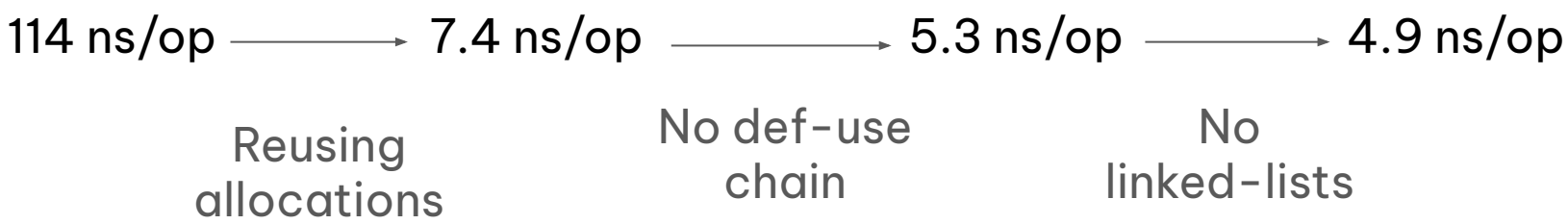
Specializing MLIR Data Structures: an Experiment



Compilation time matters

- Faster development cycle

Compilation time matters

- Faster development cycle
- Faster CI times

Compilation time matters

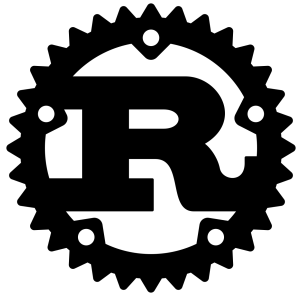
- Faster development cycle
- Faster CI times
- Happiness

Compilation time matters

- Faster development cycle
- Faster CI times
- Happiness

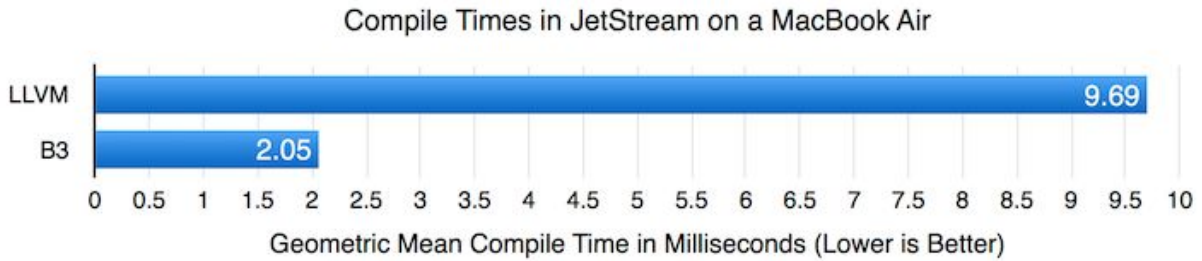
MLIR is known to be slow

MLIR is known to be slow



MLIR is known to be slow

MLIR is known to be slow



Compile-time is a trade-off

Compile-time is a trade-off

What are you willing to pay to get faster compile-time?

First trade-off : Extensibility vs. Performance

Machine learning

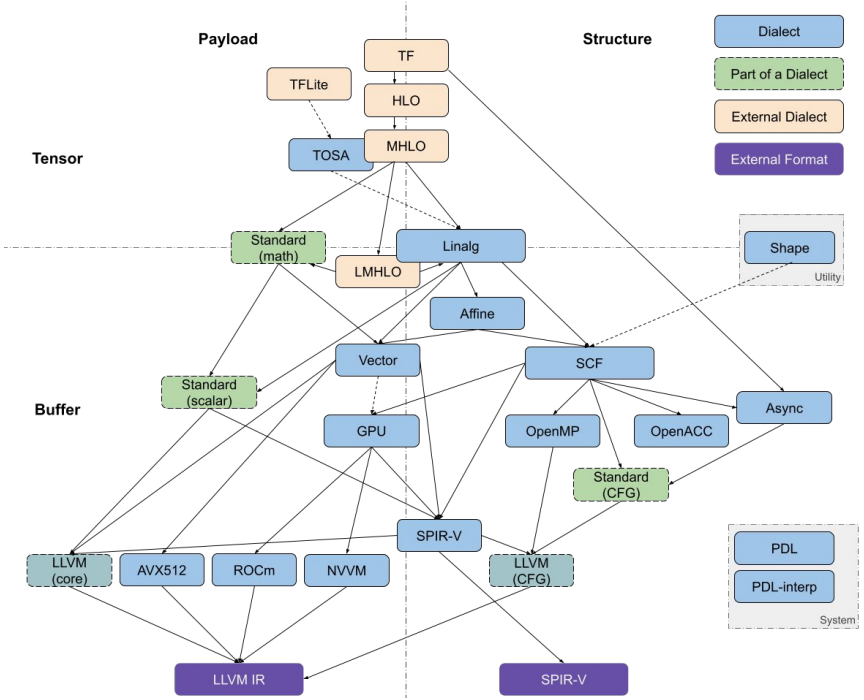
Hardware design

Fully Homomorphic encryption

General-purpose languages

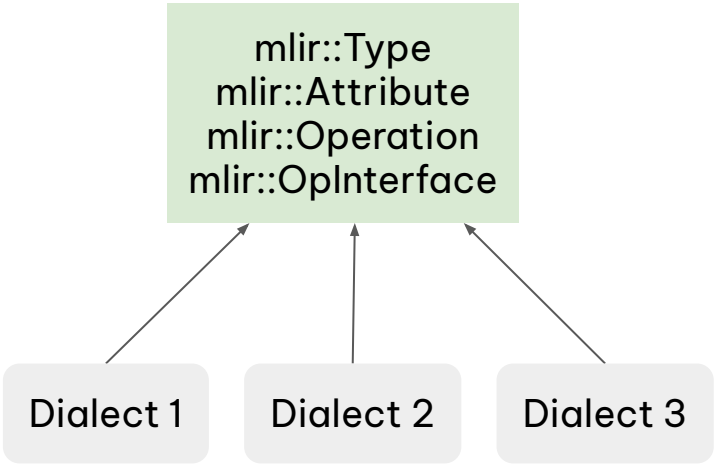
Computer Graphics

All abstractions using a single IR



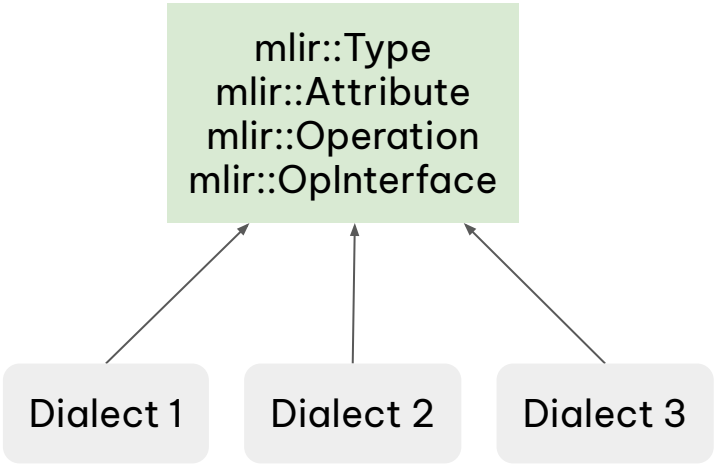
By Alex Zinenko

Built on an extensible foundation



Built on an extensible foundation

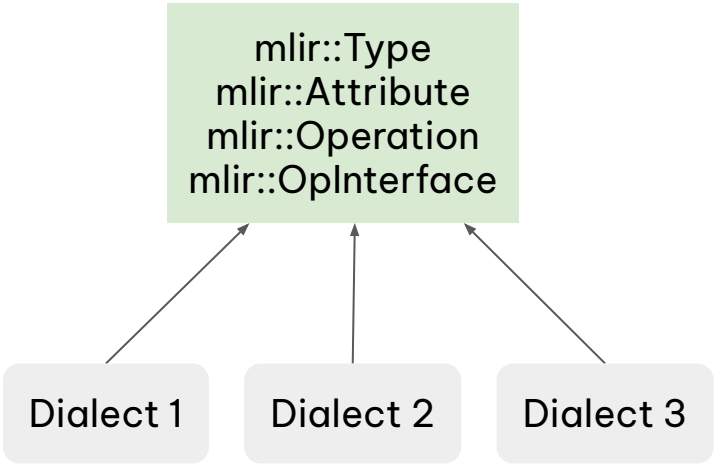
```
%a, %b = arith.addi_extended %x, %y : i32
```



Built on an extensible foundation

```
%a, %b = arith.addi_extended %x, %y : i32
```

```
func.call @foo(%a, %b, %c, %d, %e) : ...
```

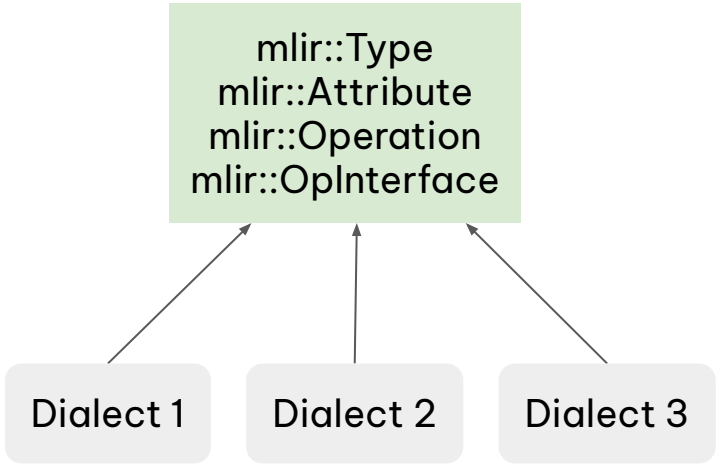


Built on an extensible foundation

```
%a, %b = arith.addi_extended %x, %y : i32
```

```
func.call @foo(%a, %b, %c, %d, %e) : ...
```

```
cf.br ^bb0(%x : i32)
```



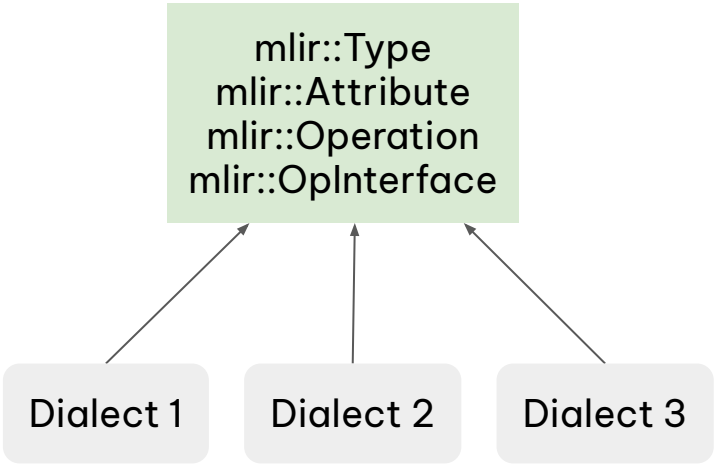
Built on an extensible foundation

```
%a, %b = arith.addi_extended %x, %y : i32
```

```
func.call @foo(%a, %b, %c, %d, %e) : ...
```

```
cf.br ^bb0(%x : i32)
```

```
scf.for %arg3 = %c0 to %0 step %c1 {
   ...
}
```



Built on an extensible foundation

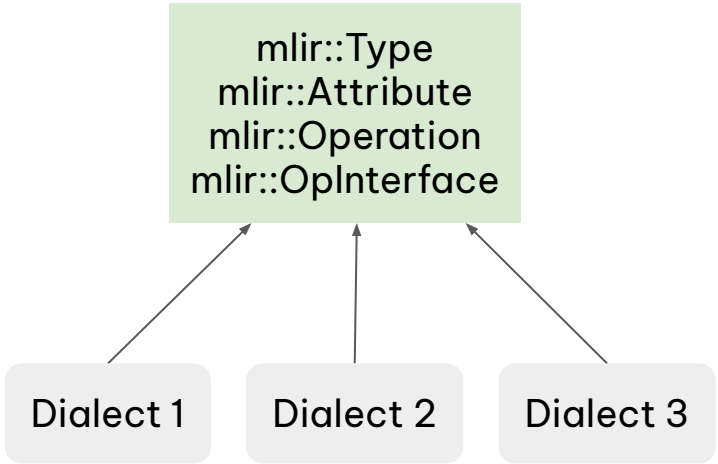
```
%a, %b = arith.addi_extended %x, %y : i32
```

```
func.call @foo(%a, %b, %c, %d, %e) : ...
```

```
cf.br ^bb0(%x : i32)
```

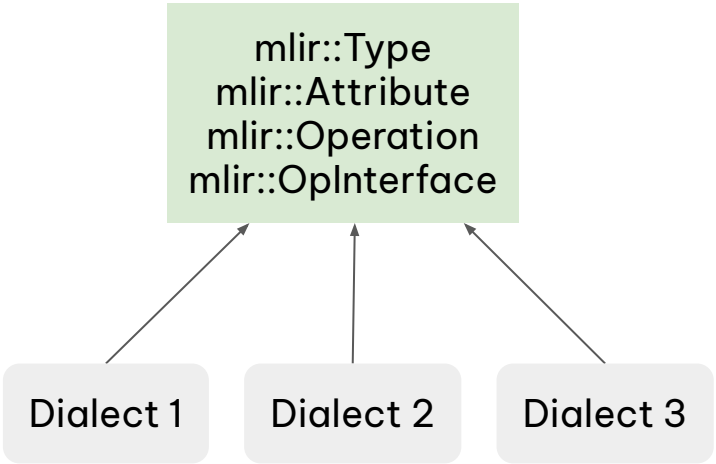
```
scf.for %arg3 = %c0 to %0 step %c1 {
   ...
}
```

```
%x, %y = test.test(%a, %b, %c) ^bb0, ^bb1 {
  ...
}, {
  ...
}
```



Built on an extensible foundation

```
dyn_cast<MemoryEffectOpInterface>(op);
```



First trade-off : Extensibility vs. Performance

- Operation memory size
- Interfaces and traits
- Checking operation Opcode
- Attribute dictionary

Second trade-off: Usability vs. Performance

Second trade-off: Usability vs. Performance

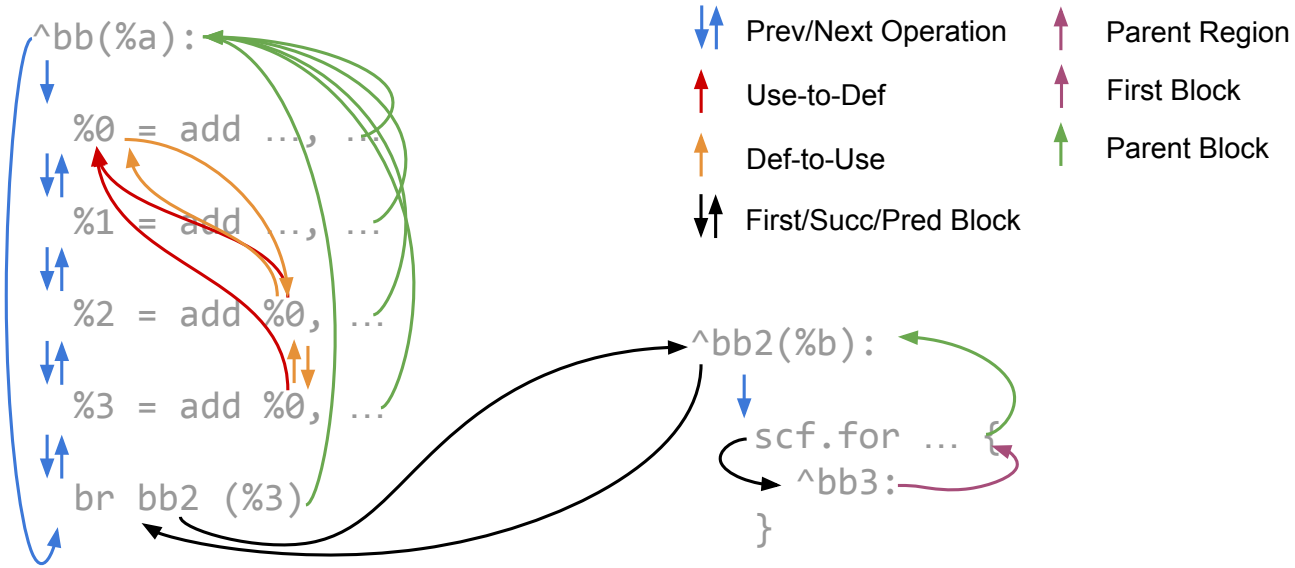
Operation	Complexity
Creating an operation *op*	$O(\|op\|)$
Inserting an operation	$O(1)$
Detaching an operation	$O(1)$
Erasing an operation *op*	$O(\|op_{rec}\| + \|op_{rec}.uses\|)$
Replacing an operand/block operand	$O(1)$
Replacing a value v with another value	$O(\|v.uses\|)$
Creating a block *block*	$O(\|block\|)$
Inserting a block	$O(1)$
Detaching a block	$O(1)$
Erasing a block	$O(\|block_{rec}\| + \|block_{rec}.uses\|)$
Creating a region	$O(1)$
Erasing a region	$O(\|region_{rec}\| + \|region_{rec}.uses\|)$

A pointer-heavy data structure...

```
^bb(%a):

  %0 = add ..., ...

  %1 = add ..., ...

  %2 = add %0, ...

  %3 = add %0, ...

  br bb2 (%3)
```

```
^bb2(%b):

   scf.for ... {
     ^bb3:
   }
```

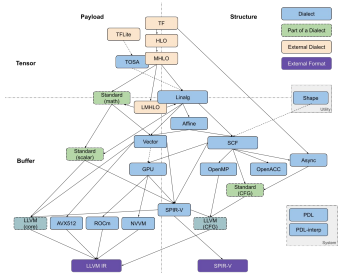
A pointer-heavy data structure...



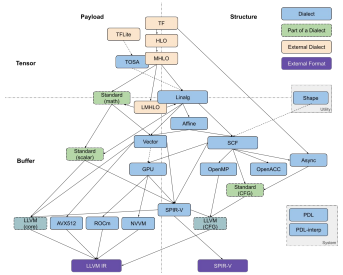
...resulting in costly bookkeeping

```
setOperand:
ldr    x8, [x0, #0x48]
mov    w9, w1
add    x8, x8, x9, lsl #5
ldr    x10, [x8, #0x8]
cbz    x10, 0x5f0
ldr    x9, [x8]
str    x9, [x10]
cbz    x9, 0x5f0
ldr    x10, [x8, #0x8]
str    x10, [x9, #0x8]
str    x2, [x8, #0x18]
ldr    x9, [x2]
stp    x9, x2, [x8]
cbz    x9, 0x604
str    x8, [x9, #0x8]
str    x8, [x2]
ret
```

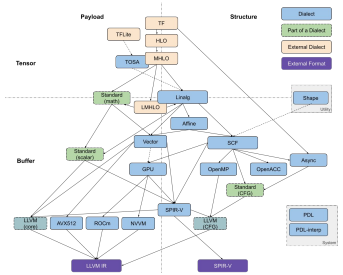
Could we get the best of both worlds?



Could we get the best of both worlds?

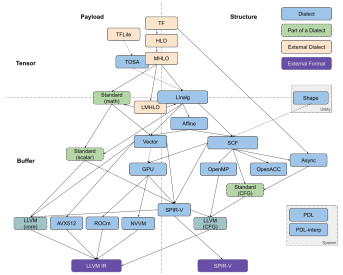


Could we get the best of both worlds?



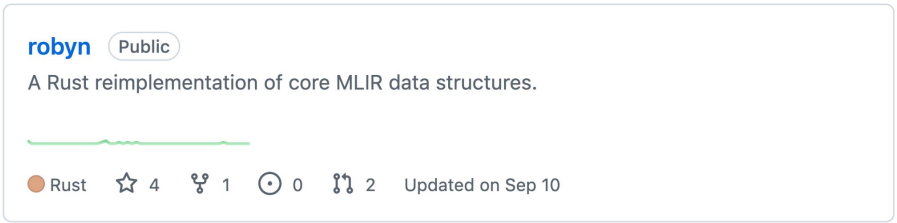
Operation	Complexity
Creating an operation op	$O(\|op\|)$
Inserting an operation	$O(1)$
Detaching an operation	$O(1)$
Erasing an operation op	$O(\|op_{rec}\| + \|op_{rec}.uses\|)$
Replacing an operand/block operand	$O(1)$
Replacing a value v with another value	$O(\|v.uses\|)$
Creating a block $block$	$O(\|block\|)$
Inserting a block	$O(1)$
Detaching a block	$O(1)$
Erasing a block	$O(\|block_{rec}\| + \|block_{rec}.uses\|)$
Creating a region	$O(1)$
Erasing a region	$O(\|region_{rec}\| + \|region_{rec}.uses\|)$

Could we get the best of both worlds?



Operation	Complexity
Creating an operation *op*	$O(\|op\|)$
Inserting an operation	$O(1)$
Detaching an operation	$O(1)$
Erasing an operation *op*	$O(\|op_{rec}\| + \|op_{rec}.uses\|)$
Replacing an operand/block operand	$O(1)$
Replacing a value v with another value	$O(\|v.uses\|)$
Creating a block *block*	$O(\|block\|)$
Inserting a block	$O(1)$
Detaching a block	$O(1)$
Erasing a block	$O(\|block_{rec}\| + \|block_{rec}.uses\|)$
Creating a region	$O(1)$
Erasing a region	$O(\|region_{rec}\| + \|region_{rec}.uses\|)$

What if MLIR data structures were an interface?

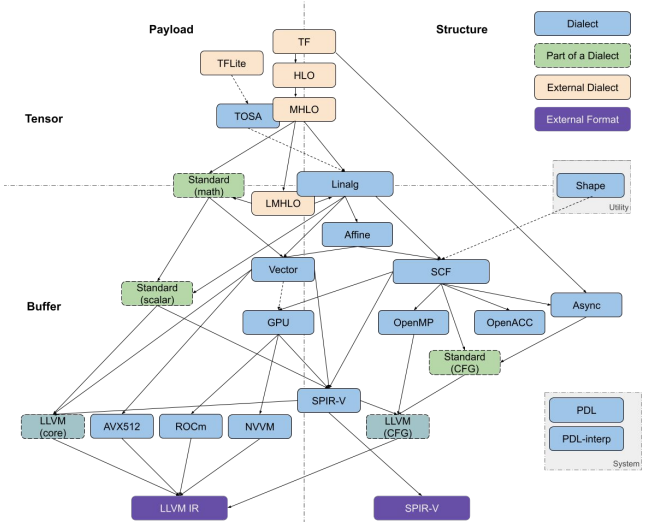



```rust
pub trait Context: Sized {
    type Operation<'a, 'b>: Operation<'a, 'b, Self>;
    ...
}
```

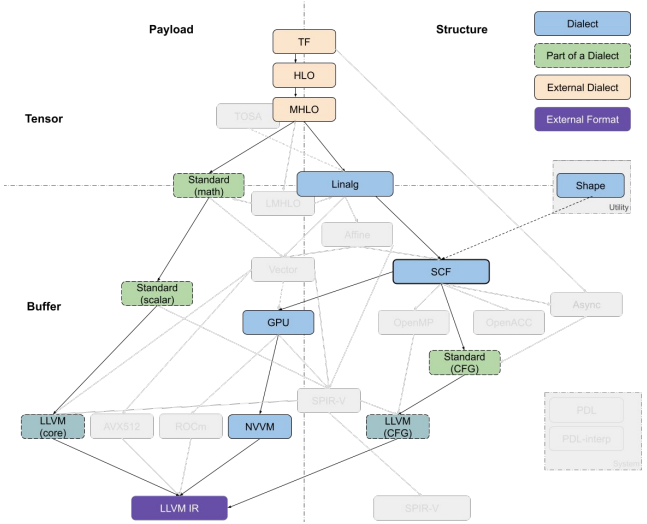
This talk:

What are the performance opportunities of specializing MLIR data structures?

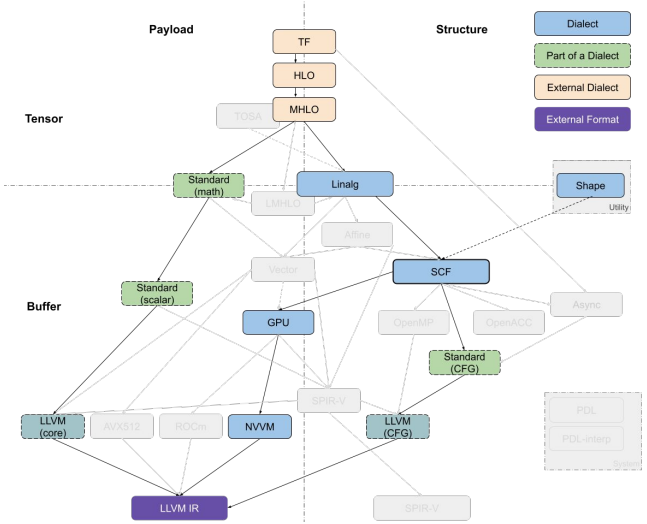
Could we specialize MLIR to a dialect set?



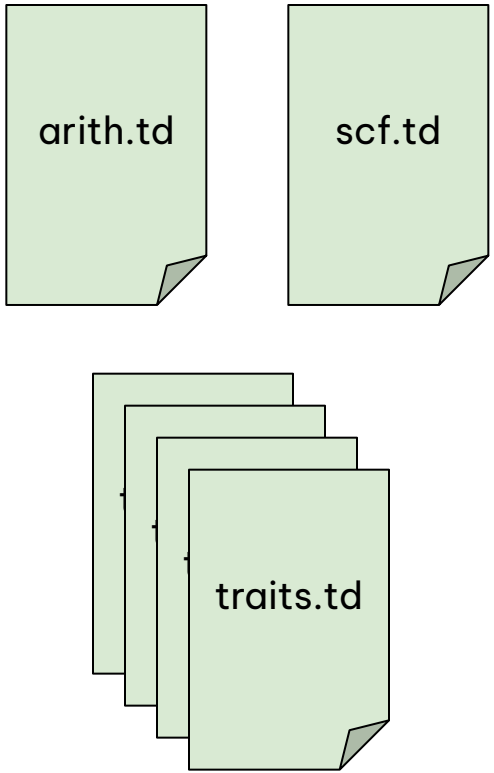
Could we specialize MLIR to a dialect set?



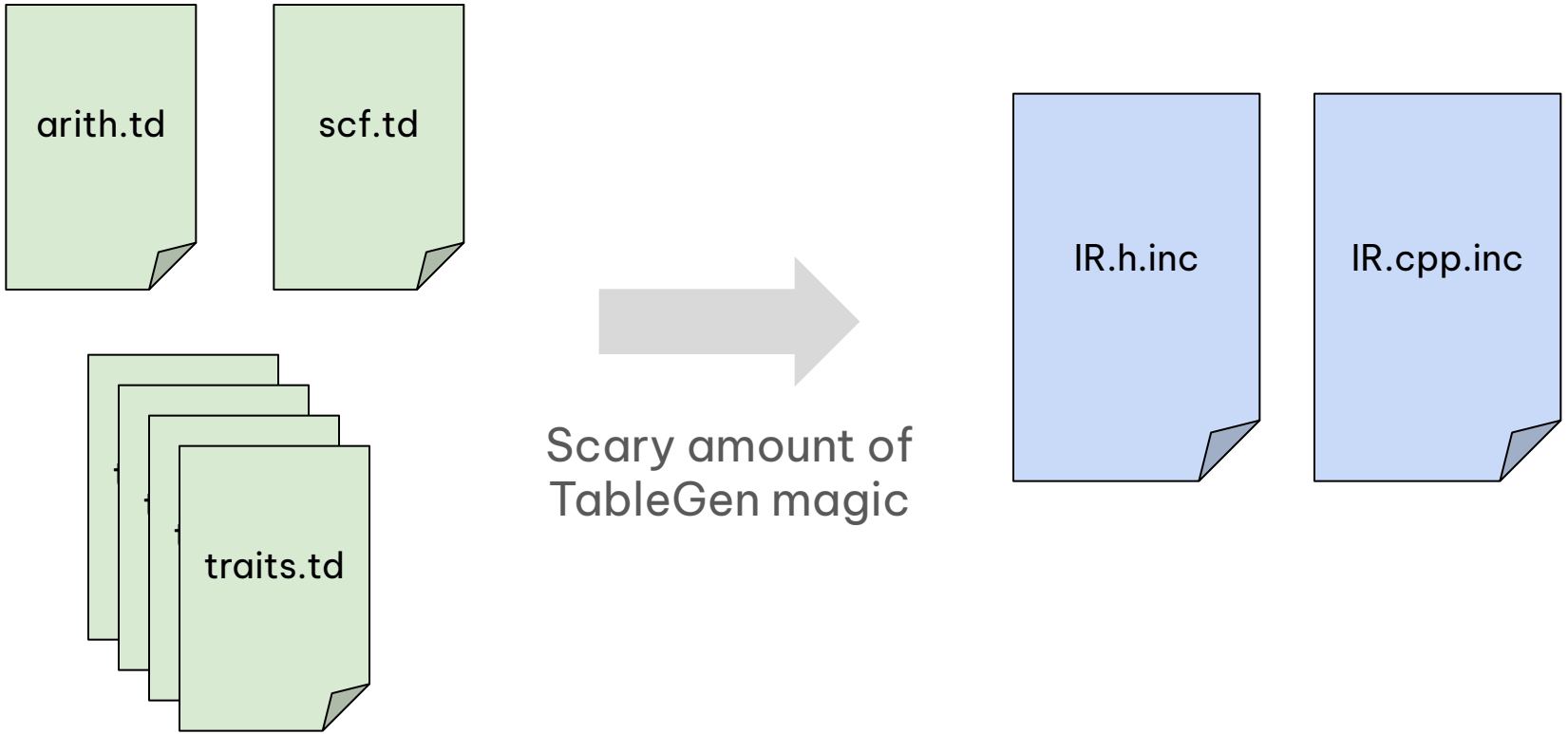
Could we specialize MLIR to a dialect set?



Generating MLIR IR from TableGen?



Generating MLIR IR from TableGen?



Where are the speedup opportunities?

- Checking operation Opcode
- Accessing interfaces and traits
- Reducing memory footprint
- Fixed property size

Checking operation opcodes

`dyn_cast<AddIOp>(op)`

```
ldr    x8, [x0, #0x30]
ldr    x8, [x8, #0x10]
adrp    x9, 0x0
ldr    x9, [x9]
cmp    x8, x9
csel    x0, x0, xzr, eq
ret
```

`AddIOp::opcode == op.opcode`

```
 ldr    w8, [x0, #0x20]
cmp    w8, #0x0
csel    x0, x0, xzr, eq
ret
```

Checking operation opcodes

```
for (Operation *op : /*std::vector*/ ops) {                0.31 ns/op
}
```

Checking operation opcodes

```
for (Operation *op : /*std::vector*/ ops) {
}
```

0.31 ns/op

```
for (Operation *op : ops)
    isa<AddIOp>(op)
```

0.51 ns/op

Checking operation opcodes

```
for (Operation *op : /*std::vector*/ ops) {
}
```
0.31 ns/op

```
for (Operation *op : ops)
    isa<AddIOp>(op)
```
0.51 ns/op

```
for (Operation *op : ops)
    op.opCode == AddIOp::opcode
```
0.39 ns/op

Checking operation opcodes

```
for (Operation *op : /*std::vector*/ ops) {
}
```
0.31 ns/op

```
for (Operation *op : ops)
    isa<AddIOp>(op)
```
2.6 ns/op

```
for (Operation *op : ops)
    op.opCode == AddIOp::opcode
```
2.3 ns/op

Accessing interfaces

Accessing interfaces

µBenchmark: Interfaces vs LLVM


```
static int64_t getCost(llvm::Instruction *op) {
   using namespace llvm;
 switch (op->getOpcode()) {
 // Terminators
 case Instruction::Ret:    return 42;
 case Instruction::Br:     return 13;
 case Instruction::Switch: return 18;
 ...
```

2.7ns/op

```
  for (llvm::Instruction *op : ops) {
    auto cost = getCost(op);
```

```
for (Operation *op : /*std::vector*/ops) {
  if (auto costIface = dyn_cast<CostModel>(op))
     auto cost = casted.getCost();
```

11.71ns/op

Memory footprint improvements

```
struct Operation {
  /*64*/     Block              parent
  /*64*/     Location           location
  /*32*/     unsigned           orderIndex
  /*32*/     unsigned           numResults
  /*32*/     unsigned           numSuccs
  /*23*/     unsigned           numRegions
  /*1*/      bool               hasOperandStorage
  /*8*/      unsigned           propertiesStorageSize
  /*64*/     OperationName      name
  /*64*/     DictionaryAttr     attrs
}
```

Memory footprint improvements

```
struct Operation {
  /*64*/     Block              parent
  /*64*/     Location           location
  /*32*/     unsigned           orderIndex
  /*32*/     unsigned           numResults
  /*32*/     unsigned           numSuccs
  /*23*/     unsigned           numRegions
  /*1*/      bool               hasOperandStorage
  /*8*/      unsigned           propertiesStorageSize
  /*64*/     OperationName      name
  /*64*/     DictionaryAttr     attrs
}
```

Memory footprint improvements

```
struct Operation {
  /*64*/     Block                parent
  /*64*/     Location             location
  /*32*/     unsigned             orderIndex
  /*32*/     unsigned             numResults
  /*32*/     unsigned             numSuccs
  /*23*/     unsigned             numRegions
  /*1*/      bool                 hasOperandStorage
  /*8*/      unsigned             propertiesStorageSize
  /*64*/     OperationName        name
  /*64*/     DictionaryAttr       attrs
}
```

Memory footprint improvements

```
struct Operation {
  /*64*/     Block                 parent
  /*64*/     Location              location
  /*32*/     unsigned              orderIndex
  /*32*/     unsigned              numResults
  /*32*/     unsigned              numSuccs
  /*23*/     unsigned              numRegions
  /*1*/      bool                  hasOperandStorage
  /*8*/      unsigned              propertiesStorageSize
  /*64*/     OperationName         name
  /*64*/     DictionaryAttr        attrs
}
```

24 bytes to save per operation

Memory footprint improvements

```
for (int i = 0; i < numOps; i++) {
    MyOp::create(builder, operands, i32);
}
```

Memory footprint improvements

```
for (int i = 0; i < numOps; i++) {
    MyOp::create(builder, operands, i32);
}
```

	Before	After
0 operands	44 ns/op	
2 operands	75 ns/op	

Memory footprint improvements

```
for (int i = 0; i < numOps; i++) {
    MyOp::create(builder, operands, i32);
}
```

	Before	After
0 operands	44 ns/op	37 ns/op
2 operands	75 ns/op	69 ns/op

Memory footprint improvements

```
for (int i = 0; i < numOps; i++) {
    MyOp::create(builder, operands, i32);
}
```

	Before	After
0 operands	44 ns/op	37 ns/op
2 operands	75 ns/op	69 ns/op

10% speedup

Memory footprint improvements

```
my.addi %cst, %x        ->      my.addi %x, %cst
```

Memory footprint improvements

```
my.addi %cst, %x        ->      my.addi %x, %cst
```

	Before	After
All matches	7.1 ns/op	
No matches	3.6 ns/op	

Memory footprint improvements

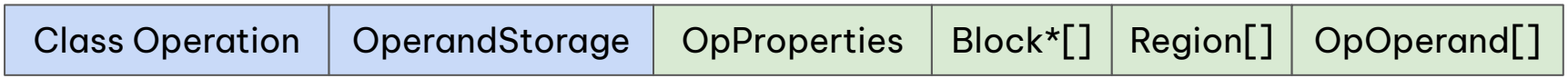
```
my.addi %cst, %x        ->      my.addi %x, %cst
```

	Before	After
All matches	7.1 ns/op	6.1 ns/op
No matches	3.6 ns/op	3 ns/op

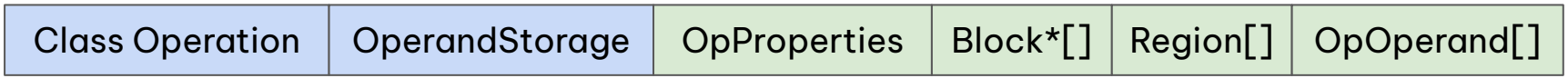
10/20% speedup

Fixed property size

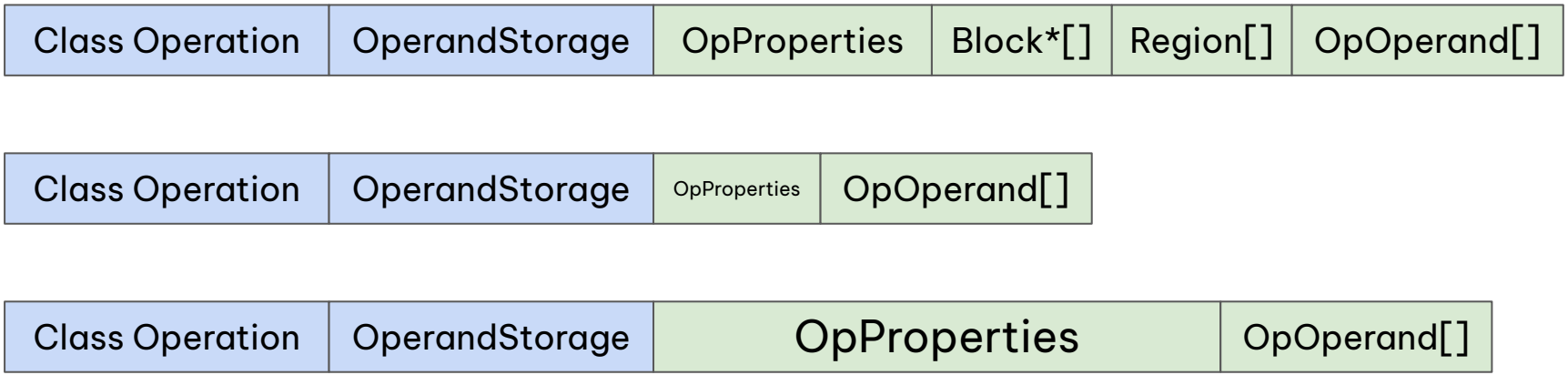
Fixed property size



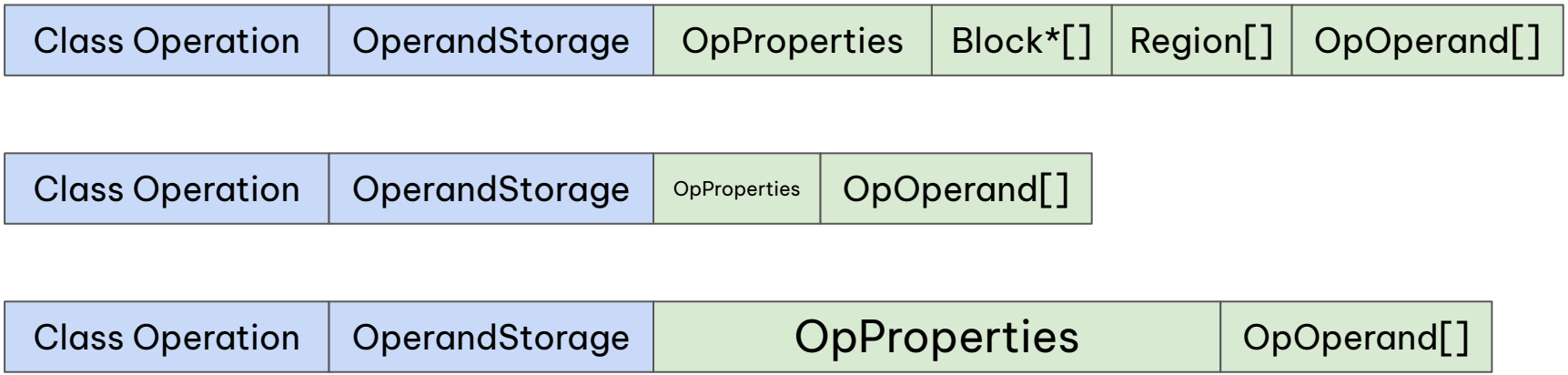
Fixed property size



Fixed property size



Fixed property size



If we fix OpProperties size, can we reuse allocations during rewrites?

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands
- Deallocate old operation

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

```
op.name = myConstantName;
op.eraseOperands(0, op.getNumOperands());
cast<MyConstant>(op).setValue(0);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands
- Deallocate old operation

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands
- Deallocate old operation

```
op.name = myConstantName;
op.eraseOperands(0, op.getNumOperands());
cast<MyConstant>(op).setValue(0);
```

- Change name field

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands
- Deallocate old operation

```
op.name = myConstantName;
op.eraseOperands(0, op.getNumOperands());
cast<MyConstant>(op).setValue(0);
```

- Change name field
- Erase old operands

Fixed property size

```
auto newOp = MyConstant::create(rewriter, value);
rewriter.replaceOp(&op, newOp);
```

- Create a new allocation
- Initialize the new operation
- **Replace results**
- Erase old operands
- Deallocate old operation

```
op.name = myConstantName;
op.eraseOperands(0, op.getNumOperands());
cast<MyConstant>(op).setValue(0);
```

- Change name field
- Erase old operands
- Initialize the new operation

Fixed property size

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

Fixed property size

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

Iterating the IR	3.6 ns/op

Fixed property size

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op

Fixed property size

```
add(constant(cst1), constant(cst2))  ->  constant(cst1 + cst2)
```

Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op
Reusing storage	7.4 ns/op

Fixed property size

```
add(constant(cst1), constant(cst2))  ->  constant(cst1 + cst2)
```

Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op
Reusing storage	7.4 ns/op

15x speedup!!!

Takeaways from these μ-benchmarks

- We can get a few improvements from memory footprint
- Interfaces/traits heavy passes would benefit a lot
- Huge potential from reusing allocations

Could we restrict MLIR features for performance?

Operation	Complexity
Creating an operation *op*	$O(\|op\|)$
Inserting an operation	$O(1)$
Detaching an operation	$O(1)$
Erasing an operation *op*	$O(\|op_{rec}\| + \|op_{rec}.uses\|)$
Replacing an operand/block operand	$O(1)$
Replacing a value v with another value	$O(\|v.uses\|)$
Creating a block *block*	$O(\|block\|)$
Inserting a block	$O(1)$
Detaching a block	$O(1)$
Erasing a block	$O(\|block_{rec}\| + \|block_{rec}.uses\|)$
Creating a region	$O(1)$
Erasing a region	$O(\|region_{rec}\| + \|region_{rec}.uses\|)$

Where could we get performance from?

- Use-def chain updates
- Linked lists walks
- Memory optimizations

Removing the def-use chain

With def-use chain

```
setOperand:
ldr    x8, [x0, #0x48]
mov    w9, w1
add    x8, x8, x9, lsl #5
ldr    x10, [x8, #0x8]
cbz    x10, 0x5f0
ldr    x9, [x8]
str    x9, [x10]
cbz    x9, 0x5f0
ldr    x10, [x8, #0x8]
str    x10, [x9, #0x8]
str    x2, [x8, #0x18]
ldr    x9, [x2]
stp    x9, x2, [x8]
cbz    x9, 0x604
str    x8, [x9, #0x8]
str    x8, [x2]
ret
```

Removing the def-use chain

With def-use chain

```
setOperand:
ldr     x8, [x0, #0x48]
mov     w9, w1
add     x8, x8, x9, lsl #5
ldr     x10, [x8, #0x8]
cbz     x10, 0x5f0
ldr     x9, [x8]
str     x9, [x10]
cbz     x9, 0x5f0
ldr     x10, [x8, #0x8]
str     x10, [x9, #0x8]
str     x2, [x8, #0x18]
ldr     x9, [x2]
stp     x9, x2, [x8]
cbz     x9, 0x604
str     x8, [x9, #0x8]
str     x8, [x2]
ret
```

Without def-use chain

```
setOperand:
ldr     x8, [x0, #0x48]
add     x8, x8, w1, uxtw #4
str     x2, [x8, #0x8]
ret
```

Removing the def-use chain

```
MyAddIOp::create(builder, {})                    45 ns/op

MyAddIOp::create(builder, {a,b,c,d,e,f,g})       77 ns/op
```

Removing the def-use chain

`MyAddIOp::create(builder, {})`	**45** ns/op	->	**45** ns/op
`MyAddIOp::create(builder, {a,b,c,d,e,f,g})`	**77** ns/op	->	**54** ns/op

Removing the def-use chain

`MyAddIOp::create(builder, {})`	**45** ns/op	->	**45** ns/op
`MyAddIOp::create(builder, {a,b,c,d,e,f,g})`	**77** ns/op	->	**54** ns/op

1.42x speedup

Removing the def-use chain

```
my.addi %cst, %x        ->      my.addi %x, %cst
```

	MLIR	Reduce memory	
All matches	7.1 ns/op	6.1 ns/op	
No matches	3.6 ns/op	3 ns/op	

Removing the def-use chain

```
my.addi %cst, %x        ->      my.addi %x, %cst
```

	MLIR	Reduce memory	Without def-use chain
All matches	7.1 ns/op	6.1 ns/op	4.8 ns/op
No matches	3.6 ns/op	3 ns/op	3.6 ns/op

1.47x speedup

Removing the def-use chain

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op
Reusing storage	7.4 ns/op

Removing the def-use chain

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

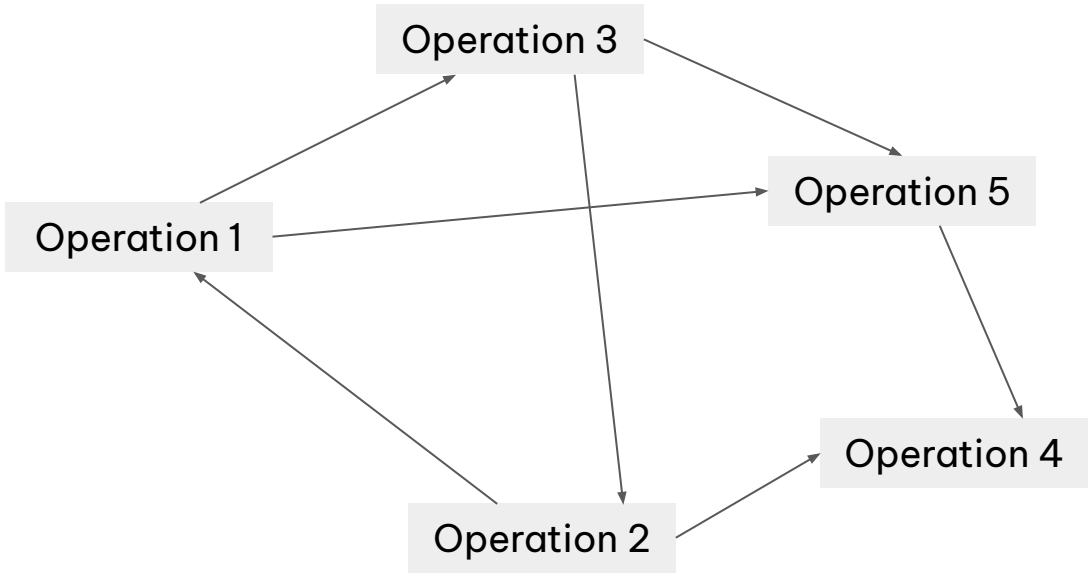
Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op
Reusing storage	7.4 ns/op
No def-use chain	5.3 ns/op

Removing the linked-list

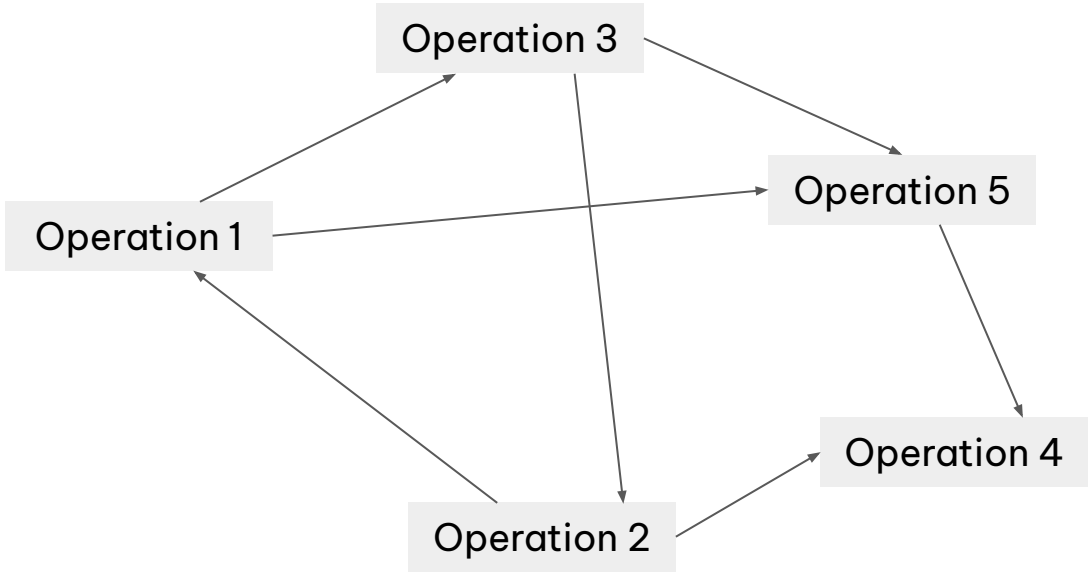
add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

Iterating the IR	3.6 ns/op
Replacing ops	114 ns/op
Reusing storage	7.4 ns/op
No def-use chain	5.3 ns/op
No linked-list	4.9 ns/op

"Arena" allocation for operations

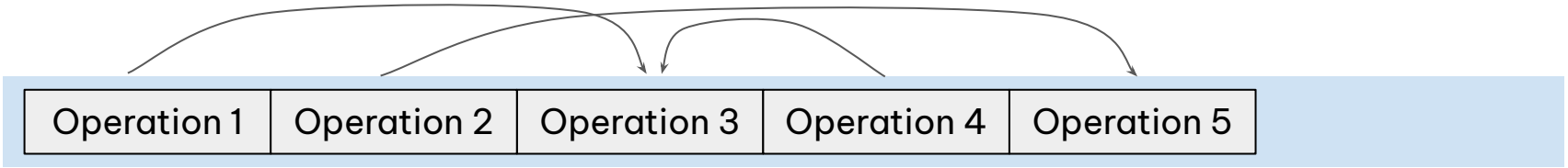


"Arena" allocation for operations

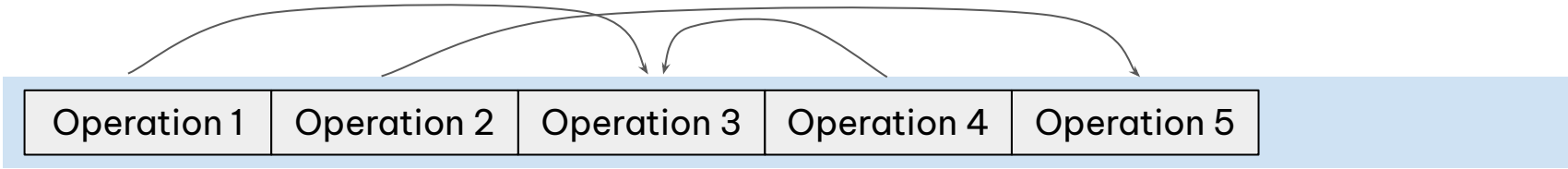


- Each operation requires a new allocation
- Each pointer has to be 64 bits

"Arena" allocation for operations



"Arena" allocation for operations



- One global resizable allocation
- Pointers are offsets (possibly 32 bits)
- All operations have the same size

"Arena" allocation for operations

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

"Arena" allocation for operations

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

	MLIR
Creating the IR	75 ns/op
Rewriting the IR	114 ns/op

"Arena" allocation for operations

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

	MLIR	7 operands
Creating the IR	75 ns/op	19 ns/op
Rewriting the IR	114 ns/op	76 ns/op

"Arena" allocation for operations

add(constant(cst1), constant(cst2)) -> constant(cst1 + cst2)

	MLIR	7 operands	3 operands
Creating the IR	75 ns/op	19 ns/op	15 ns/op
Rewriting the IR	114 ns/op	76 ns/op	62 ns/op

Conclusion

- Very powerful optimization opportunities when the IR size is fixed

Conclusion

- Very powerful optimization opportunities when the IR size is fixed
- Memory optimizations go a long way

Conclusion

- Very powerful optimization opportunities when the IR size is fixed
- Memory optimizations go a long way
- Maybe there is hope to have an LLVM dialect as fast as LLVM?

Conclusion

- Very powerful optimization opportunities when the IR size is fixed
- Memory optimizations go a long way
- Maybe there is hope to have an LLVM dialect as fast as LLVM?
- Can we apply some of these optimizations in MLIR already?

Conclusion

- Very powerful optimization opportunities when the IR size is fixed
- Memory optimizations go a long way
- Maybe there is hope to have an LLVM dialect as fast as LLVM?
- Can we apply some of these optimizations in MLIR already?