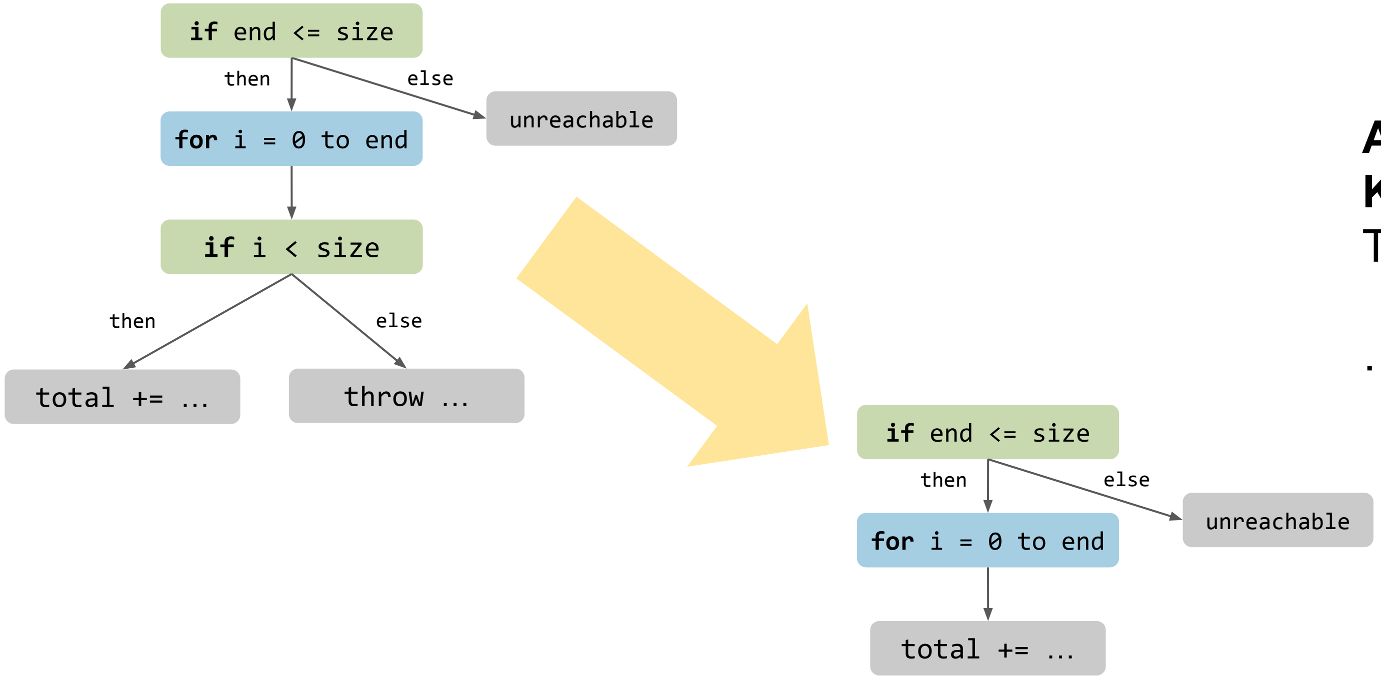
Precise Polyhedral Analyses for MLIR using the FPL Presburger Library



Arjun Pitchanathan
Kunwar Grover
Tobias Grosser

...and many more

Growing interest in MLIR's polyhedral functionality

[mlir][Affine] Add support for multi-store producer fusion

This patch adds support for producer-consumer fusion scenarios with multiple producer stores to the AffineLoopFusion pass. The patch introduces some changes to the producer-consumer algorithm, including:

* For a given consumer loop, producer-consumer fusion iterates over its producer candidates until a fixed point is reached.

* Producer candidates are gathered beforehand for each iteration of the consumer loop and visited in reverse program order (not strictly guaranteed) to maximize the number of loops fused per iteration.

In general, these changes were needed to simplify the multi-store producer support and remove some of the workarounds that were introduced in the past to support more fusion cases under the single-store producer limitation.

This patch also preserves the existing functionality of AffineLoopFusion with one minor change in behavior. Producer-consumer fusion didn't fuse scenarios with escaping memrefs and multiple outgoing edges (from a single store). Multi-store producer scenarios will usually (always?) have multiple outgoing edges so we couldn't fuse any with escaping memrefs, which would greatly limit the applicability of this new feature. Therefore, the patch enables fusion for these scenarios. Please, see modified tests for specific details.

Reviewed By: andydavis1, bondhugula

Differential Revision: https://reviews.llvm.org/D92876

main
llvmorg-15-init ··· llvmorg-12.0.0-rc1

dcaballe committed on 25 Jan 2021

**[Analysis] Add a DependenceAnalysis for checking memory accesses. (#1845
)**

This is currently a simple class that traverses pairs of Affine memory access operations and uses the upstream `checkMemrefAccessDependence` function. The results are stored in a convenient data structure that can be queried to inform scheduling decisions. A test pass is added, which outputs the results as attributes for verification.

main (#1845)
sifive/0/7/0 ··· pycde-0.0.1

mikeurbach committed on Sep 24, 2021 Verified

Eliminating redundant checks

```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    // data.at(i) internally checks if i < data.size().

    total += data.at(i);
  }
  return total;
}
```

Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM.* 2021 LLVM Dev Mtg.

Eliminating redundant checks

```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    if (i >= data.size())
      throw std::out_of_range("");
    total += data[i]; // operator[] has no bounds check.
  }
  return total;
}
```

Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM*. 2021 LLVM Dev Mtg.

Eliminating redundant checks

```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    if (i >= data.size())
      throw std::out_of_range("");
    total += data[i];
  }
  return total;
}
```

Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM*. 2021 LLVM Dev Mtg.

Eliminating redundant checks

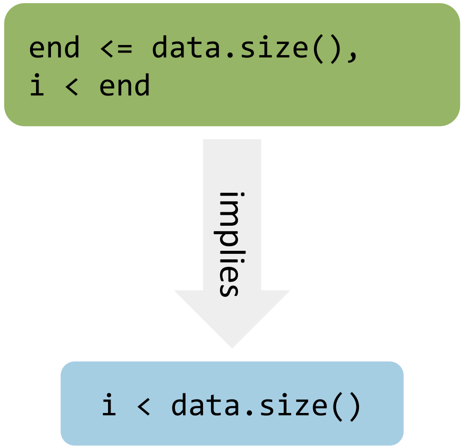
```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    if (i >= data.size())
      throw std::out_of_range("");
    total += data[i];
  }
  return total;
}
```

```
end <= data.size()
```

Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM*. 2021 LLVM Dev Mtg.

Eliminating redundant checks

```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    if (i >= data.size())
      throw std::out_of_range("");
    total += data[i];
  }
  return total;
}
```



Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM.* 2021 LLVM Dev Mtg.

Eliminating redundant checks

```
uint64_t foo(std::vector<uint64_t> &data, size_t end) {
  assert(end <= data.size());
  uint64_t total = 0;
  for (size_t i = 0; i < end; ++i) {
    if (i >= data.size())
      throw std::out_of_range("");
    total += data[i];
  }
  return total;
}
```

```
end <= data.size(),
i < end
```

implies

```
i < data.size()
```

Florian Hahn. *A New Approach to Removing Redundant Conditions in LLVM*. 2021 LLVM Dev Mtg.

Beyond Affine: Subview Fusion

```
%0 = memref.alloc() : memref<3x512xbf16, 1>
%1 = memref.subview %0[0, 0]   [3, 256] [1, 1] : ...
%2 = memref.subview %0[0, 256] [3, 256] [1, 1] : ...
// write to %1
// write to %2
```

Analyzing Partial Writes

MLIR

manbearian Nov '21

I'm looking at doing some analysis to detect when a series of partial writes combines to be equal to a larger write. Is there any existing technology in MLIR i can leverage for this?

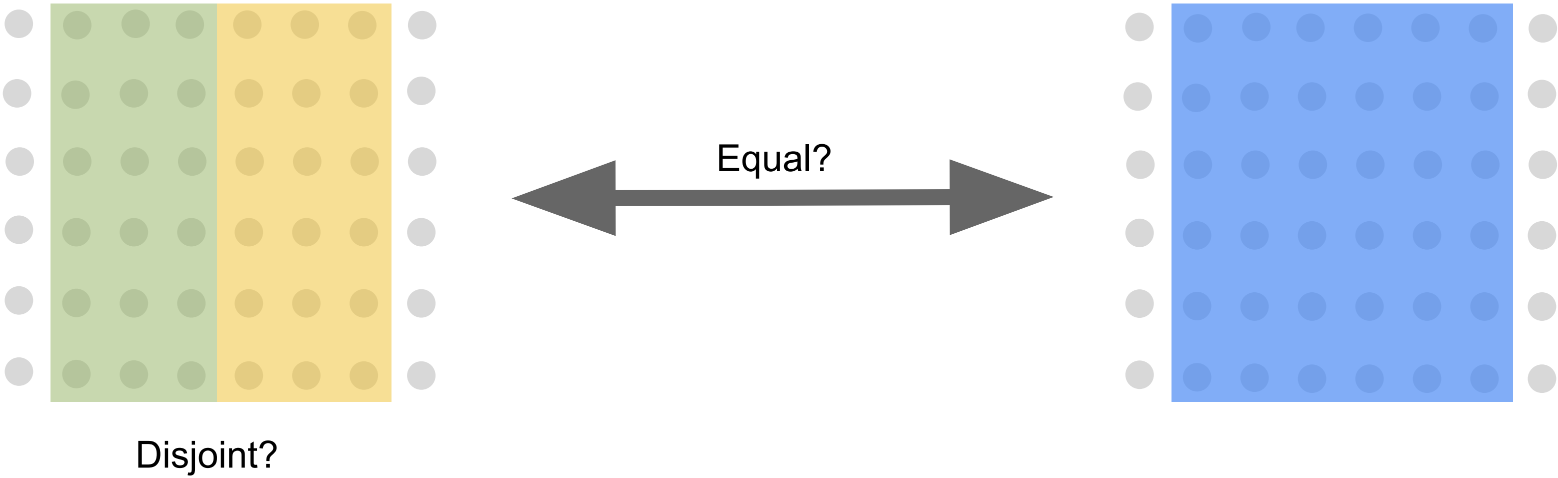
Example:

```
%0 = memref.alloc() : memref<1x3x512x256xbf16, 1>
%1 = memref.subview %0[0, 0, 0, 0] [1, 3, 256, 256] [1, 1, 1, 1] : memref<1x
%2 = memref.subview %0[0, 0, 256, 0] [1, 3, 256, 256] [1, 1, 1, 1] : memref<
// write to %1
// write to %2
```

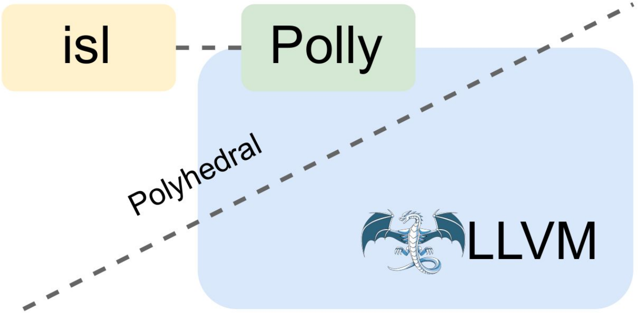
In this example, I'm looking to be able to detect that the two subviews combine to be totally overlapping with the original tensor.

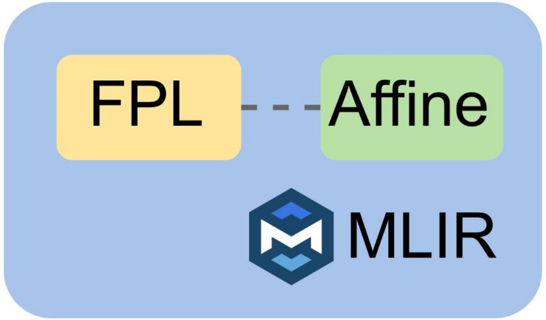
This analysis would need to detect both a sequence of writes as well as a subview that is created dynamically within a loop.

Thanks,
ian



Polyhedral infrastructure: LLVM/Polly vs FPL/MLIR

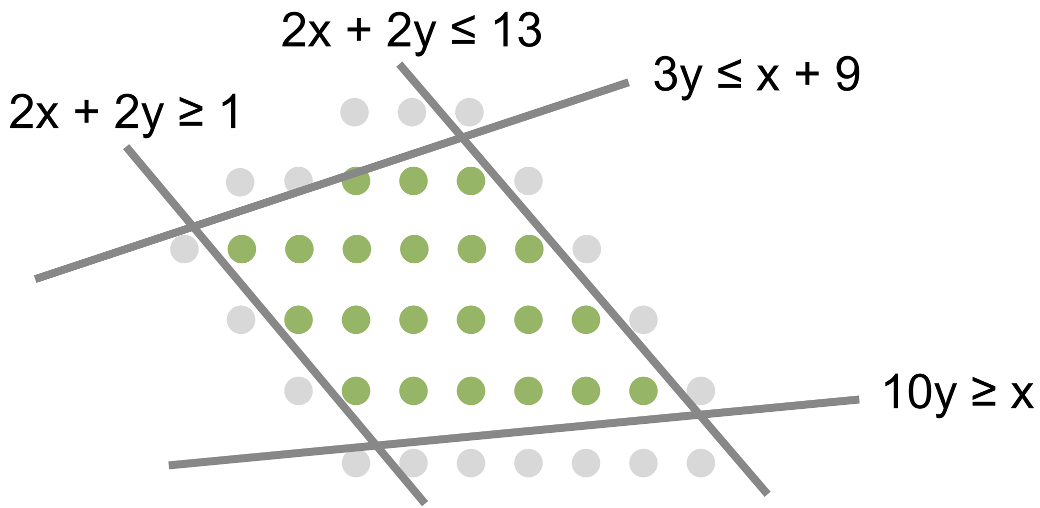




Built from the ground-up for LLVM/MLIR

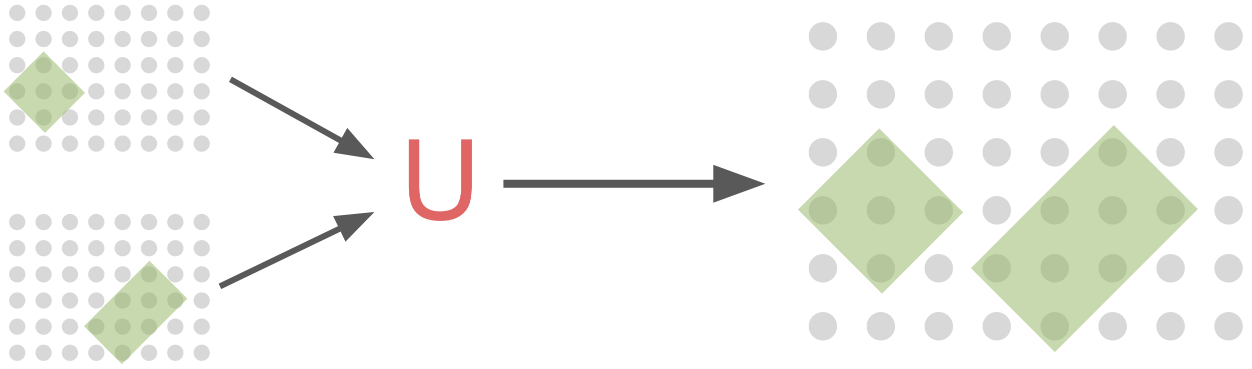
Conversion to & from MLIR IR constructs

The basic building blocks: Integer Polyhedra



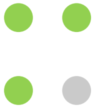
$\{(x, y) \in \mathbf{Z}^2 : 2x + 2y \geq 1 \land 2x + 2y \leq 13 \land 3y \leq x + 9 \land 10y \geq x\}$

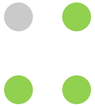
Presburger Sets: Unions of Integer Polyhedra

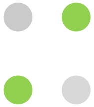


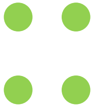
$\{(x, y) \in \mathbf{Z}^2 : (0 \leq x - y \leq 2 \ \wedge \ 2 \leq x + y \leq 4) \vee (-4 \leq x - y \leq -2 \ \wedge \ 4 \leq x + y \leq 8)\}$

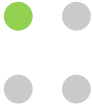
Operations on Integer Sets

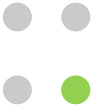
A 

B 

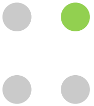
A intersect B 

A union B 

A subtract B 

complement A 

is A empty false

sample A 

coalesce A

Affine Dialect

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      ...

  }
}
```

Affine loop bounds

Affine conditions

Which `i` values get executed?

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

Kinds of IDs in `IntegerPolyhedrons`

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

Set Dimensions

Kinds of IDs in `IntegerPolyhedrons`

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

Set Dimensions

Symbols

Kinds of IDs in `IntegerPolyhedrons`

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

Set Dimensions

Symbols

Locals

PresburgerSpaces

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

PresburgerSpace

Set Dimensions | Symbols | Locals

PresburgerSpaces: `mlir::Value` associations

```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

PresburgerSpace

Set Dimensions | Symbols | Locals

PresburgerSpaces: `merging`

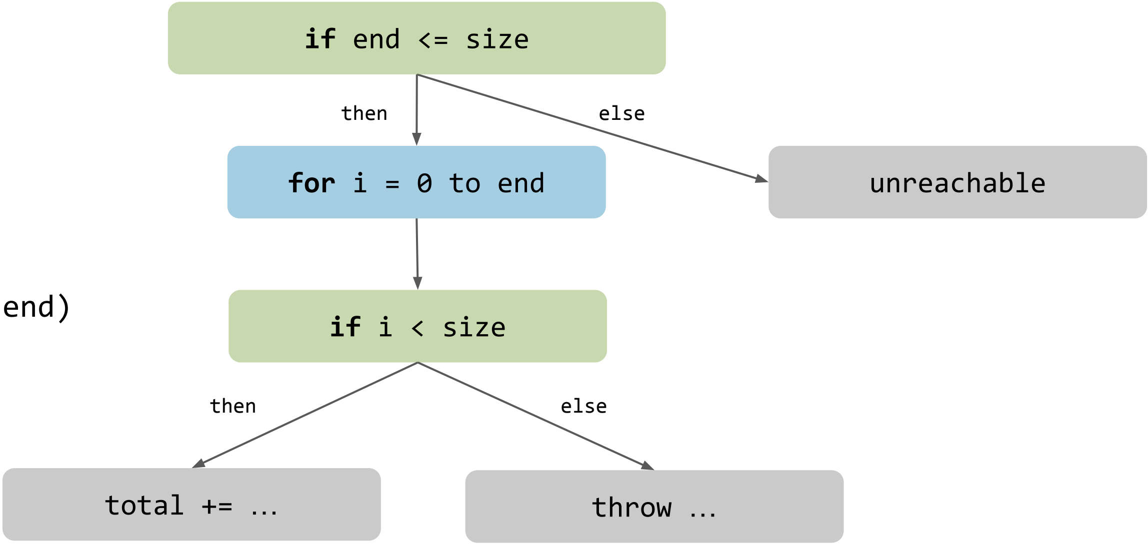
```
affine.for %i = 2 * %S + 4 to 3 * %T + 8 step 2
{
  affine.if (%i >= 0, %i < %N) {

      (i)[S, T, N] : (i == 2*(i floordiv 2),
                      2*S + 4 <= i <= 3*T + 8,
                      0 <= i < N)

  }
}
```

PresburgerSpace

Set Dimensions | Symbols | Locals

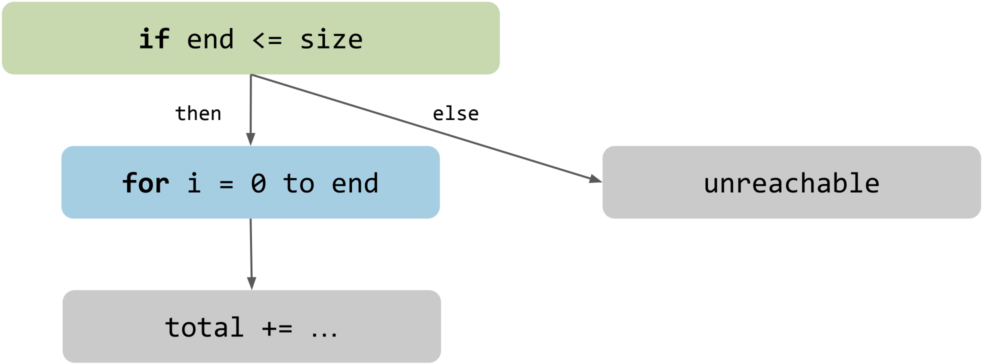
Simplifying Affine Ifs

```
()[] : ()

()[end, size] : (end <= size)

(i)[end, size] : (end <= size, 0 <= i < end)
```



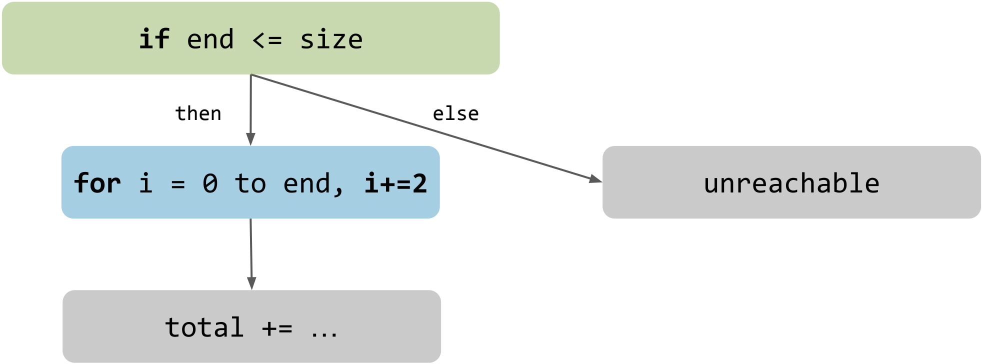
Simplifying Affine Ifs

```
()[] : ()

()[end, size] : (end <= size)

(i)[end, size] : (end <= size, 0 <= i < end)
```



Simplifying Affine Ifs

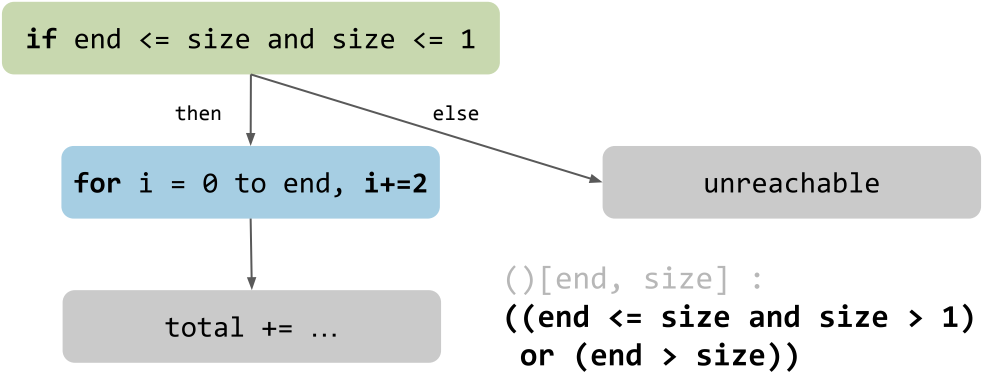
```
       ()[] : ()

()[end, size] : (end <= size)

(i)[end, size] : (end <= size, 0 <= i < end
                  i == 2*(i floordiv 2)     )
```



Full support for strides

Simplifying Affine Ifs

```
     ()[] : ()

()[end, size] : (end <= size)

(i)[end, size] : (end <= size, 0 <= i < end
                  i == 2*(i floordiv 2)    )
```




```
()[end, size] :
((end <= size and size > 1)
 or (end > size))
```

Full support for strides

Full support for analyzing the **else** branch

Code Walk: Simplifying Affine Ifs

Converting IR to Sets

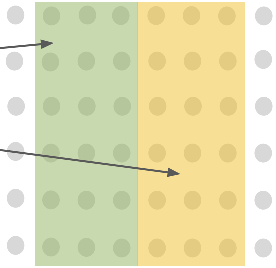
```
void FlatAffineValueConstraints::addAffineIfOpDomain(AffineIfOp ifOp) {
  // Create the base constraints from the integer set attached to ifOp.
  FlatAffineValueConstraints cst(ifOp.getIntegerSet());

  // Bind ids in the constraints to ifOp operands.
  SmallVector<Value, 4> operands = ifOp.getOperands();
  cst.setValues(0, cst.getNumDimAndSymbolIds(), operands);

  // Merge the constraints from ifOp to the current domain. We need first merge
  // and align the IDs from both constraints, and then append the constraints
  // from the ifOp into the current one.
  mergeIds(cst);
  intersectInPlace(cst);
}
```

simplifyGivenHolds

```
void IntegerPolyhedron::simplifyGivenHolds(const IntegerPolyhedron &cst) {
  IntegerPolyhedron mergedCst = cst;
  this->mergeIds(mergedCst);
  Simplex simplex(mergedCst);

  // These loop bounds change during the loop!
  for (unsigned i = 0; i < getNumInequalities();) {
     if (simplex.isRedundantInequality(getInequality(i)))
       removeInequality(i);
     else
       ++i;
  }
}
```

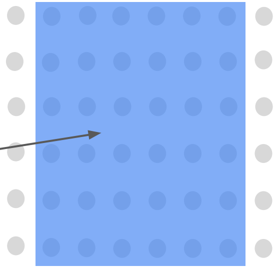
Subview Fusion in MLIR

```
%mem  = memref.alloc() : memref<3x512xbf16, 1>
%sub1 = memref.subview %mem[0, 0]   [3, 256] [1, 1] : ...
%sub2 = memref.subview %mem[0, 256] [3, 256] [1, 1] : ...

// Fill operation on subviews
linalg.fill ins(%val) outs(%sub1)
linalg.fill ins(%val) outs(%sub2)
```



```
(x, y) : (0 <= x < 3, 0 <= y < 256)
                   U
         (0 <= x < 3, 256 <= y < 512)
```

```
// Fill operation on main memref
linalg.fill ins(%val) outs(%mem)
```

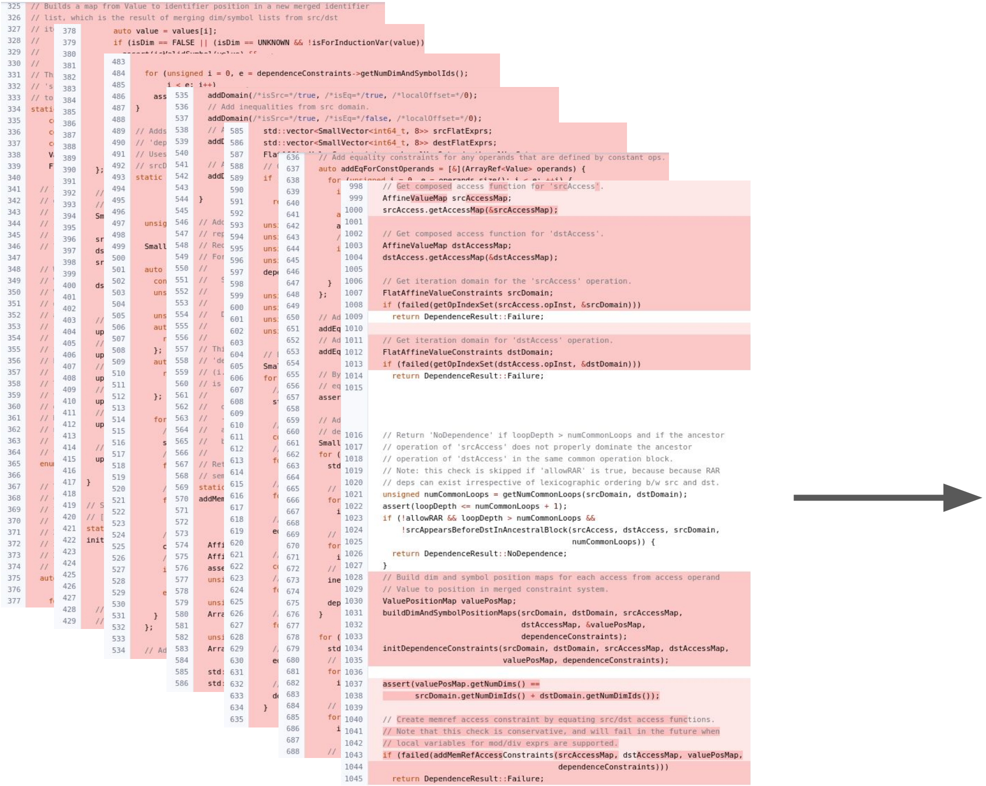


```
(x, y) : (0 <= x < 3, 0 <= y < 512)
```

Code Walk: Subview Fusion

https://reviews.llvm.org/D110563

Simplified Dependence Analysis with FPL

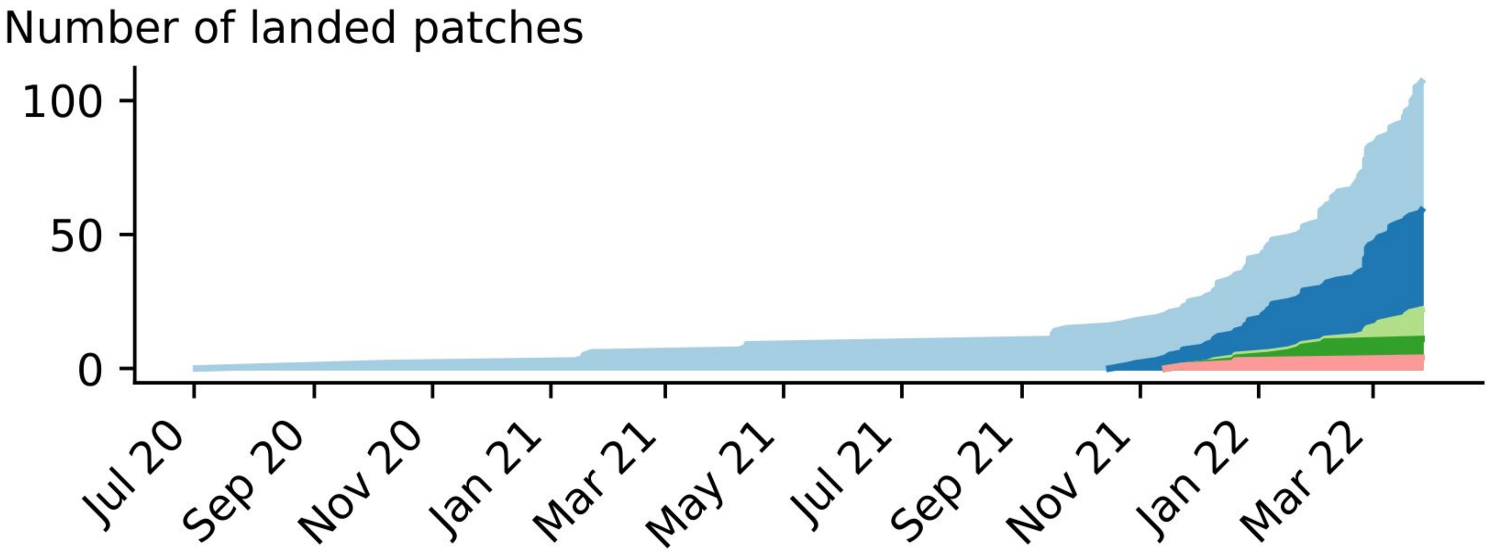


```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;
if (failed(srcAccess.getAccessRelation(srcRel)))

  return DependenceResult::Failure;
if (failed(dstAccess.getAccessRelation(dstRel)))

  return DependenceResult::Failure;

FlatAffineValueConstraints srcDomain = srcRel.getDomainSet();
FlatAffineValueConstraints dstDomain = dstRel.getDomainSet();

// Return 'NoDependence' if loopDepth > numCommonLoops and if the ancestor
// operation of 'srcAccess' does not properly dominate the ancestor
// operation of 'dstAccess' in the same common operation block.
// Note: this check is skipped if 'allowRAR' is true, because because RAR
// deps can exist irrespective of lexicographic ordering b/w src and dst.
unsigned numCommonLoops = getNumCommonLoops(srcDomain, dstDomain);
assert(loopDepth <= numCommonLoops + 1);
if (!allowRAR && loopDepth > numCommonLoops &&
    !srcAppearsBeforeDstInAncestralBlock(srcAccess, dstAccess, srcDomain,
                                         numCommonLoops)) {
  return DependenceResult::NoDependence;
}

// Compute the dependence relation by composing `srcRel` with the inverse of
// `dstRel`. Doing this builds a relation between iteration domain of
// `srcAccess` to the iteration domain of `dstAccess` which access the same
// memory locations.
dstRel.inverse();
dstRel.compose(srcRel);
*dependenceConstraints = dstRel;
```

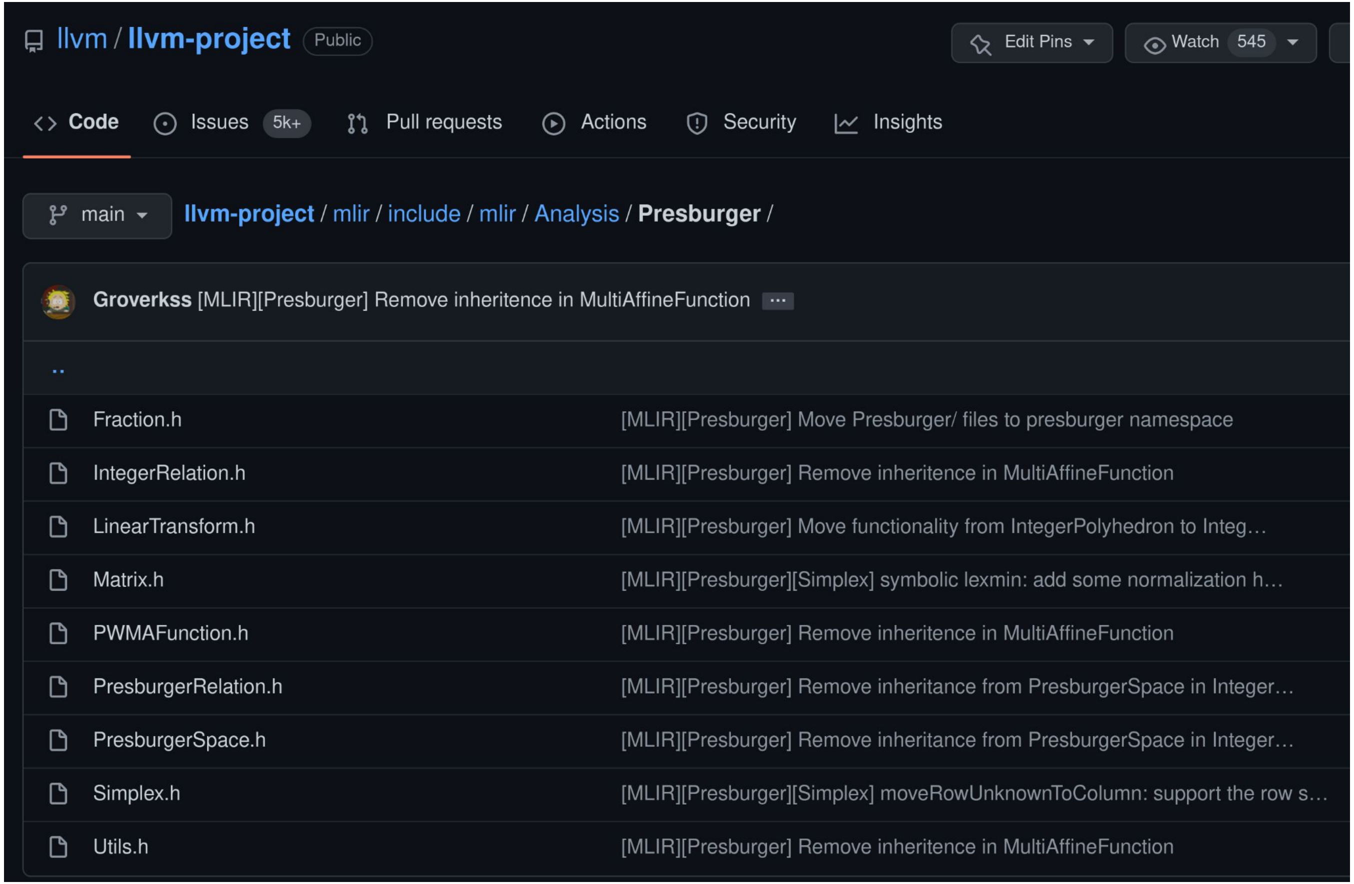
FPL: A Complete Set of Operations

	union	intersect	subtract	equality	emptiness	lexmin
MLIR (before)	no	no local ID support	no	no	inexact heuristic	no
MLIR’s FPL	yes	yes	yes	yes	yes	yes

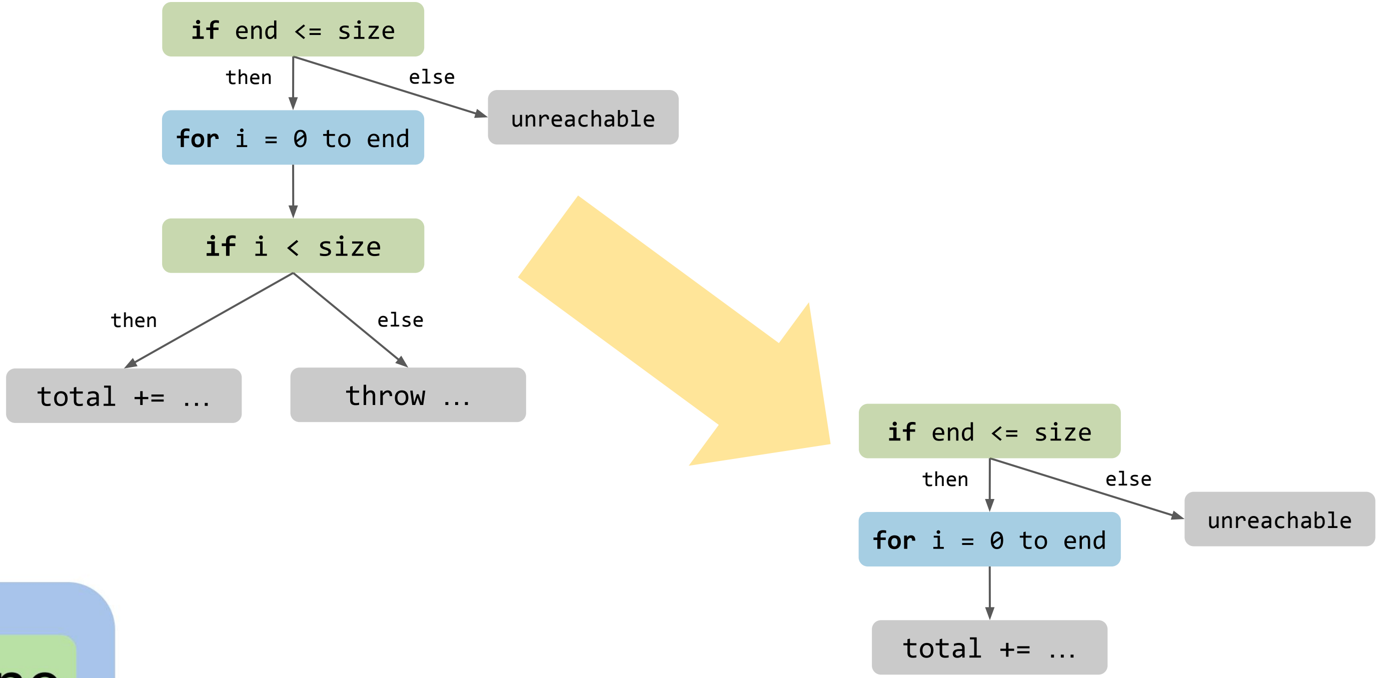
FPL’s Growing Developer Community

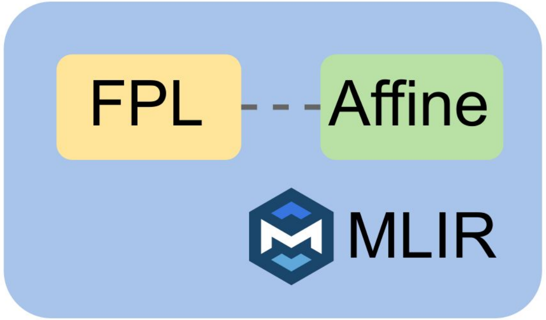


Available in the monorepo today!



Conclusion





grosser.science/FPL