
    %cond = arith.cmpi slt, %iv, %ub : i8  
    scf.condition(%cond) %iv : i8  
  } do {  
    ^bb0(%iv: i8):  
    use(%iv)  
    %next = arith.addi %iv, %step : i8  
    scf.yield %next : i8  
  }  
  return  
}
```



```
func.func @f(%lb: i8, %ub: i8, %step: i8) {  
  scf.for %iv = %lb to %ub step %step : i8 {  
    use(%iv)  
  }  
  return  
}
```

- scf.for does not allow negative step size or IV overflow.
- This example scf.while loop cannot be uplifted safely without range info.

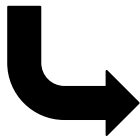


Case 8: IV Overflow During Loop Range Folding

```
func.func @example(%lb: i8, %ub: i8, %step: i8) -> i8 {
  %zero = arith.constant 0 : i8
  %two = arith.constant 2 : i8
  %last = scf.for %i = %lb to %ub step %step iter_args(%previous = %zero) -> (i8) : i8 {
    %twice = arith.muli %i, %two overflow<nsw> : i8
    scf.yield %twice : i8
  }
  return %last : i8
}
```

Incorrect for (0, 64, 1)

- Before returns $63 \cdot 2 = 126$.
- After returns “poison”.
- Range information could make this transformation safe.



```
func.func @example(%lb: i8, %ub: i8, %step: i8) -> i8 {
  %zero = arith.constant 0 : i8
  %two = arith.constant 2 : i8
  %new_lb = arith.muli %lb, %two overflow<nsw> : i8
  %new_ub = arith.muli %ub, %two overflow<nsw> : i8
  %new_step = arith.muli %step, %two overflow<nsw> : i8
  %last = scf.for %i = %new_lb to %new_ub step %new_step
    iter_args(%previous = %zero) -> (i8) : i8 {
      scf.yield %i : i8
    }
  return %last : i8
}
```

poison



Case 9: Missed TMA Optimization in Cuda Tile IR

```
%tv = cuda_tile.make_tensor_view %base, shape = [%n], strides = [1]
      : tile<i64> -> tensor_view<?xf32, strides=[1]>
%pv = cuda_tile.make_partition_view %tv : partition_view<tile=(32), padding_value=zero,
                                          tensor_view<?xf32, strides=[1]>>
%tile, %tok = cuda_tile.load_view_tko weak %pv[%i]
              : partition_view<...>, tile<i64> -> tile<32xf32>, token
```

- Efficient TMA instruction can be chosen only if $i < 2^{26}$.
- Otherwise, compiler must fall back to the LDG instruction.



Design 1: Controllable / Manual Transformation

- No blanket canonicalization.
- Instead, apply transformations only to IR that is known to satisfy the requirements.



Design 2: Utilize Range Information

- Perform an `IntegerRangeAnalysis` and apply transformations only to IR that satisfies the requirements.
- Data flow analyses cannot run as part of the canonicalizer, so these transformations must move into separate passes.
- What about IR where the inputs are known to satisfy the requirements “by construction”?
- What about user-provided IR that satisfies the requirements but it cannot be proven by analyzing the IR? (E.g. user knows that an input value always has a certain range.)



Design 2.1: Assert Style

```
// An integer that is strictly positive.  
arith.assert %x, gt<0> : i32
```

- Assumed property holds for %x.
- `arith.assert` is not speculatable and may have side effects (trigger program abort). Therefore, it is not pure.
- Does not compose well with a sparse dataflow analysis (?)



Design 2.2: Assume Style

```
// An integer that is strictly positive.
```

```
%r = arith.assume %x, gt<0> : i32
```

- Assumed property holds only for %r.
- `arith.assume` is speculatable and has no effects, therefore, pure.
- Composes well with sparse data flow analyses.
- Design is similar to `memref.assume_alignment` and `cuda_tile.assume`.



Design 2.3: Assume Style with SSA Constraints

```
// An integer that is greater than %y.
```

```
%r1 = arith.assume %x, gt<%y> : i32
```

```
// An integer that is greater than %a + %b.
```

```
%t = arith.addi %a, %b : i32
```

```
%r2 = arith.assume %x, gt<%t> : i32
```

- Assumed property holds only for %r.
- `arith.assume` is speculatable and has no effects, therefore, pure.
- Could feed into `ValueBoundsConstraintSet` (heavy weight).



What's Next?

- Comprehensive correctness analysis of integer / float arithmetics in MLIR with a combination of coding agents, fuzzers and formal verification.
- Some transformations can be fixed directly by dropping overflow flags etc.
- Some transformations cannot be fixed without integer range information.
- Add an `arith.assume` operation?