How Slow is MLIR?

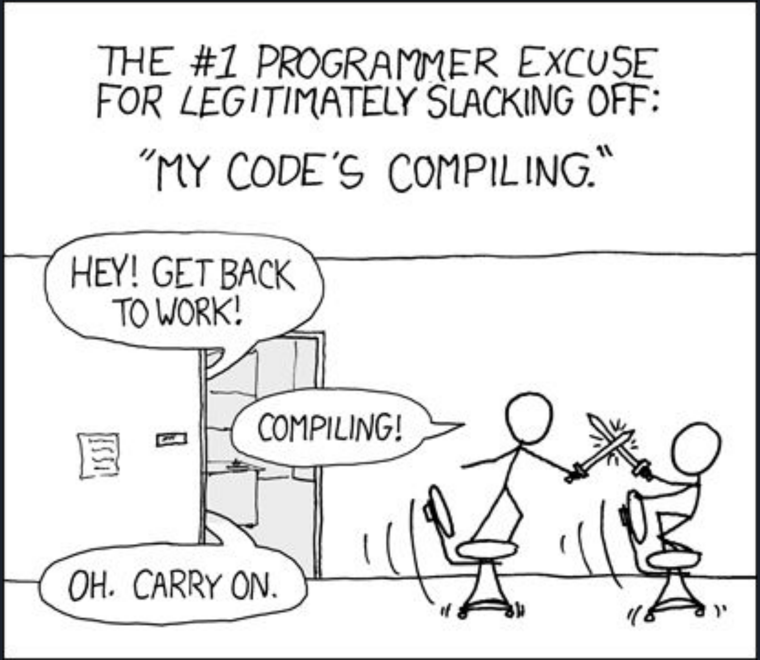
Mehdi Amini (NVIDIA)
Jeff Niu (Modular)

Agenda

- Motivation
- MLIR IR internals and data structures
- Quantifying costs
- Ideas to improve MLIR performance?

Compile Time is Important

- Engineering time, developer productivity (💰💰💰)
- UX (torch.compile, etc.)
- p95 model latency
- CI/CD/dev hardware costs



Ultra-fast JITs are out-of-scope:
https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/

Is MLIR slow?

- How slow is too slow? Is LLVM slow?
- Cost of abstraction / extensibility (indirection)
- Runtime extensibility of interfaces
- Are the fundamental IR building blocks slow?
- Kitchen-sink batteries

```
class OperationName {
public:
  /// This class represents a type erased version of an operation. It contains
  /// all of the components necessary for opaquely interacting with an
  /// operation. If the operation is not registered, some of these components
  /// may not be populated.
  struct InterfaceConcept {
    virtual ~InterfaceConcept() = default;
    virtual LogicalResult foldHook(Operation *, ArrayRef<Attribute>,
                                   SmallVectorImpl<OpFoldResult> &) = 0;
    virtual void getCanonicalizationPatterns(RewritePatternSet &,
                                             MLIRContext *) = 0;
    virtual bool hasTrait(TypeID) = 0;
    virtual OperationName::ParseAssemblyFn getParseAssemblyFn() = 0;
    virtual void populateDefaultAttrs(const OperationName &,
                                      NamedAttrList &) = 0;
    virtual void printAssembly(Operation *, OpAsmPrinter &, StringRef) = 0;
    virtual LogicalResult verifyInvariants(Operation *) = 0;
    virtual LogicalResult verifyRegionInvariants(Operation *) = 0;
    /// Implementation for properties
    virtual std::optional<Attribute> getInherentAttr(Operation *,
                                                     StringRef name) = 0;
    virtual void setInherentAttr(Operation *op, StringAttr name,
                                 Attribute value) = 0;
    virtual void populateInherentAttrs(Operation *op, NamedAttrList &attrs) = 0;
```

MLIR Internals 🛠️

```
class Operation[sizeof=64]

       0 | PointerIntPair<Operation *, 1> prevAndSentinel;
       8 | Operation *next;
      16 | Block *block;
      24 | Location location;
      32 | unsigned orderIndex;
      36 | unsigned numResults;
      40 | unsigned numSuccs;
44[0,22] | unsigned numRegions;
  46[7,7] | bool hasOperandStorage;
      47 | unsigned char propertiesStorageSize;
      48 | OperationName name;
      56 | DictionaryAttr attrs;
```

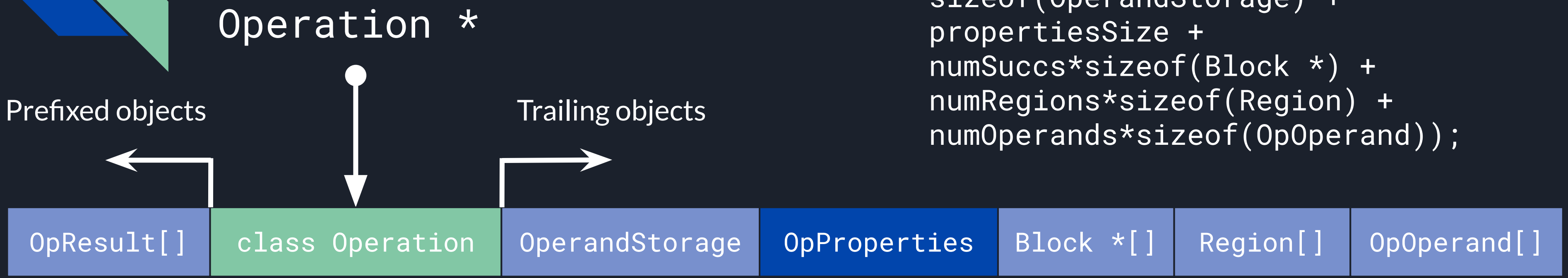
Doubly-linked list (llvm::iplist)

Where are the lists stored?

TypeID and “virtual table” implementation (fold hook, properties destructor, etc.)

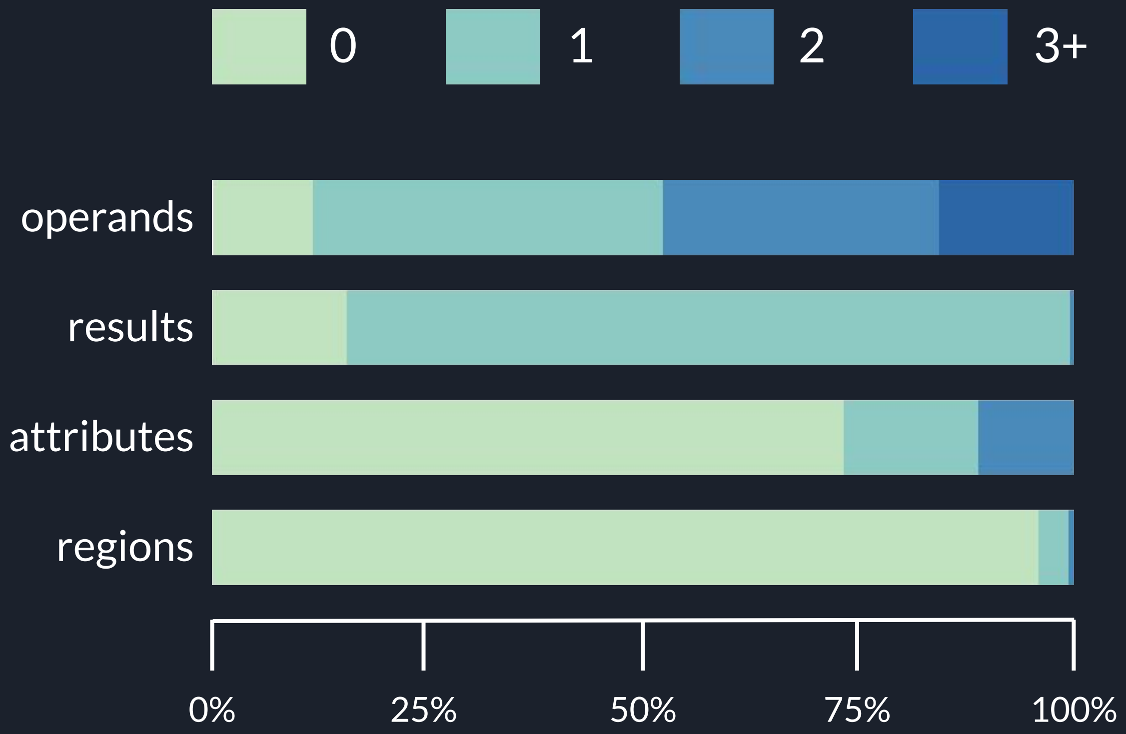
https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/

```
malloc(numResults*sizeof(OpResult) +
       sizeof(Operation) +
       sizeof(OperandStorage) +
       propertiesSize +
       numSuccs*sizeof(Block *) +
       numRegions*sizeof(Region) +
       numOperands*sizeof(OpOperand));
```



- Lists are stored in-line (less indirection, memory locality)
- Nothing is allocated if unneeded (e.g. no results, no regions, etc.)
- OpProperties stores an arbitrary C++ type!

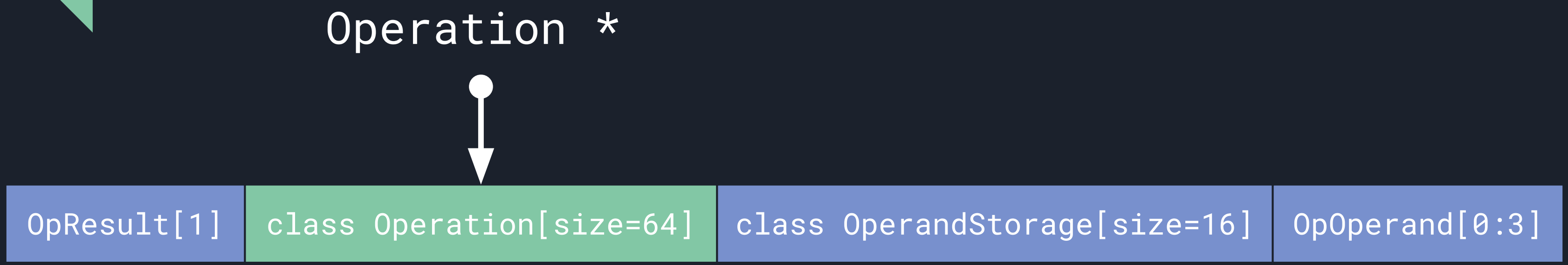
The Average* MLIR Operation



*upstream, circa 2022

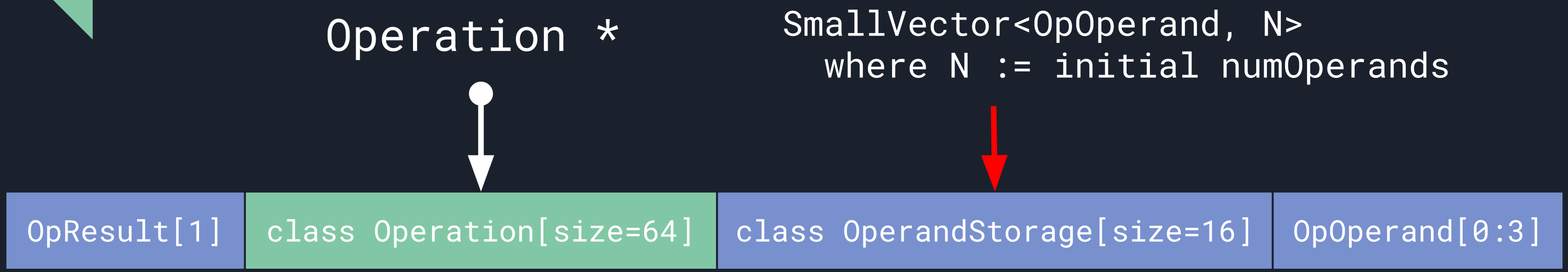
IRDL paper

The Average* MLIR Operation



- Results are to Operation* (16 bytes to the left)
- Operands are further away (80 bytes to the right)
- Cache line sizes: 64 bytes (Intel, AMD), 128 bytes (Apple)

Result vs. Operand List Mutability



- Number of results immutable
- Operands can be added or removed (cost: additional indirection)

```
class OperandStorage[sizeof=16]

 0 | unsigned capacity;
 7 | bool isStorageDynamic;
 8 | OpOperand *operandStorage;
```

Whether operandStorage is malloc'd or points to trailing storage

The Cost of Mutability

```
Value getFirstOperand(Operation *op) {
 return op->getOperand(0);
}
```

```
<getFirstOperand>:
    ldr    x8, [x0, #72]
    ldr    x0, [x8, #24]
    ret
```

```
Value getFirstResult(Operation *op) {
 return op->getResult(0);
}
```

```
<getFirstResult>:
    sub    x0, x0, #16
    ret
```

Optimizing Result Storage Size

OpResult

```
class InlineOpResult[sizeof=16]

 0 | IROperandBase *firstUse;
 8 | PointerIntType<Type, 3, Kind>
     typeAndKind;
```

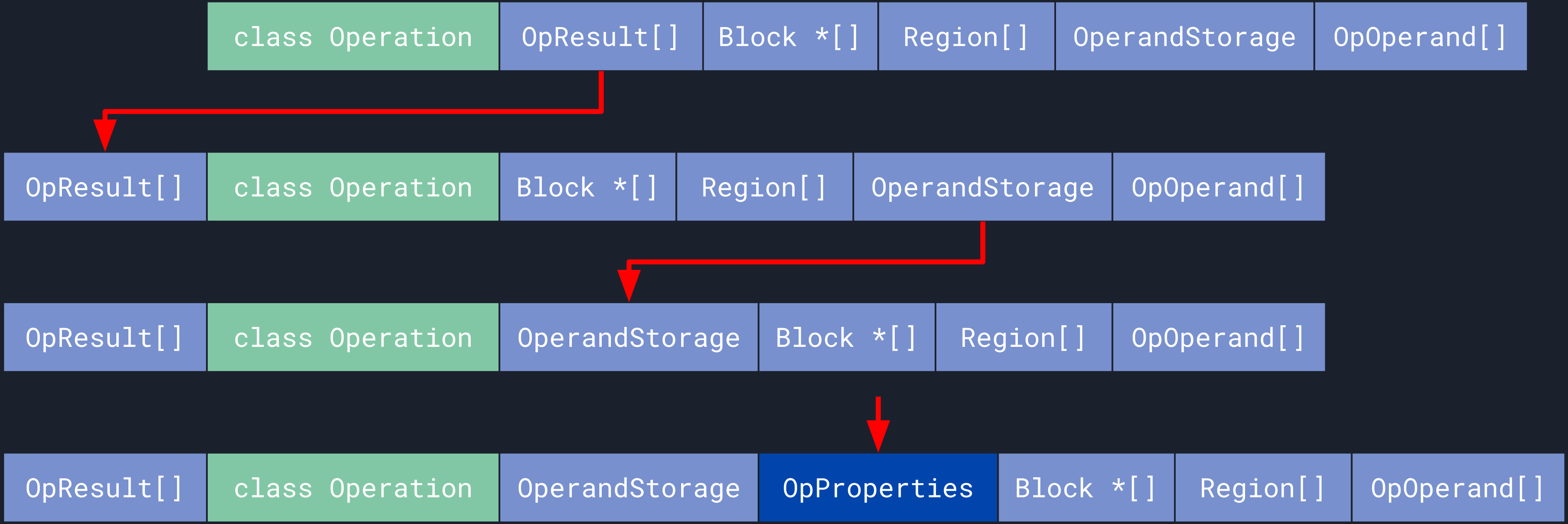
Values [1,6] used to store result index [0,5]

```
class OutOfLineOpResult[sizeof=24]

 0 | IROperandBase *firstUse;
 8 | PointerIntType<Type, 3, Kind>
     typeAndKind;
16 | int64_t outOfLineIndex;
```

Result indices greater than 5 stored separately

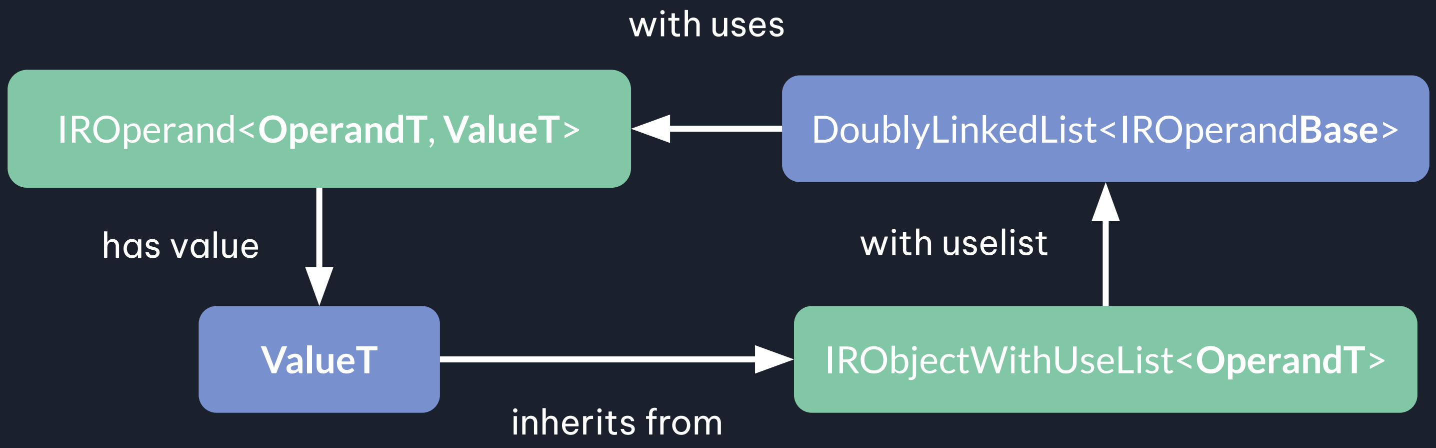
Evolution of Operation Trailing Objects

Main cost: computing offsets
to the Nth trailing objects



Use-Def Lists

- Each IRValue (Value, BlockOperand, etc.) contains a linked list of users



Use-Def Lists Properties

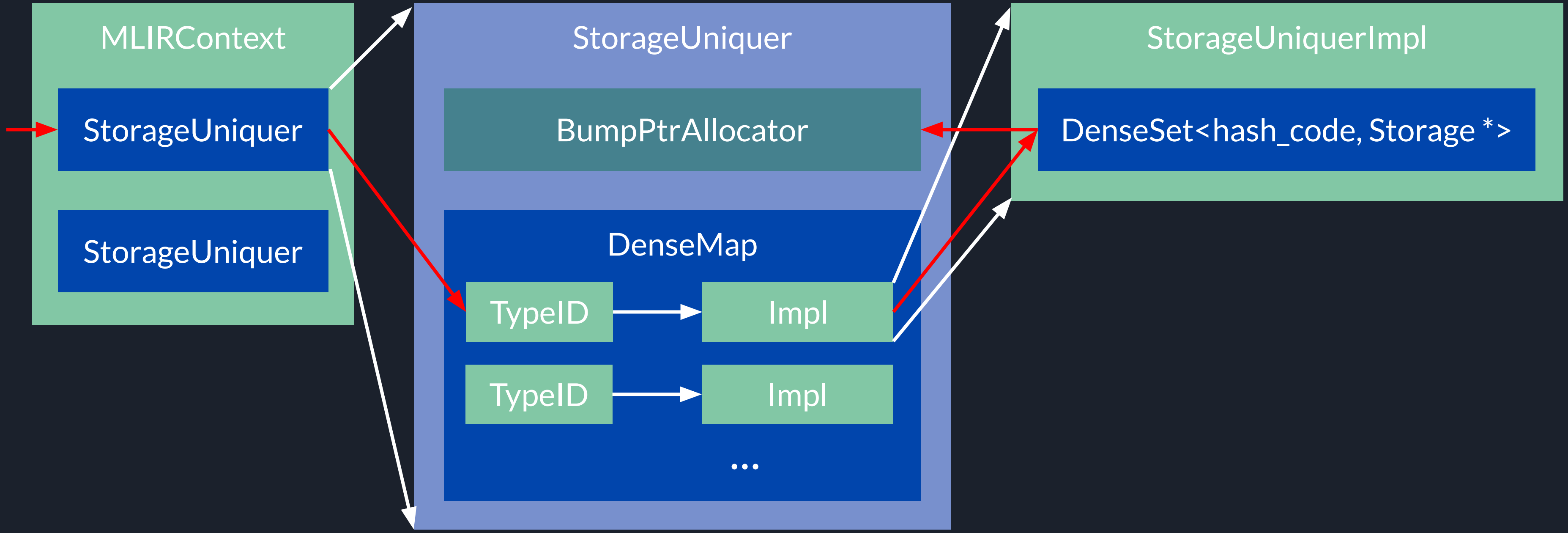
- O(1) insertion and removal
- Not thread-safe (*IsIsolatedFromAbove*!)
- **Sparse**

```
class ValueImpl[sizeof=16]

 0 | OpOperand *firstUse;
 8 | PointerIntType<Type, 3, Kind>
     typeAndKind;
```

Note: user Operations may be duplicated

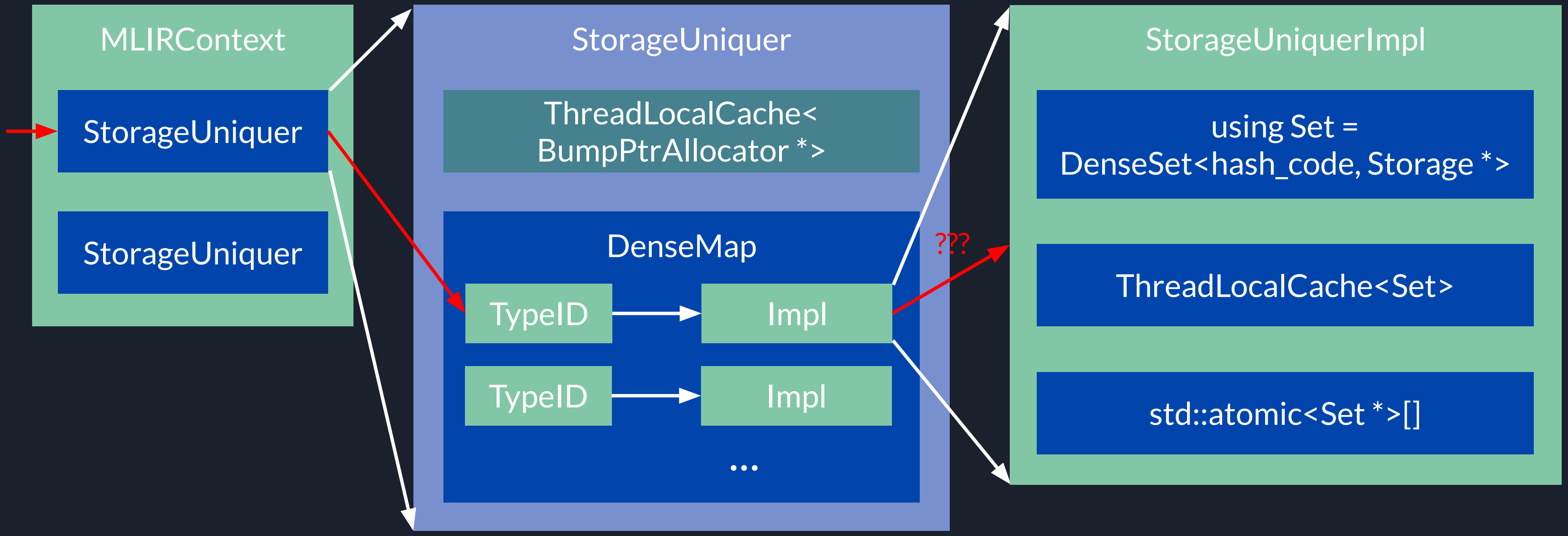
StorageUniquer

- Manages unique, **immortal** (lifetime of MLIRContext) storage for attributes and types

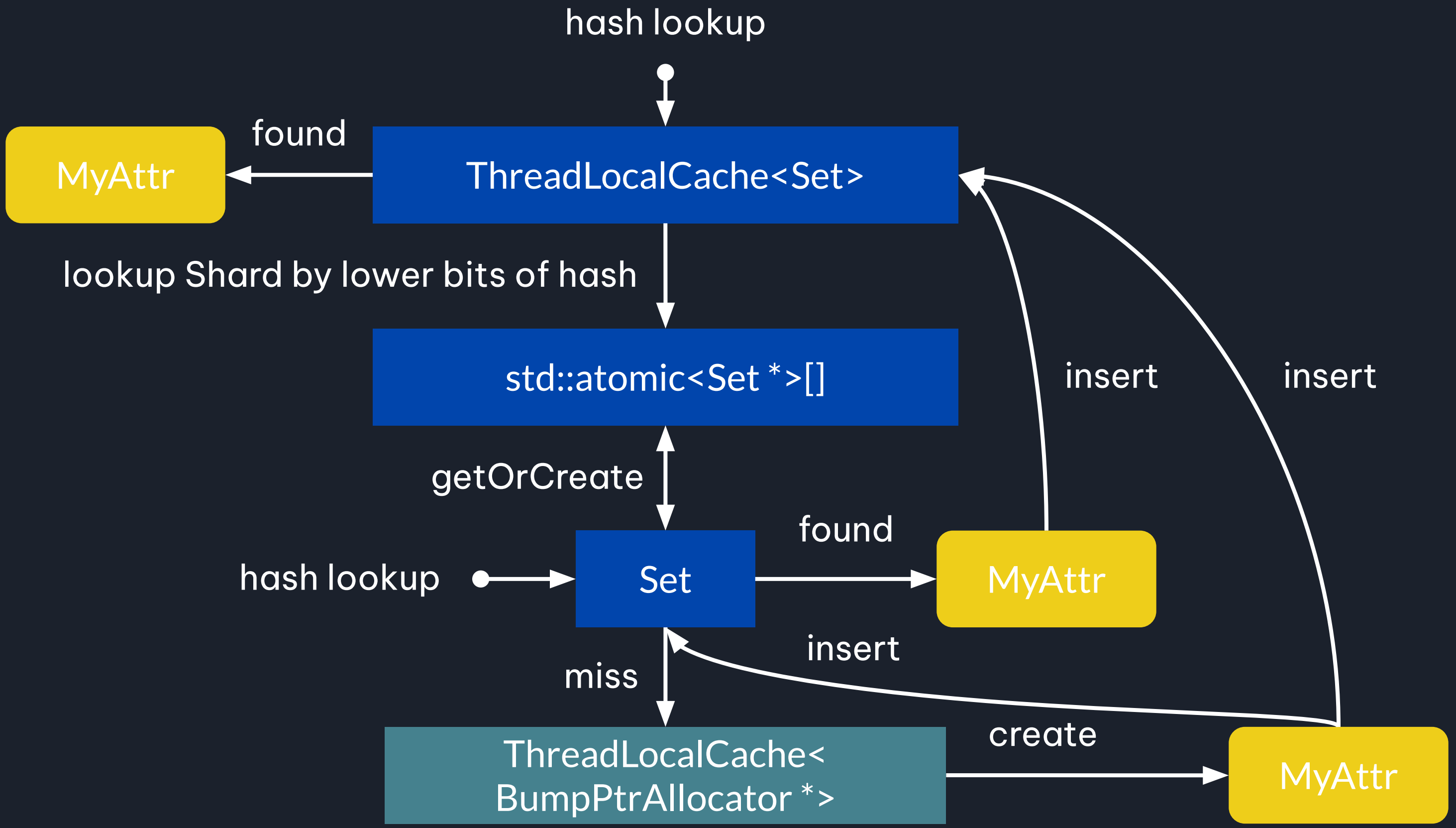


StorageUniquer

- Sharded across threads! Needs to be efficiently thread-safe



Multithreaded StorageUniquer



StorageUniquer – Takeaways

- Benefits of immutability: pointer identity
  - Cheap to copy around
  - Precomputed hash and equality
- Costs are **paid up-front**
  - Computing hashes is slow
  - Hashmaps are slow
- "Leaks" memory – long-lived MLIRContext?

Operation Properties!

In-line storage of arbitrary C++ types

OpResult[1]	class Operation	OperandStorage	CmpOpProperties	OpOperand[2]

```
enum class IndexCmpPredicateKind {
  EQ, NE, SLT, SLE, SGT, SGE,
          ULT, ULE, UGT, UGE
};

struct CmpOpProperties {
  IndexCmpPredicateKind pred;
};
```

Type dispatch through RegisteredOperationName

- Ctor, copy assignment, dtor
- Equality, hashing (OpEquivalence)
- setPropertiesFromAttr, getPropertiesAsAttr

Operation Attributes

Linear scan over DictionaryAttr keys

```
template <typename IteratorT>
std::pair<IteratorT, bool> findAttrSorted(IteratorT first, IteratorT last,
                                          StringAttr name) {
  constexpr unsigned kSmallAttributeList = 16;
  if (std::distance(first, last) > kSmallAttributeList)
    return findAttrSorted(first, last, name.strref());
  return findAttrUnsorted(first, last, name);
}
```

setAttr is even worse!
(rehash the DictionaryAttr)

(New!) Attributes Stored as Properties

Default setting for all MLIR dialects

```
def OpWithInteger : Op<"with_integer"> {
  let arguments = (ins I32Attr:$intValue);
}

struct OpWithIntegerProperties {
  IntegerAttr intValue;
};
```

```
<OpWithInteger::getIntValueAttr>:
   ldr x8, [x0]                  // *this
   ldr w9, [x8, #44]
   ubfx    x10, x9, #23, #1      // hasOperandStorage?
   add x8, x8, x10, lsl #4       // += sizeof(OperandStorage)
   add x8, x8, #64               // += sizeof(Operation)
   ubfx    x9, x9, #24, #8
   cmp w9, #0                    // propertiesStorageSize?
   csel    x8, xzr, x8, eq
   ldr x0, [x8]
   ret
```

Still storing an attribute...

Why store an `IntegerAttr` when you want an `int32_t`?

```
<OpWithInteger::getIntValue>:
   // ... x8 = getIntValueAttr
   str x8, [sp, #24]
   add x8, sp, #8
   add x0, sp, #24
   bl  0x2958 <OpWithInteger::getIntValue+0x40>
   ldr w8, [sp, #16]
   cmp w8, #64
   b.hi    0x2970 <OpWithInteger::getIntValue+0x58>
   ldr x19, [sp, #8]
   b   0x297c <OpWithInteger::getIntValue+0x64>
   ldr x0, [sp, #8]
   ldr x19, [x0]
   bl  0x2978 <OpWithInteger::getIntValue+0x60>
   mov x0, x19
   ldp x29, x30, [sp, #48]
   ldp x20, x19, [sp, #32]
   add sp, sp, #64
   ret
```

`IntegerAttr::getValue`

`APInt::getZExtValue`

Using Native Properties!

```
def OpWithInteger : Op<"with_integer"> {
  let arguments = (ins
    IntProperty<"int32_t">:$intValue
  );
}
```

```
<OpWithInteger::getIntValue>:
  ldr w8, [x0, #44]
  ubfx    x9, x8, #23, #1
  add x9, x0, x9, lsl #4
  add x9, x9, #64
  ubfx    x8, x8, #24, #8
  cmp w8, #0
  csel    x8, xzr, x9, eq
  ldr w0, [x8]              // load the int32_t directly
  ret
```

TrailingObjects overhead

...And more!

- Block structure
  - `BlockArgument`
  - `Operation::getBlock` – splice is O(n)
- Region structure – `iplist<Block>`
- Traits, interfaces (yesterday: Deep Dive on MLIR Interfaces)
- Dynamic dispatch (`RegisteredOperationName`, dialect fallbacks, ...)

μBenchmarking MLIR ⏱️

Disclaimers ⚠️

- Goal: build intuition about performance “orders of magnitude” of MLIR APIs
- Asymptotic numbers – not always representative (benefits dense structures)

µBenchmark: IR Traversals

```
for (Operation *op : /*std::vector*/ops) {                          0.35ns/op
for (Operation &op : *block) {                                      2.15ns/op
for (llvm::Instruction &op : *block) {                              2.15ns/op
for (ModuleOp op : moduleOp->getBody()->getOps<ModuleOp>())         2.21ns/op

block->walk([](Operation *op) {}); /*no region in the IR!*/         6.11ns/op
block->walk([](Operation *op) {}); /*ops with 1 region*/            7.34ns/op
```

µBenchmark: Interfaces and Traits Lookups

```
for (Operation *op : /*std::vector*/ops) {                              0.35ns/op
  assert( ! dyn_cast<OpT>(op) )                                          2.16ns/op
  assert( dyn_cast<OpT>(op) )                                            2.16ns/op

  assert( ! dyn_cast<InterfaceT>(op) ) /*op without interface*/          5.85ns/op
  assert( ! dyn_cast<InterfaceT>(op) ) /*op with interface*/             6.92ns/op
  assert( dyn_cast<InterfaceT>(op) )                                     9.68ns/op

  assert( ! op->hasTrait<TraitT>(op) )                                   13.4ns/op
  assert( op->hasTrait<TraitT>(op) )                                     18.1ns/op
```

µBenchmark: Interfaces vs LLVM

4x faster!

```
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

IR Traversals – Takeaways

- Walking IR is >10x slower than traversing a vector

=> Faster to push_back into a vector for subsequent traversals

- Interfaces extensibility comes with overhead

µBenchmark StorageUniquer (Type/Attribute)

Maximum contention possible

	1 thread	4 threads	`ctx.disableMultithreading();`
`for (int counter = 0; counter < N; ++counter)` `  StringAttr::get(&ctx, Twine(0));`	62.8ns/op	446.93ns/op	53.36ns/op
`for (int counter = 0; counter < N; ++counter)` `  StringAttr::get(&ctx, Twine(counter));`	651ns/op	1439ns/op	454.27ns/op

Reader-lock contention

30% speedup when disabling MLIRContext thread-safety!

StorageUniquer – Takeaways

- Type/Attribute creation, even on a hit, is slow
  - Cache types and attributes on pass instances and re-use them
  - Avoid sugared custom op builders (e.g. wrap IntegerAttrs)
- ***Noticeable*** multithreading overhead
  - Be wary of hitting StorageUniquer frequently (inlining creating CallSiteLoc, rewriting complex metadata)
- **Use native properties!**

µBenchmark: Operation Creation

```
for (int counter = 0; counter < N; ++counter) {
  OperationState opState(unknownLoc, "testbench.empty");
  Operation::create(opState);
}
```

118ns/op

```
OperationState opState(unknownLoc, "testbench.empty");
for (int counter = 0; counter < N; ++counter)
  Operation::create(opState);
```

82ns/op

```
for (int counter = 0; counter < N; ++counter)
  opBuilder.create<EmptyOp>(unknownLoc);
```

99ns/op

```
for (int counter = 0; counter < N; ++counter)
  llvmBuilder.CreateUnreachable();
```

37ns/op

```
OwningOpRef<ModuleOp> moduleClone = moduleOp->clone();
```

631ns/op

μBenchmark: Pattern Drivers

```
applyPatternsAndFoldGreedily(moduleOp, /*empty*/frozenPatterns);
```

167ns/op

```
RewritePatternSet patterns(ctx.get());
populateCanonicalizationPatterns(patterns);
FrozenRewritePatternSet frozenPatterns(std::move(patterns));
applyPatternsAndFoldGreedily(moduleOp, frozenPatterns);
```

293ns/op

```
applyPartialConversion(moduleOp.get(), target, /*empty*/patterns)
```

458ns/op

```
applyPartialConversion(moduleOp.get(), target, /*full*/patterns)
```

5398ns/op

µBenchmark

Too much bookkeeping required
(but encode more info, like use-list)

```
(void)writeBytecodeToFile(*moduleOp, os);
```

1049ns/op

```
AsmState asmState(*moduleOp, OpPrintingFlags());
moduleOp->print(os, asmState);
```

866ns/op

```
readBytecodeFile(*buf, &owningBlock, config);
```

1571ns/op

```
parseSourceFile<ModuleOp>(sourceMgr, config);
```

3359ns/op

Takeaways

- Cloning is unfortunately slow, for something that is the basis of many transformations (inlining, unrolling)
- Writing bytecode can be slower than writing text!
  - Dialect resources are another concern
  - Other ways to stably hash IR?
- Pattern rewriter is too often reached for as the base API for applying IR transformations
- Bookkeeping seems more impactful than traversing sparse IR!

“Real world” benchmarking: Constant Folding

```
define i64 @folding() {
 %1 = add i64 13, 7907
 %2 = sub i64 7907, 13
 %3 = add i64 %1, %2
 %4 = sub i64 %2, %1
 %5 = add i64 %3, %4
 %6 = sub i64 %4, %3
 %7 = add i64 %5, %6
 %8 = sub i64 %6, %5
 %9 = add i64 %7, %8
 %10 = sub i64 %8, %7
 %11 = add i64 %9, %10
 %12 = sub i64 %10, %9
 %13 = add i64 %11, %12
 %14 = sub i64 %12, %11
 %15 = add i64 %13, %14
 %16 = sub i64 %14, %13
 %17 = add i64 %15, %16
 %18 = sub i64 %16, %15
 %19 = add i64 %17, %18
 %20 = sub i64 %18, %17
```

Constant Folding →

```
define i64 @folding() {
 ret i64 55834574848
}
```

“Real world” benchmarking: Constant Folding

```
func.func @folding() -> index {
 %c13 = arith.constant 13 : index
 %c7907 = arith.constant 7907 : index
 %0 = arith.addi %c13, %c7907 : index
 %1 = arith.subi %c7907, %c13 : index
 %2 = arith.addi %0, %1 : index
 %3 = arith.subi %1, %0 : index
 %4 = arith.addi %2, %3 : index
 %5 = arith.subi %3, %2 : index
 %6 = arith.addi %4, %5 : index
 %7 = arith.subi %5, %4 : index
 %8 = arith.addi %6, %7 : index
 %9 = arith.subi %7, %6 : index
 %10 = arith.addi %8, %9 : index
 %11 = arith.subi %9, %8 : index
 %12 = arith.addi %10, %11 : index
 %13 = arith.subi %11, %10 : index
 %14 = arith.addi %12, %13 : index
 %15 = arith.subi %13, %12 : index
 %16 = arith.addi %14, %15 : index
 %17 = arith.subi %15, %14 : index
```

Constant Folding →

```
func.func @folding() -> index {
 %cst = arith.constant 55834574848 : in
 return %cst : index
}
```

“Real world” benchmarking: Constant Folding

`78ns / operation`

`325ns / operation`

4x faster!

“Real world” benchmarking: Loop Unrolling

```
int loopUnroll(int a, int b) {
  for (int i = 0; i < 32; ++i) {
    int add = a + b;
    int sub = b - a;
    a = sub;
    b = add;
  }
  for (int i = 0; i < 32; ++i) {
    int add = a + b;
    int sub = b - a;
    a = sub;
    b = add;
  }
  <repeat loop N times>
  return a;
}
```

Unroll by 4

```
for (int i = 0; i < 32; i+=4) {
  int add = a + b;
  int sub = b - a;
  a = sub;
  b = add;
  add = a + b;
  sub = b - a;
  a = sub;
  b = add;
  add = a + b;
  sub = b - a;
  a = sub;
  b = add;
  add = a + b;
  sub = b - a;
  a = sub;
  b = add;
}
}
```

"Real world" benchmarking: Loop Unrolling

```
module {
 func.func @loopUnroll(%arg0: index, %arg1: index)
      -> index {
   %c0 = arith.constant 0 : index
   %c32 = arith.constant 32 : index
   %c1 = arith.constant 1 : index
   %0:2 = scf.for %arg2 = %c0 to %c32 step %c1
       iter_args(%arg3 = %arg0, %arg4 = %arg1)
       -> (index, index) {
     %8 = arith.addi %arg3, %arg4 : index
     %9 = arith.subi %arg4, %arg3 : index
     scf.yield %9, %8 : index, index
   }
   %1:2 = scf.for %arg2 = %c0 to %c32 step %c1
       iter_args(%arg3 = %arg0, %arg4 = %arg1)
       -> (index, index) {
     %8 = arith.addi %arg3, %arg4 : index
     %9 = arith.subi %arg4, %arg3 : index
     scf.yield %9, %8 : index, index
   }
   return %1#0 : index
 }
```

Unroll by 4

```
%c4 = arith.constant 4 : index
%0:2 = scf.for %arg2 = %c0 to %c32 step %c4
    iter_args(%arg3 = %arg0, %arg4 = %arg1)
    -> (index, index) {
  %1 = arith.addi %arg3, %arg4 : index
  %2 = arith.subi %arg4, %arg3 : index
  %3 = arith.addi %2, %1 : index
  %4 = arith.subi %1, %2 : index
  %5 = arith.addi %4, %3 : index
  %6 = arith.subi %3, %4 : index
  %7 = arith.addi %6, %5 : index
  %8 = arith.subi %5, %6 : index
  scf.yield %8, %7 : index, index
}
```

“Real world” benchmarking: Loop Unrolling

```
define i64 @loopUnroll(i64 %0, i64 %1) {
 br label %3
3: ; preds = %8, %2
 %4 = phi i64 [ %11, %8 ], [ 0, %2 ]
 %5 = phi i64 [ %10, %8 ], [ %0, %2 ]
 %6 = phi i64 [ %9, %8 ], [ %1, %2 ]
 %7 = icmp slt i64 %4, 32
 br i1 %7, label %8, label %12
8: ; preds = %3
 %9 = add i64 %5, %6
 %10 = sub i64 %6, %5
 %11 = add i64 %4, 1
 br label %3
12: ; preds = %3
 ret i64 %5
}
```

Unroll by 4

```
3: ; preds = %23, %2
 %4 = phi i64 [ 0, %2 ], [ %26, %23 ]
 %5 = phi i64 [ %0, %2 ], [ %25, %23 ]
 %6 = phi i64 [ %1, %2 ], [ %24, %23 ]
 %7 = icmp slt i64 %4, 32
 br i1 %7, label %8, label %27
8: ; preds = %3
 %9 = add i64 %5, %6
 %10 = sub i64 %6, %5
 %11 = add i64 %4, 1
 %12 = icmp slt i64 %11, 32
 br label %13
13: ; preds = %8
 %14 = add i64 %10, %9
 %15 = sub i64 %9, %10
 %16 = add i64 %11, 1
 %17 = icmp slt i64 %16, 32
 br label %18
18: ; preds = %13
 %19 = add i64 %15, %14
 %20 = sub i64 %14, %15
 %21 = add i64 %16, 1
 %22 = icmp slt i64 %21, 32
 br label %23
23: ; preds = %18
 %24 = add i64 %20, %19
 %25 = sub i64 %19, %20
 %26 = add i64 %21, 1
 br label %3
```

“Real world” benchmarking: Loop Unrolling

177000ns / loop

2000ns / loop

88x faster!

“Real world” benchmarking: Loop Unrolling

Profile llvm::UnrollLoop(llvm::Loop*, llvm::UnrollLoopOptions, llvm::LoopInfo*, llvm::ScalarEvolution*, llvm::DominatorTree*, llvm::AssumptionCache*, llvm::T

Weight		Self Weight	Symbol Name
3.21 Gc	100.0%	18.00 Mc	llvm::UnrollLoop(llvm::Loop*, llvm::UnrollLoopOptions, llvm::LoopInfo*, llvm::ScalarEvolution*, llvm::DominatorTree*,
2.61 Gc	81.3%	-	llvm::ScalarEvolution::getSmallConstantTripMultiple(llvm::Loop const*, llvm::SCEV const*) MLIR_IR_Benchmark
85.54 Mc	2.6%	4.00 Mc	llvm::CloneBasicBlock(llvm::BasicBlock const*, llvm::ValueMap<llvm::Value const*, llvm::WeakTrackingVH, llvm::Va
62.68 Mc	1.9%	-	llvm::remapInstructionsInBlocks(llvm::ArrayRef<llvm::BasicBlock*>, llvm::ValueMap<llvm::Value const*, llvm::Weal
40.01 Mc	1.2%	-	llvm::ValueMap<llvm::Value const*, llvm::WeakTrackingVH, llvm::ValueMapConfig<llvm::Value const*, llvm::sys::Sm
36.88 Mc	1.1%	1.00 Mc	llvm::ScalarEvolution::forgetLoop(llvm::Loop const*) MLIR_IR_Benchmark

177000ns / loop

2000ns / loop

88x faster!

Takeaways: IR Design and Compile Time

- High-level dialects provide coarser-grain IR representation that is more efficient (`scf.for` vs loop with CFG)
  => what about adding first-class loops/regions in LLVM?
- Structural guarantees of reducible control flow mean algorithmic wins
- More levels of IR means more Dialect Conversion: it's costly!
  => Tradeoffs: new levels of abstraction should be well motivated.

4:15 PM: Efficient Data-Flow Analysis on Region-Based Control Flow in MLIR

Top K overheads (Mojo)

1. 60-80% is LLVM
2. Lock contention
3. MLIR verifier (lots of hash maps)
4. Allocator pressure (create/destroy ops)
5. Greedy rewriter overheads (?)
6. MLIR interface lookup
7. IR structure overhead (*iplist*, *Block->getParentOp()*, *getAttrDictionary()*)
8. Region dominance checking

881.00 ms	2.2%	881.00 ms	std::__1::__shared_weak_count::lock()	libc++.1.dylib
830.00 ms	2.0%	830.00 ms	__psynch_rw_wrlock	libsystem_kernel.dylib
750.00 ms	1.8%	750.00 ms	__psynch_rw_unlock	libsystem_kernel.dylib
732.00 ms	1.8%	732.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<llvm::PointerUnion<m	
656.00 ms	1.6%	656.00 ms	llvm::DenseMapInfo<llvm::PointerUnion<mlir::Operation*, mlir::Blc	
613.00 ms	1.5%	613.00 ms	__psynch_mutexwait	libsystem_kernel.dylib
450.00 ms	1.1%	450.00 ms	(anonymous namespace)::GreedyPatternRewriteDriver::addSingle	
433.00 ms	1.0%	433.00 ms	__psynch_mutexdrop	libsystem_kernel.dylib
397.00 ms	1.0%	397.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<mlir::Value, mlir::data	
392.00 ms	0.9%	392.00 ms	_nanov2_free	libsystem_malloc.dylib
331.00 ms	0.8%	331.00 ms	llvm::SmallVectorTemplateBase<llvm::PointerUnion<mlir::Operatic	
315.00 ms	0.7%	315.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<llvm::PointerUnion<m	
314.00 ms	0.7%	314.00 ms	_platform_memcmp	libsystem_platform.dylib
281.00 ms	0.7%	281.00 ms	char* llvm::hashing::detail::hash_combine_recursive_helper::com	
259.00 ms	0.6%	259.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<mlir::Operation*, unsi	
251.00 ms	0.6%	251.00 ms	llvm::ilist_detail::SpecificNodeAccess<llvm::ilist_detail::node_opti	
232.00 ms	0.5%	232.00 ms	_platform_memmove	libsystem_platform.dylib
209.00 ms	0.5%	209.00 ms	void mlir::detail::walk<mlir::ForwardIterator>(mlir::Operation*, llvm	
200.00 ms	0.5%	200.00 ms	nanov2_malloc	libsystem_malloc.dylib
200.00 ms	0.5%	200.00 ms	free	libsystem_malloc.dylib
199.00 ms	0.5%	199.00 ms	mlir::Operation::getAttrDictionary()	kgen
195.00 ms	0.4%	195.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<std::__1::pair<M::KGI	
195.00 ms	0.4%	195.00 ms	llvm::detail::PunnedPointer<mlir::Region*>::asInt() const [inlined]	
183.00 ms	0.4%	183.00 ms	llvm::hashing::detail::hash_short(char const*, unsigned long, unsi	
172.00 ms	0.4%	172.00 ms	mlir::detail::TypeIDResolver<mlir::OpTrait::ZeroRegions<mlir::Type	
157.00 ms	0.3%	157.00 ms	madvise	libsystem_kernel.dylib
153.00 ms	0.3%	153.00 ms	mlir::PatternApplicator::matchAndRewrite(mlir::Operation*, mlir::P	
152.00 ms	0.3%	152.00 ms	mlir::Region::isProperAncestor(mlir::Region*)	kgen
149.00 ms	0.3%	149.00 ms	mlir::detail::InterfaceMap::lookup(mlir::TypeID) const [inlined]	kg
147.00 ms	0.3%	147.00 ms	std::__1::__shared_count::__release_shared[abi:v15006]() [inline	
140.00 ms	0.3%	140.00 ms	nanov2_find_block_and_allocate	libsystem_malloc.dylib
137.00 ms	0.3%	137.00 ms	(anonymous namespace)::SimpleOperationInfo::isEqual(mlir::Ope	
136.00 ms	0.3%	136.00 ms	mlir::detail::TypeIDResolver<mlir::OpTrait::ZeroSuccessors<mlir::T	
135.00 ms	0.3%	135.00 ms	mlir::detail::TypeIDResolver<mlir::OpTrait::OpInvariants<mlir::Type	
132.00 ms	0.3%	132.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<std::__1::pair<unsign	
132.00 ms	0.3%	132.00 ms	mach_continuous_time	libsystem_kernel.dylib
132.00 ms	0.3%	132.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<std::__1::pair<M::KGI	
130.00 ms	0.3%	130.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<mlir::Value, unsigned	
130.00 ms	0.3%	130.00 ms	(anonymous namespace)::GreedyPatternRewriteDriver::addToWo	
130.00 ms	0.3%	130.00 ms	mlir::TypeRange::dereference_iterator(llvm::PointerUnion<mlir::Va	
128.00 ms	0.3%	128.00 ms	free_small	libsystem_malloc.dylib
126.00 ms	0.3%	126.00 ms	mlir::WalkResult mlir::detail::walk<mlir::ForwardIterator>(mlir::Ope	
121.00 ms	0.3%	121.00 ms	mlir::Operation::getDiscardableAttrDictionary()	kgen
120.00 ms	0.3%	120.00 ms	mlir::TypeID mlir::TypeID::get<mlir::OpTrait::ZeroRegions<mlir::Ty	
120.00 ms	0.3%	120.00 ms	(anonymous namespace)::GreedyPatternRewriteDriver::processW	
120.00 ms	0.3%	120.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<mlir::Value, mlir::Valu	
117.00 ms	0.2%	117.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<llvm::PointerUnion<m	
114.00 ms	0.2%	114.00 ms	mlir::DominanceInfo::properlyDominatesImpl(mlir::Operation*, mli	
113.00 ms	0.2%	113.00 ms	void llvm::SmallVectorImpl<mlir::NamedAttribute>::append<llvm::	
113.00 ms	0.2%	113.00 ms	bool llvm::DenseMapBase<llvm::DenseMap<mlir::Operation*, llvm	
112.00 ms	0.2%	112.00 ms	_platform_memset	libsystem_platform.dylib

Memory Footprint Analysis

Run a large Mojo program through to LLVMDialect

- Measure peak IR size, final StorageUniquer allocation size, reachable* StorageUniquer objects

Example: Matmul for “top model” shapes (6.2 million ops)

- 1100 MB peak IR size, 90.3 MB StorageUniquer, 5.4 MB reachable objects

Case Study: DebugInfo

(Yesterday's talk: MLIR DebugInfo in Mojo)

- 272 MB total, 115 MB reachable objects

DebugInfo uses ~110 MB of StorageUniquer memory

- 4010 unique FileLineColLoc
- 1239171 unique CallSiteLoc
- 75 MB of metadata when exported to LLVMIR

Case Study: DebugInfo

Test a variety of programs

# of ops	Peak MLIR op size (MB)	Total StorageUniquer size (MB)	Reachable storage (MB)	% unreachable storage	% of LLVM DI size
16835	4.1	9.5	0.76	92	70.2
83410	20.4	16.7	3.07	81.6	64.3
102703	25.1	9.7	1.68	82.7	58.1
508064	124	34.6	8.62	75.1	64
542753	132.5	44.9	11.9	73.5	58.4
6200000	1513.7	271.5	115.1	57.6	65.5

Other Performance Considerations

Underdeveloped Analysis Preservation

- Raise your hand if you have used the AnalysisManager
- MLIR lacks established patterns for preserving fine-grained analyses across passes
  - SymbolTable, CallGraph, AliasAnalysis, DominanceInfo, memory analyses...
- Leads to monolithic pass design, or frequent recomputations

```
class PreservedAnalyses {
public:
  /// Mark all analyses as preserved.
  void preserveAll() {
   preservedIDs.insert(TypeID::get<AllAnalysesType>());
  }

  /// Returns true if all analyses were marked preserved.
  bool isAll() const {
    return preservedIDs.count(TypeID::get<AllAnalysesType>());
  }

  /// Preserve the given analyses.
  template <typename AnalysisT>
  void preserve() {
    preserve(TypeID::get<AnalysisT>());
  }
```

Legacy Dialect Design

- Anti-patterns in frequently-used upstream dialects
  - E.g. LLVMDialect
- Start at the egress dialects and push through best practices

```
def LLVM_GlobalDtorsOp : LLVM_Op<"mlir.global_dtors"> {
  let arguments = (ins
    FlatSymbolRefArrayAttr:$dtors,
    I32ArrayAttr:$priorities
  );
```

```
if (auto *structType = dyn_cast<::llvm::StructType>(llvmType)) {
  auto arrayAttr = dyn_cast<ArrayAttr>(attr);
  if (!arrayAttr || arrayAttr.size() != 2)
    emitError(loc, "expected struct type to be a complex number");
```

```
if (llvm::Attribute::isIntAttrKind(kind)) {
  if (value.empty())
    return emitError(loc) << "LLVM attribute '" << key
                          << "' expects a value";
  int64_t result;
  if (!value.getAsInteger(/*Radix=*/0, result))
    llvmFunc->addFnAttr(
        llvm::Attribute::get(llvmFunc->getContext(), kind, result));
```

ODS

- C++ code generated by ODS matters
- llvm-project/#87741 makes ODS generate getters inline
- (legacy) optimizations to attribute getters and verifiers
- Constraint deduplication / outlining
- *build* method optimization

```
auto namedAttrRange = (*this)->getAttrs();
auto namedAttrIt = namedAttrRange.begin();
Attribute tblgen_decorators;
Attribute tblgen_metadata;

while (true) {
  if (namedAttrIt == namedAttrRange.end())
    return emitOpError("requires attribute 'decorators'");
  if (namedAttrIt->getName() == getDecoratorsAttrName()) {
    tblgen_decorators = namedAttrIt->getValue();
    break;
  } else if (namedAttrIt->getName() == getMetadataAttrName()) {
    tblgen_metadata = namedAttrIt->getValue();
  }
  ++namedAttrIt;
}
Attribute tblgen_kind;
Attribute tblgen_docString;

while (true) {
  if (namedAttrIt == namedAttrRange.end())
    return emitOpError("requires attribute 'kind'");
  if (namedAttrIt->getName() == getKindAttrName()) {
    tblgen_exportKind = namedAttrIt->getValue();
    break;
  } else if (namedAttrIt->getName() == getDocStringAttrName()) {
    tblgen_docString = namedAttrIt->getValue();
  }
  ++namedAttrIt;
}
```

Dialect Resources

- Data not managed by MLIRContext
- Uniquing large data that may become unreachable is incredibly inefficient
- mmap'd from MLIR bytecode – significantly faster loading

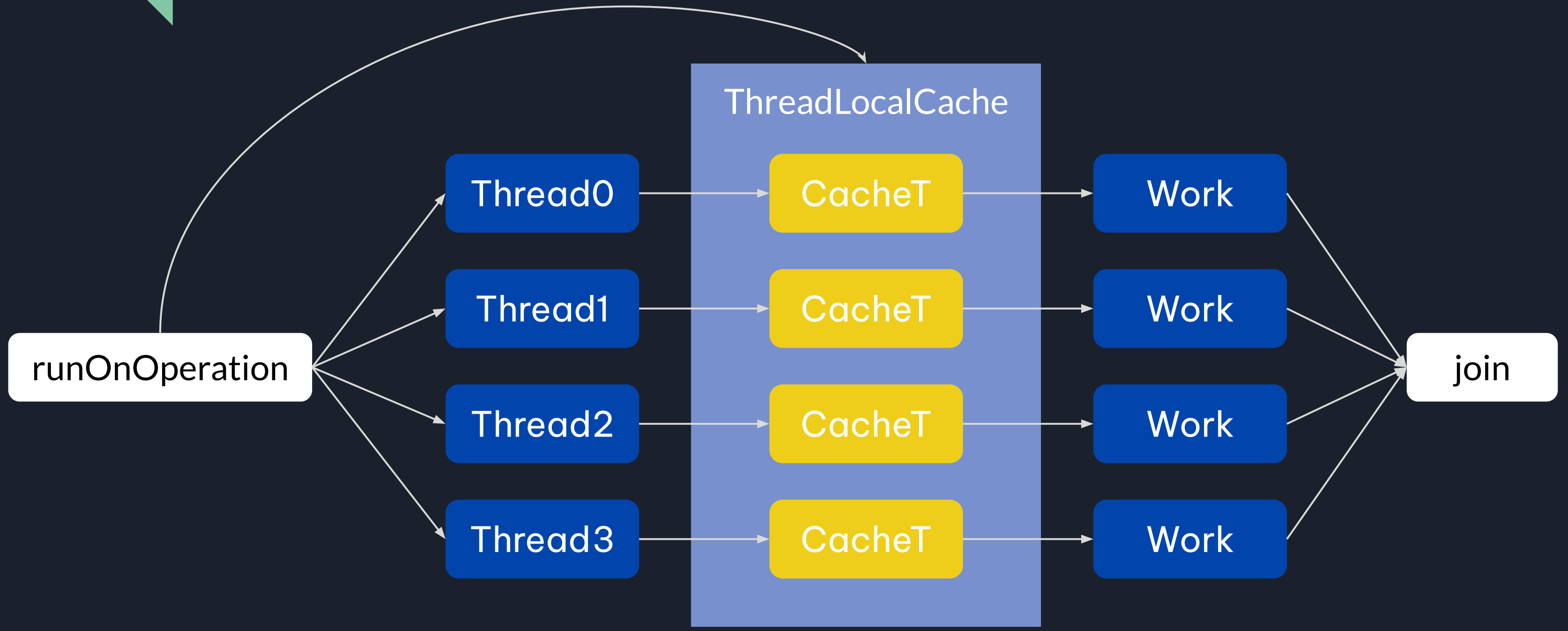
```
func @big_constant() -> tensor<6710x100x100xi64> {
  %cst = arith.constant dense<[...]>
    : tensor<6710x100x100xi64>
  return %cst : tensor<6710x100x100xi64>
}

func @big_constant() -> tensor<6710x100x100xi64> {
  %cst = arith.constant dense_resource<big_constant>
    : tensor<6710x100x100xi64>
   return %cst : tensor<6710x100x100xi64>
}
```

[Blog: Increasing Development Velocity of Giant AI Models]

Why Parallelism isn't Free

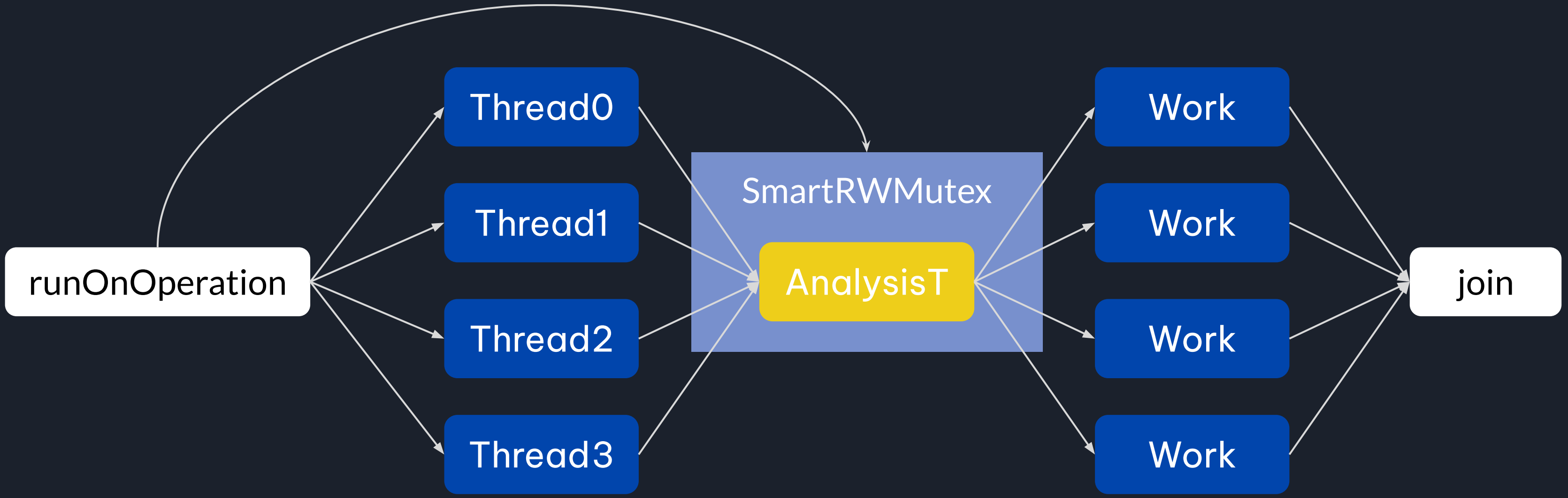
- MLIR supports parallelism, but it's not always a straightforward win
- Parallelism overheads:
    - The usual suspects: launching threads, runtime, etc.
    - StorageUniquer
    - Memory allocator (tcmalloc)
    - And more…
- *Some* MLIR workloads are perfectly parallelizable

Intra-Pass Parallelism



Sharding

Intra-Pass Parallelism



Sharing

Why Parallelism isn't Free

- Sharding work X always introduces per-shard overhead C
  - Increases compile speed of a single compilation unit to X/N + C
  - **Total work** performed by compiler is X + **N*C** (more power and CPU time used)
- *Build system* may not be aware of compiler tool parallelism
  - -j N spawns N compiler processes: N*X + **(N^2)*C**
  - If compiler process were single-threaded: N*X
  - Build system parallelism + compiler parallelism is *slower*

Example: Matmul kernels from earlier

- 24.4 seconds single-threaded
- 11.3 seconds multi-threaded (49.7 seconds CPU time)
- 2x faster but 2x more CPU time
  - Some overhead is mutex contention, a portion of which is given back to the OS
- Single compilation unit speed matters: model compilation, max latency, etc.

Choking Under Threads

N compiler processes each with N threads in a threadpool means N^2 active threads (imagine N=192 on some AWS machines)

Compiling Modular Top Models:

Arch	Threading Enabled (sec)	Single-Threaded (sec)
Graviton (m6g)	947.14	844.17
Intel (m6i)	2733.18	427.31
AMD (m6a)	2029.63	424.64

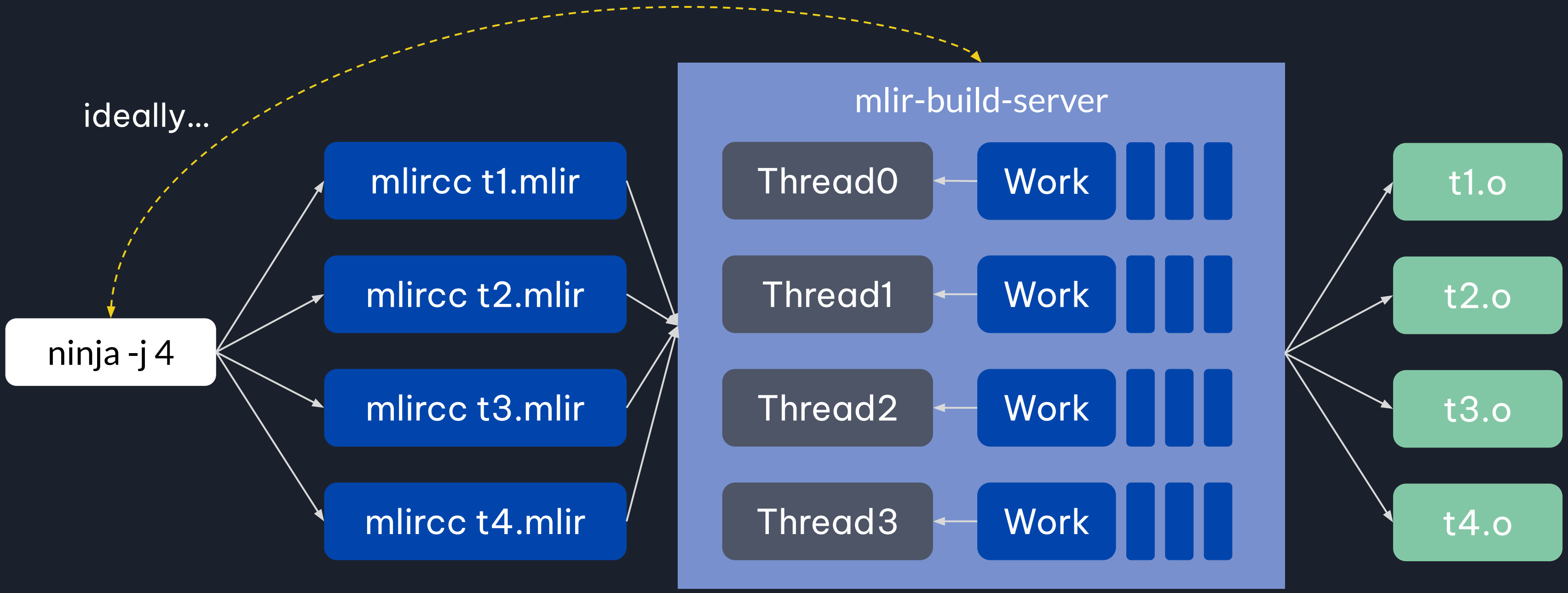
intel

amd

graviton

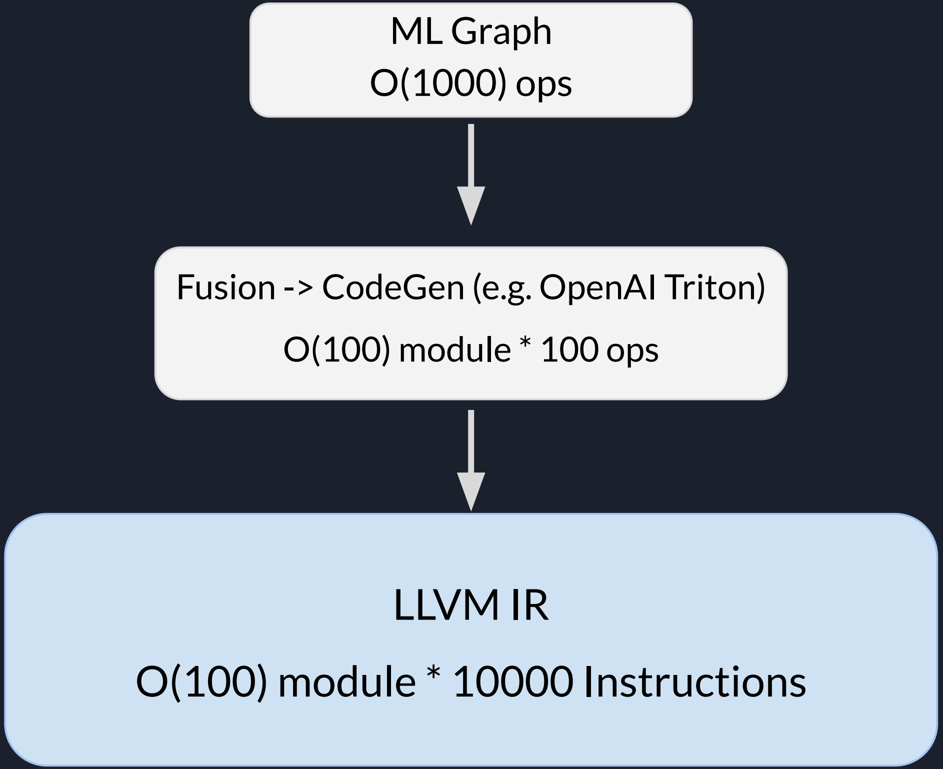
Solution?

Build system and compiler have to talk to each other



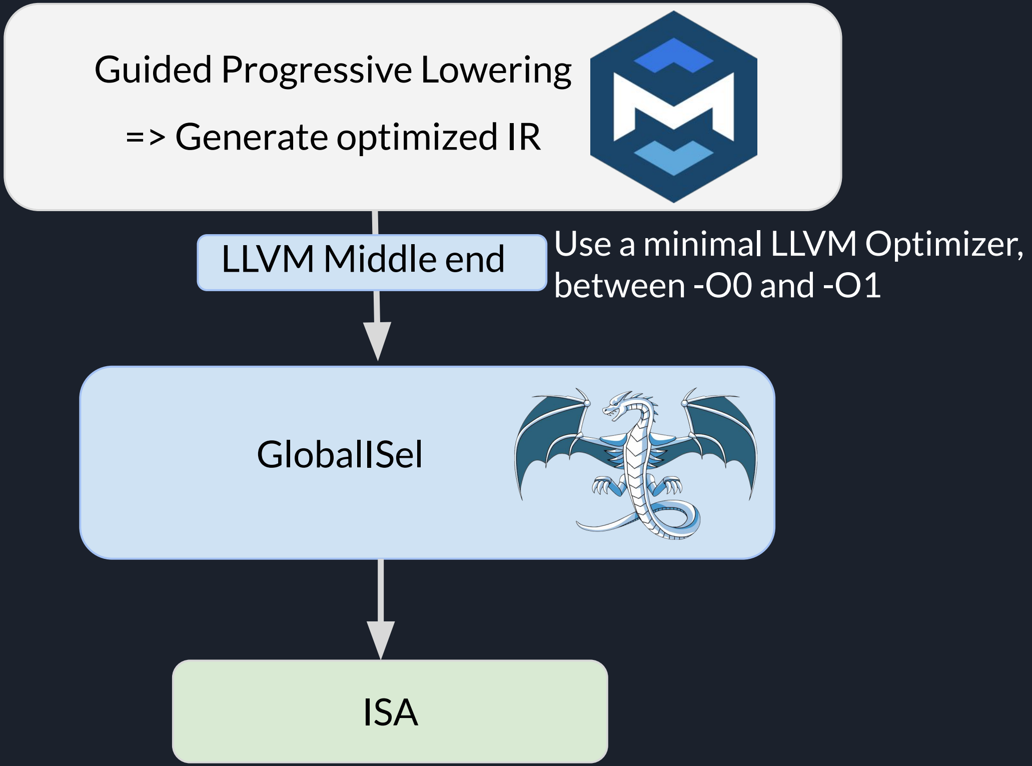
Performance Footguns and Anti-Patterns

- Symbol table methods: `lookupSymbolIn` and `lookupNearestSymbol`
  - *O(N)* methods! Walks around the IR looking at attributes
  - `ModuleOp::lookupSymbol`
- Using applyGreedyPatternRewriter with 1 pattern
- Attributes
- Running PassManager until fixed point
- DialectConversion is typically overkill (rollback support)



Inverted Funnel: a small number of high-level constructs generate a lot of LLVM instructions.
=> small constant overheads in MLIR are more than compensated by the size explosion that LLVM has to handle ; the majority of the time is spent in LLVM already!

Possible future for MLIR CodeGen



Good practices

- Cache Type/Attr instances in *Pass::initialize* hooks
- Cache commonly used types on *Dialect* instances
- Re-use OpInterface instances
- Be conscientious of sugared ODS *build* methods
- Verifier: don’t always run verify-after-all
- Canonicalize: GreedyRewriteDriver is heavy
- Use native operation properties
- Specialize and harden passes as your compiler stack matures

Future Work for MLIR

- Investigate worst performers in microbenchmarks (StorageUniquer…)
- Migrate more upstream dialects to Native properties as first-class constructs
- Revamp the constant handling to avoid attributes
- Lightweight “one-shot” drivers for Dialect Conversion and Pattern application
- Explore advanced data structures: *std::deque* variation that trades random access for O(1) insertion/deletion everywhere, *std::vector* with poison entries (re-read: https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/)
- Introduce parent Block pointer indirection on Operation for O(1) splice
- Drop pImpl pattern / out-of-line Attribute and Type Storage classes

Thank you!