Bringing up GlobalISel for optimized AArch64 codegen

Amara Emerson, Jessica Paquette - Apple

A quick recap…

What’s GlobalISel?

- A new instruction selection framework, replacing FastISel & SelectionDAG
- Most notable difference is visibility of the entire function
  - SelectionDAG is limited to analyzing a single block at a time
- Other key design goals:
  - Customizability for targets to adapt to their specific needs
  - Compile time improvements

AArch64 GlobalISel backend

Project progress since COVID started

- GlobalISel was enabled for AArch64 at -O0 in 2018.
- Then attention shifted to the next ~~big~~ gigantic milestone: enabling at all optimization levels.
- Much more difficult than -O0, because:
    - IR input to codegen is extremely diverse
    - SelectionDAG is highly optimized
    - Small differences in performance/code size **matter**

Performance metrics

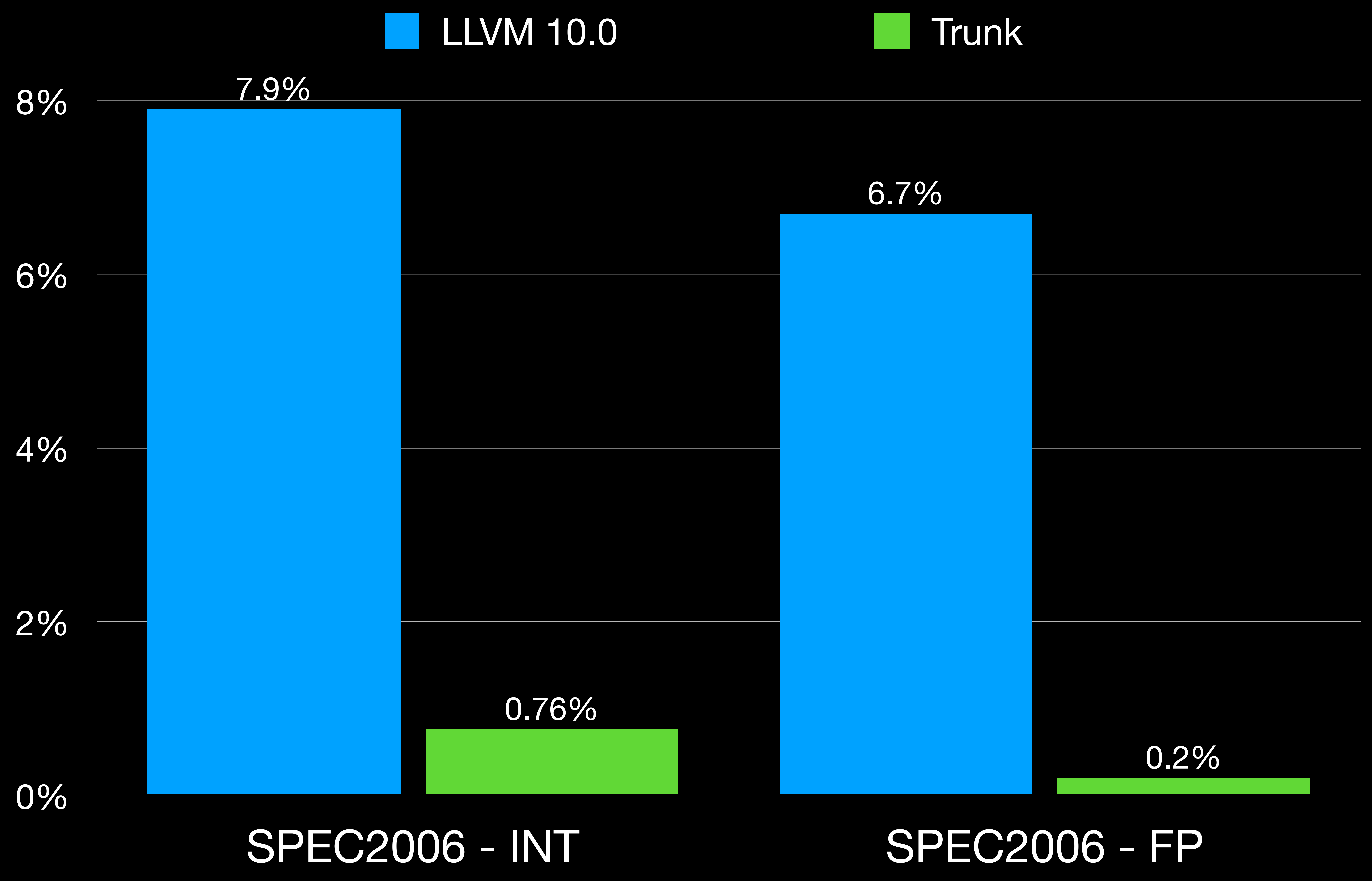
Performance metrics

How do we measure progress?

- Runtime performance (measured on Apple iOS devices)
  - SPEC2006, SPEC2017, Geekbench 5, test-suite, internal benchmarks
- Code size (-Os, -Oz)
  - CTMark, internal projects
- Compile time (-Os, -O3)
  - CTMark

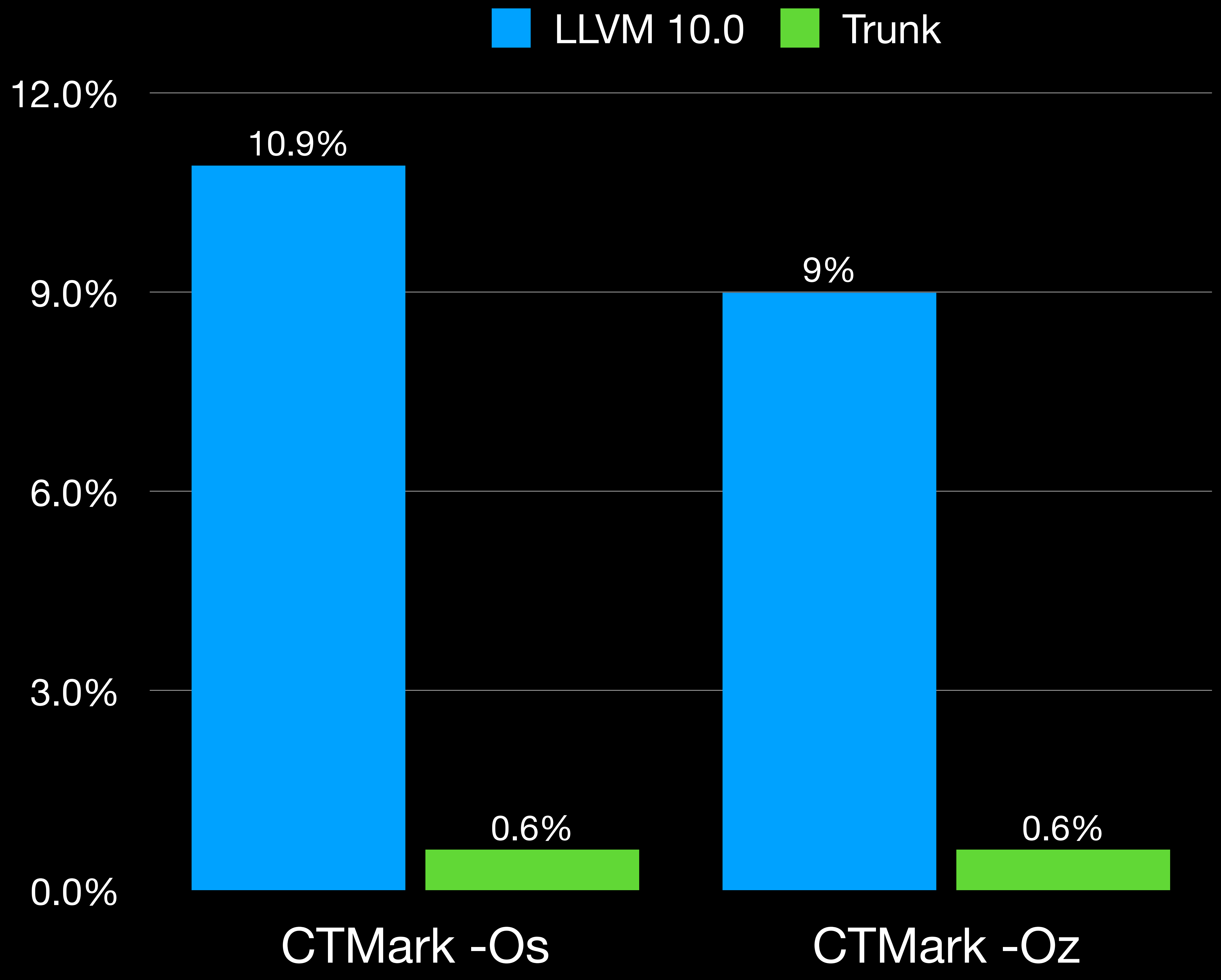
Performance metrics

Runtime geomean regression vs SelectionDAG



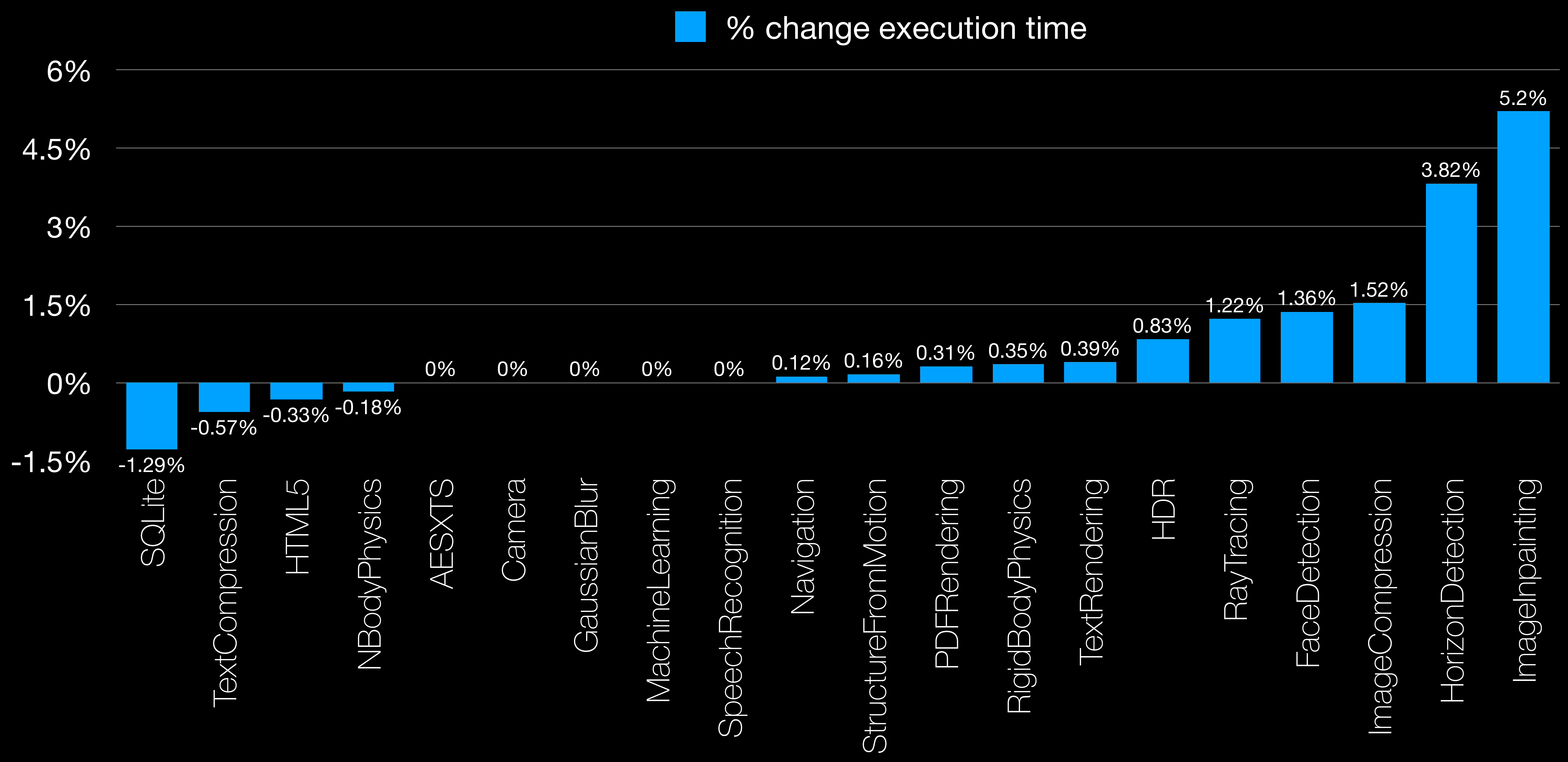
Performance metrics

Code size geomean regression vs SelectionDAG



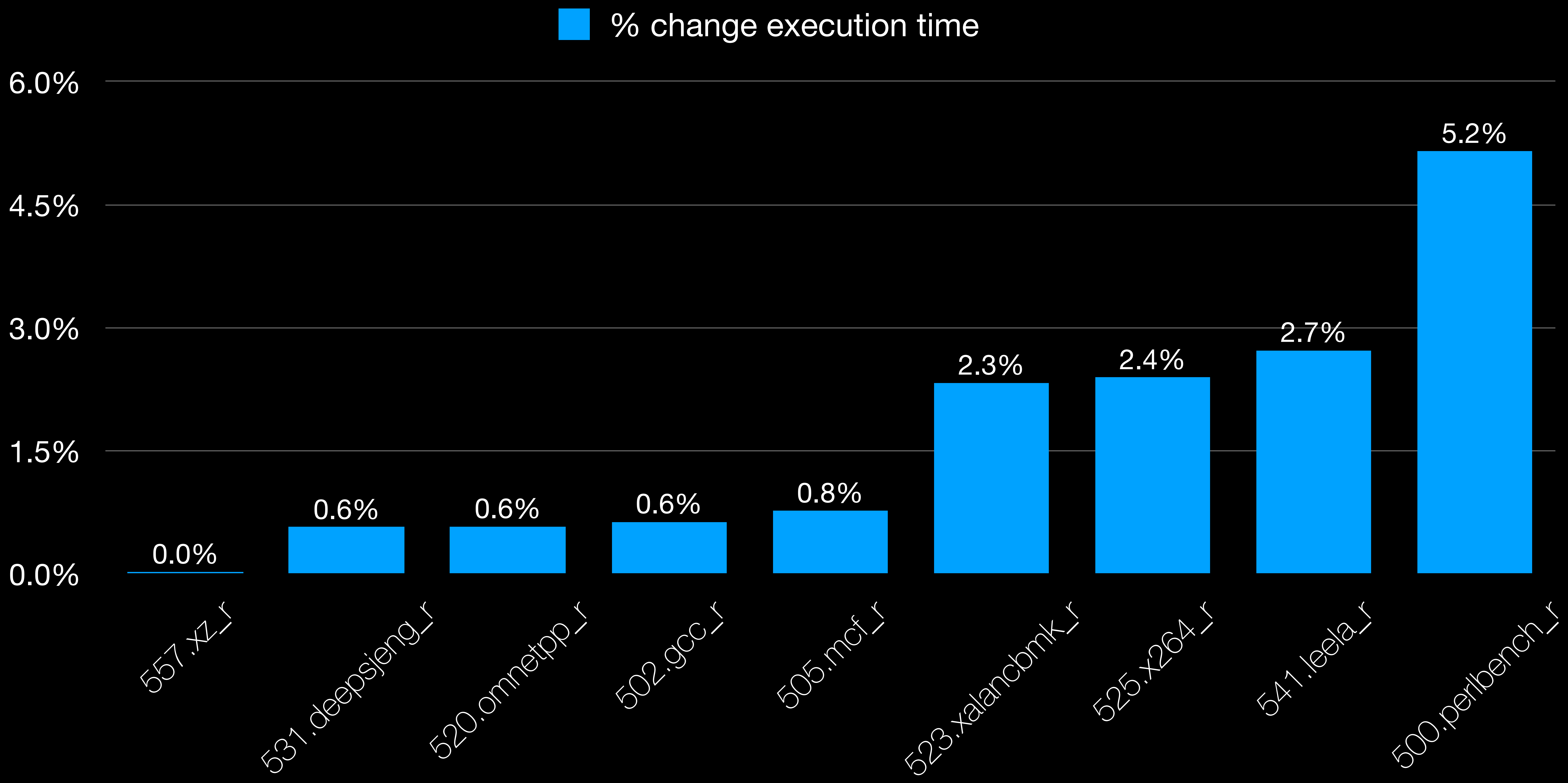
Performance metrics

Runtime geomean vs SelectionDAG: Geekbench - single threaded



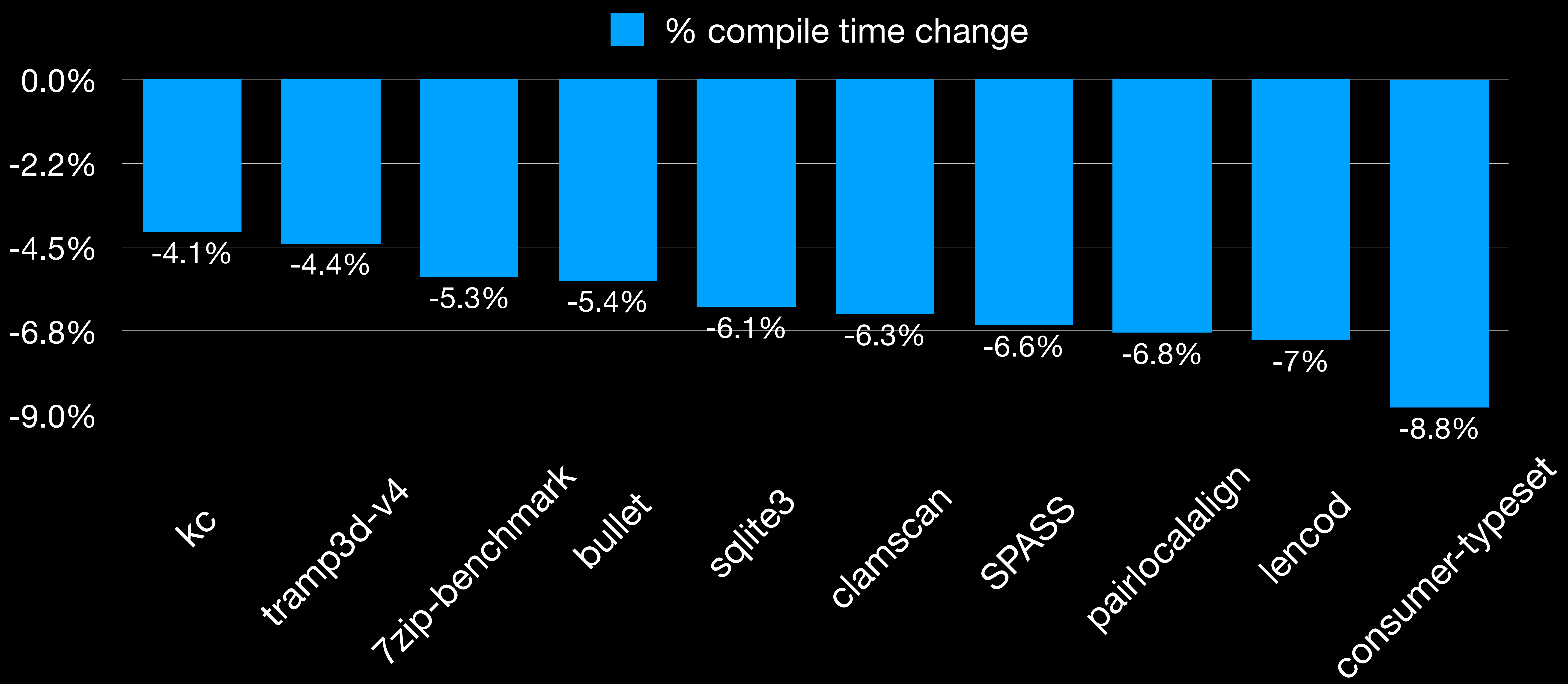
Performance metrics

Runtime geomean vs SelectionDAG: SPECINT2017 - INT



Performance metrics

ToT compile time vs SelectionDAG: CTMark -Os



Geomean improvement: **6.1%**

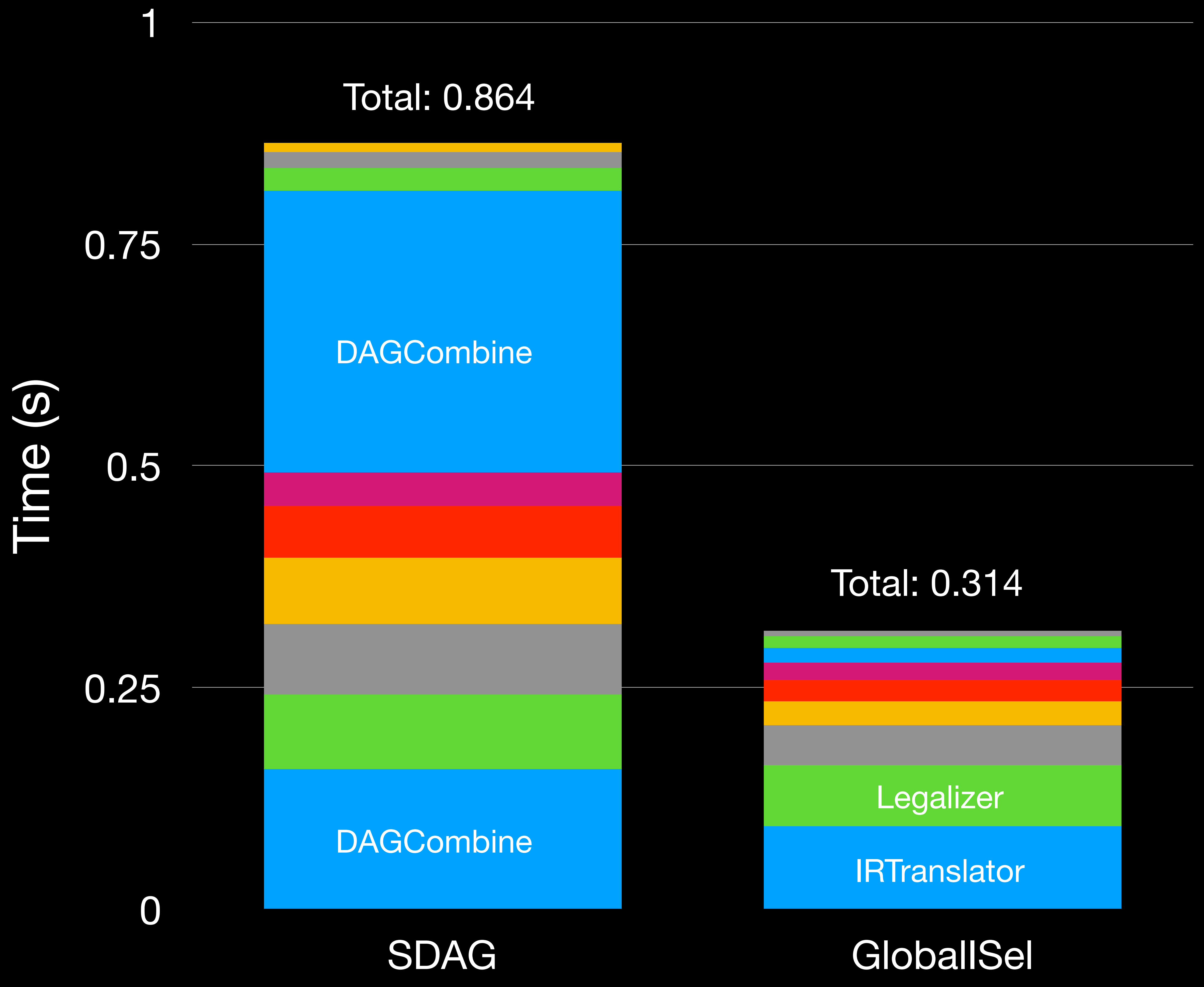
Performance metrics

SelectionDAG-only vs GlobalISel-only compile time: sqlite3

- Overall compile time improvement is nice, but front-end, IR opts, post-isel pipeline affect relative speedup.
- Isolating instruction selection only, how much faster is GlobalISel?
  - We include all combiners, legalizer code etc…

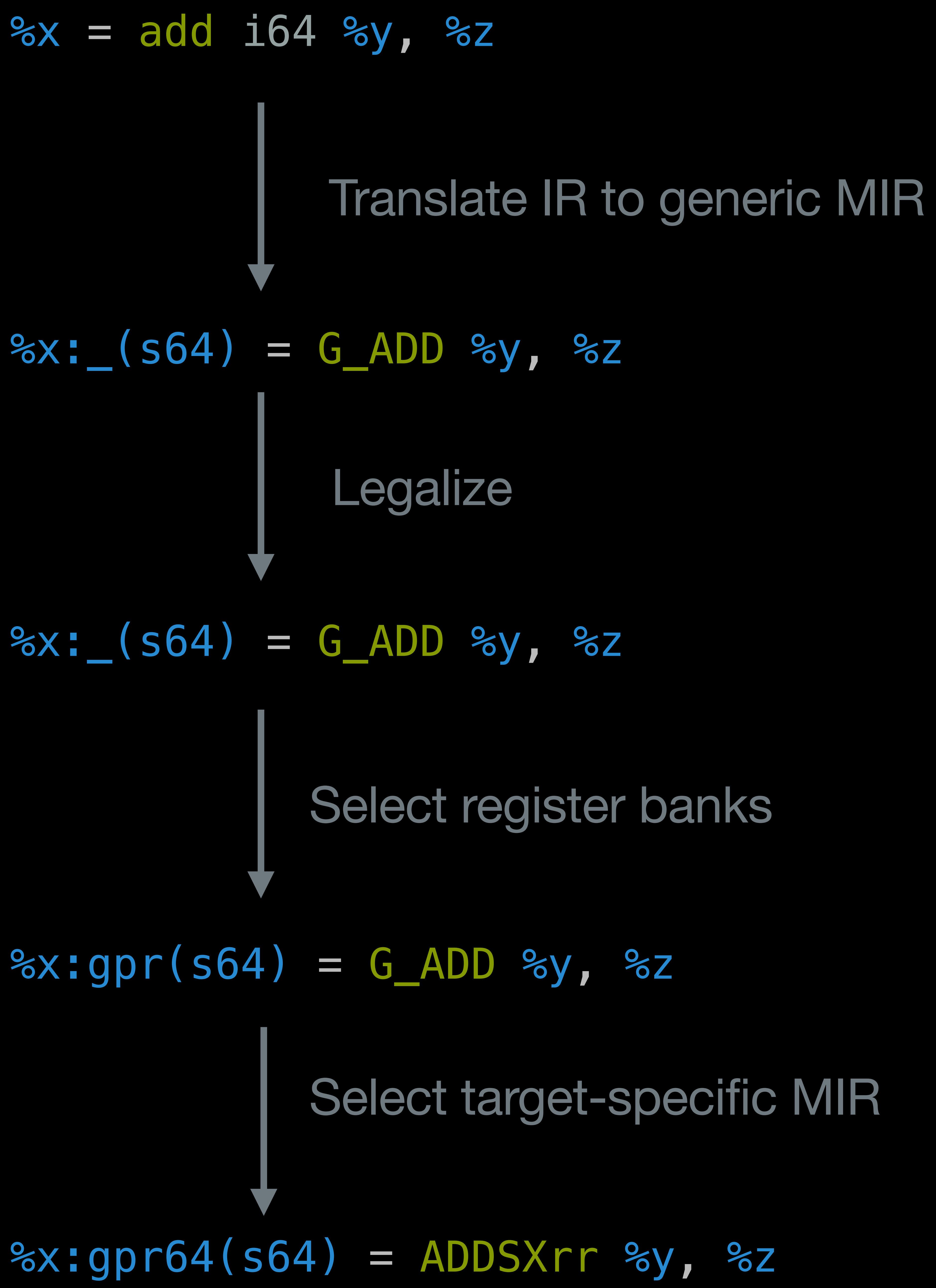
Performance metrics

SelectionDAG-only vs GlobalISel-only compile time: sqlite3 -Os



2.7x faster 🎉🎉🎉

Life of a GlobalISel instruction



```
%x = add i64 %y, %z
```

Translate IR to generic MIR

```
%x:_(s64) = G_ADD %y, %z
```

Legalize

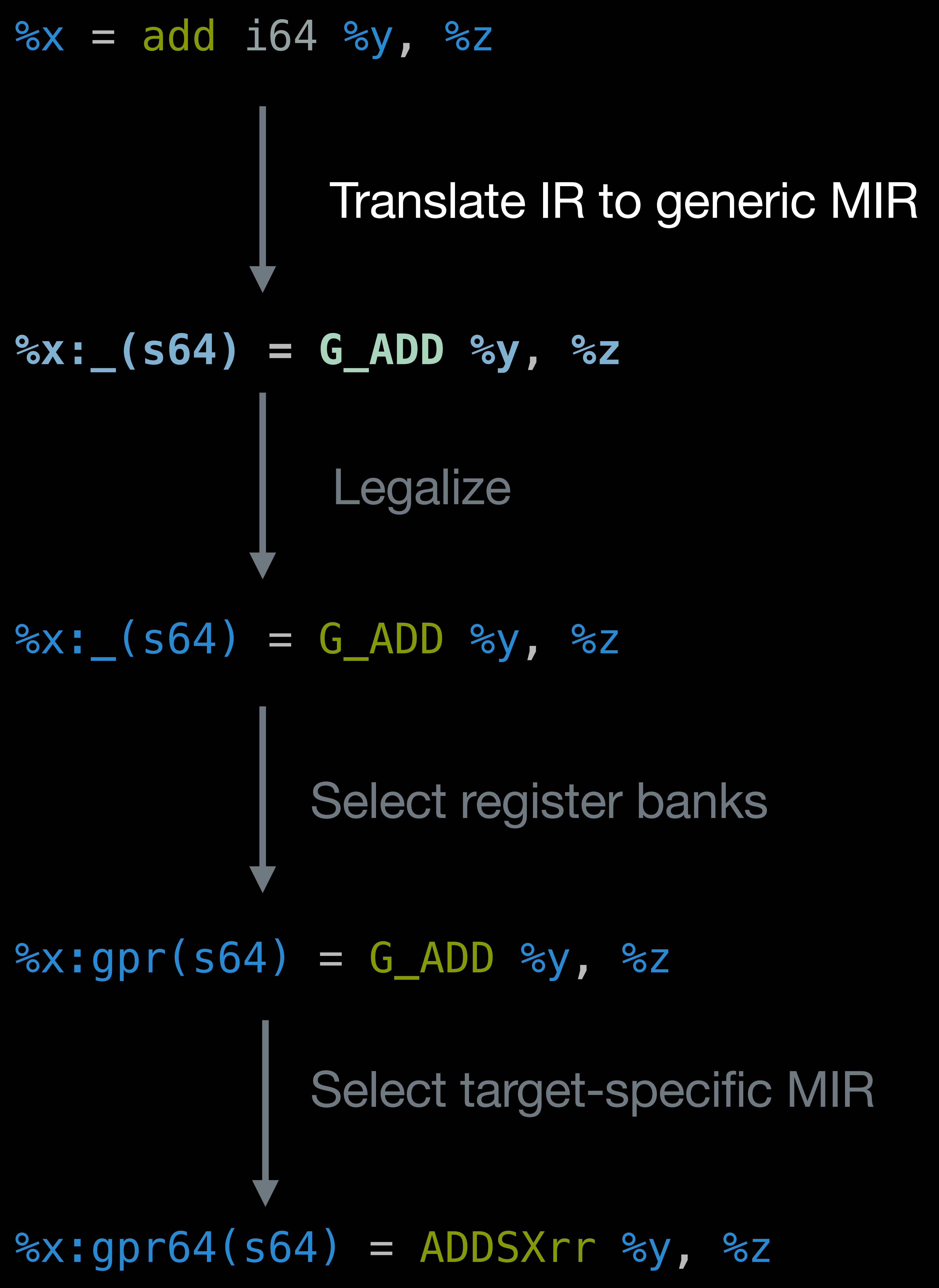
```
%x:_(s64) = G_ADD %y, %z
```

Select register banks

```
%x:gpr(s64) = G_ADD %y, %z
```

Select target-specific MIR

```
%x:gpr64(s64) = ADDSXrr %y, %z
```



```
%x = add i64 %y, %z
```

Translate IR to generic MIR

```
%x:_(s64) = G_ADD %y, %z
```

Legalize

```
%x:_(s64) = G_ADD %y, %z
```

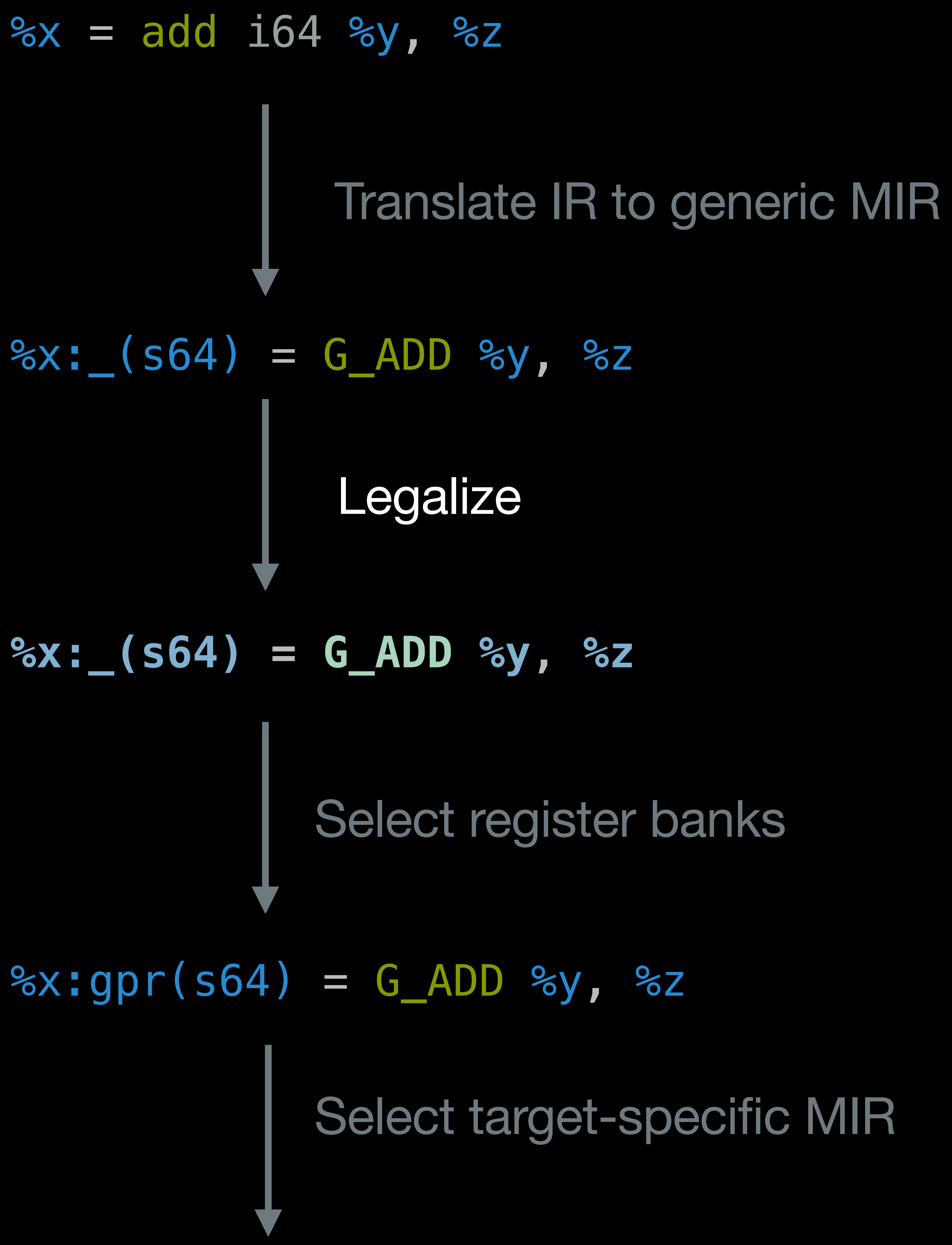
Select register banks

```
%x:gpr(s64) = G_ADD %y, %z
```

Select target-specific MIR

```
%x:gpr64(s64) = ADDSXrr %y, %z
```

IR Translation: translate LLVM IR into gMIR instructions



```
%x = add i64 %y, %z
```

Translate IR to generic MIR

```
%x:_(s64) = G_ADD %y, %z
```

Legalize

```
%x:_(s64) = G_ADD %y, %z
```

Select register banks

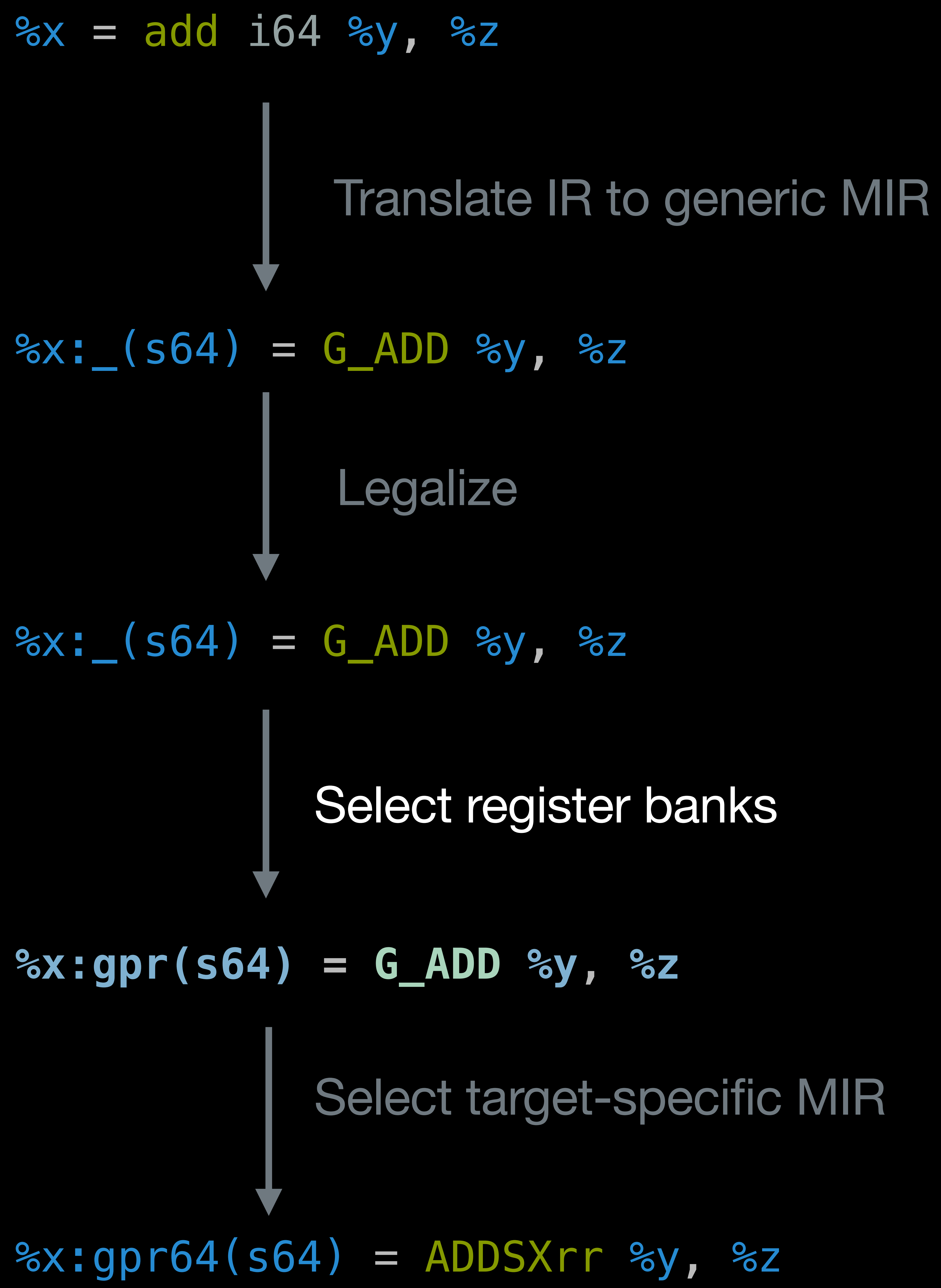
```
%x:gpr(s64) = G_ADD %y, %z
```

Select target-specific MIR

```
%x:gpr64(s64) = ADDSXrr %y, %z
```

Legalization: mutate instruction using rules until it’s “legal”.

Legal in GlobalISel means that it’s a valid, selectable instruction for the target.



```
%x = add i64 %y, %z
```

Translate IR to generic MIR

```
%x:_(s64) = G_ADD %y, %z
```

Legalize

```
%x:_(s64) = G_ADD %y, %z
```

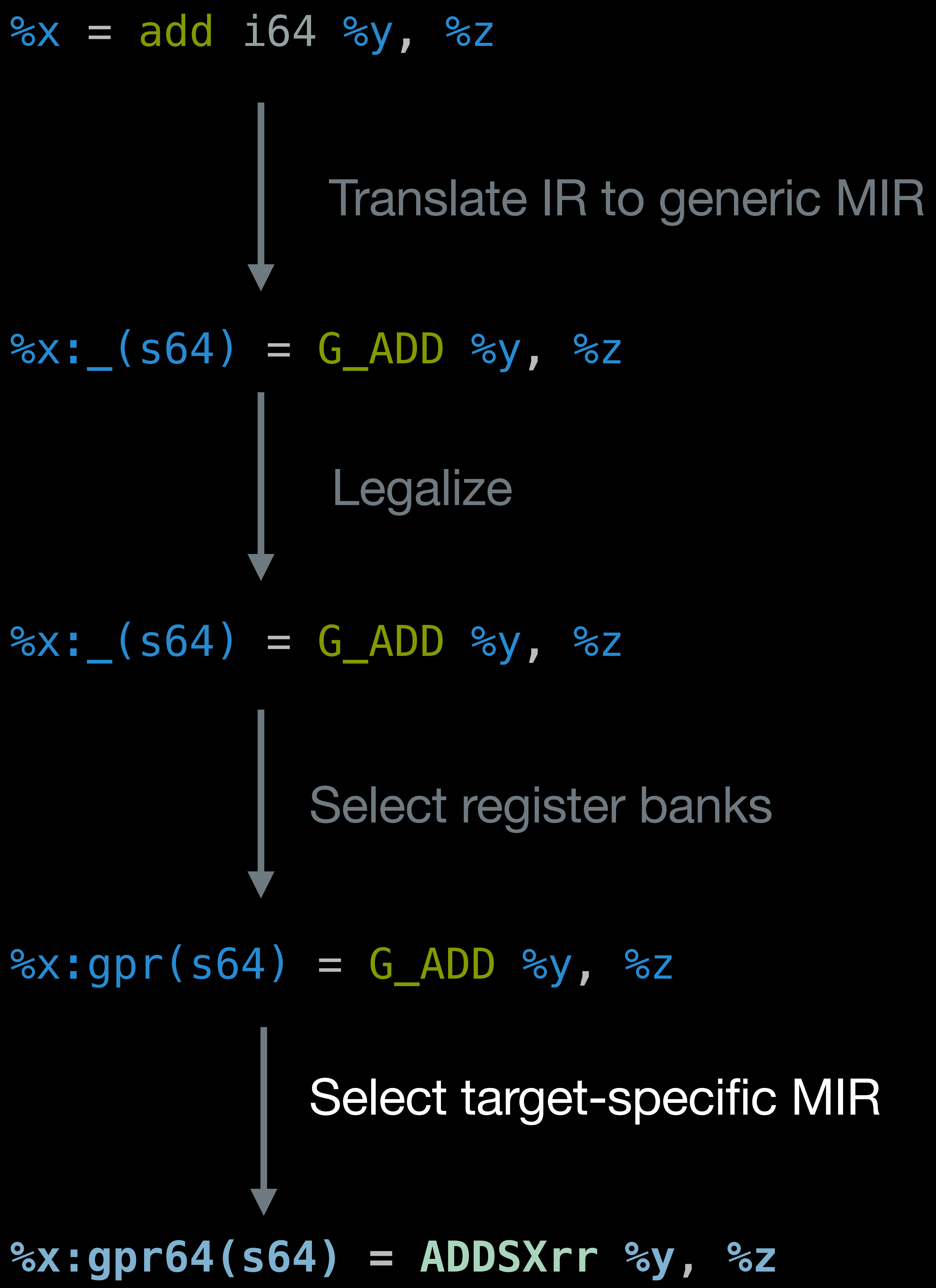
Select register banks

```
%x:gpr(s64) = G_ADD %y, %z
```

Select target-specific MIR

```
%x:gpr64(s64) = ADDSXrr %y, %z
```

RegBankSelect: assigns register banks, as defined by the target, to minimize cost of cross-bank copies.



```
%x = add i64 %y, %z
```

Translate IR to generic MIR

```
%x:_(s64) = G_ADD %y, %z
```

Legalize

```
%x:_(s64) = G_ADD %y, %z
```

Select register banks

```
%x:gpr(s64) = G_ADD %y, %z
```

Select target-specific MIR

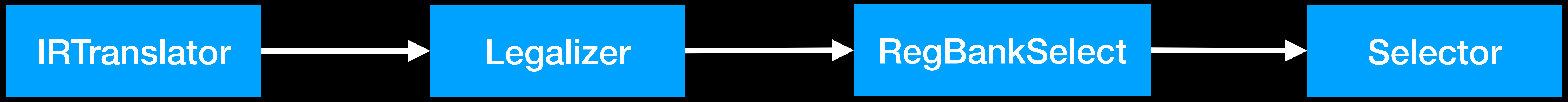
```
%x:gpr64(s64) = ADDSXrr %y, %z
```

Final instruction selection pass. Matches patterns of gMIR into target instructions.

AArch64 optimization pipeline

AArch64 optimization pipeline

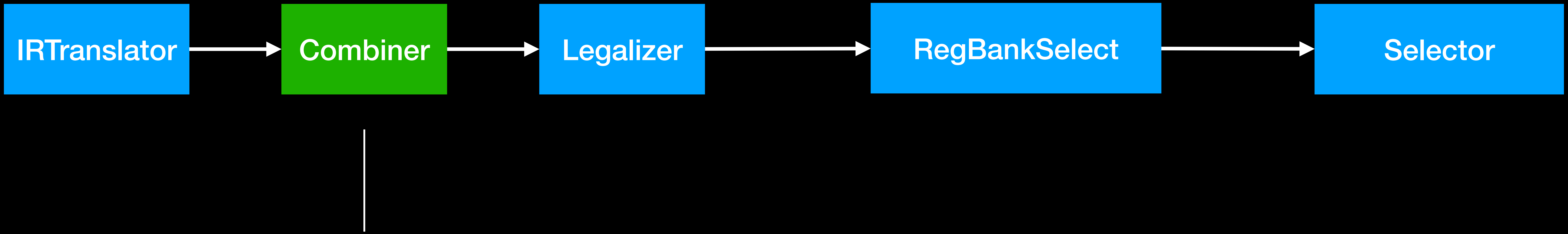
Where should we place optimizations?



IRTranslator → Legalizer → RegBankSelect → Selector

AArch64 optimization pipeline

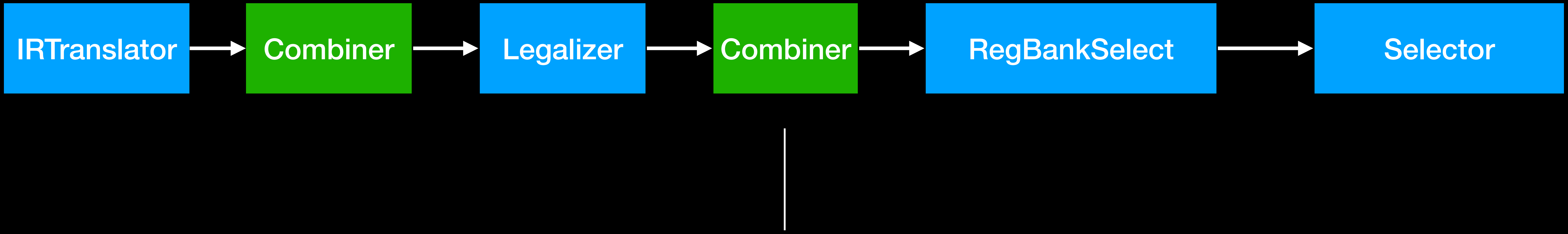
The first combine pass



We call this the *Pre-Legalizer Combiner*

AArch64 optimization pipeline

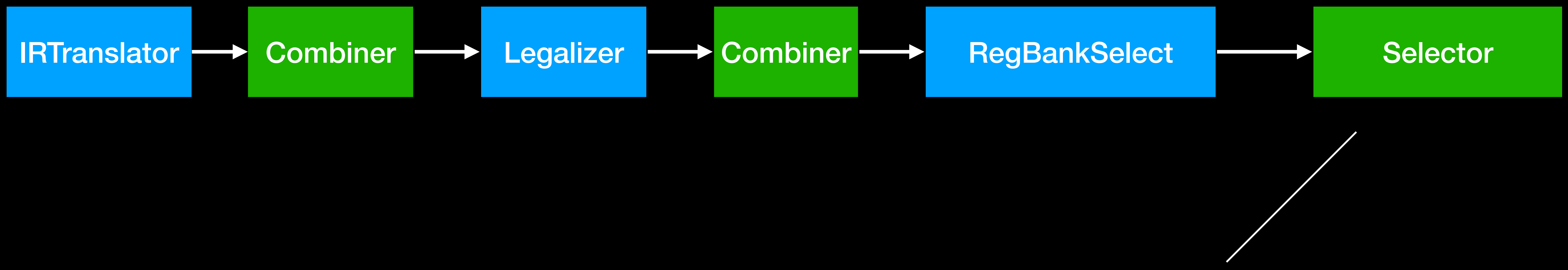
The second combine pass



We call this the *Post-Legalizer Combiner*

AArch64 optimization pipeline

Selection-time optimization

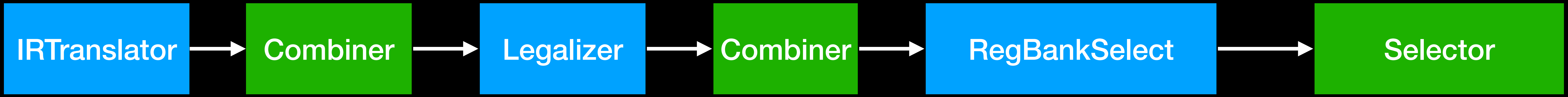


Manual optimizations implemented at selection-time

Lowering complex operations

Lowering complex operations

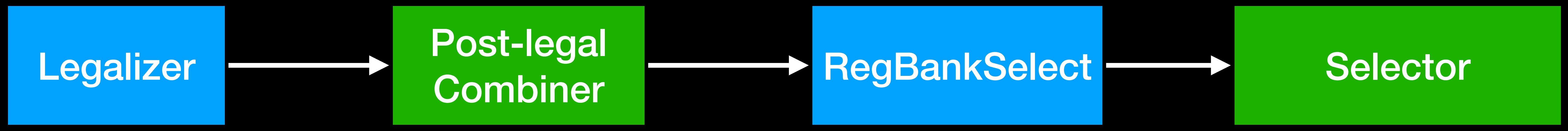
A closer look at lowering and selection



- Some operations, like G_SHUFFLE_VECTOR, can be complex to optimize and select.
- For AArch64 we don’t use custom legalization for shuffles.
- Let’s take a closer look at the post-legalizer pipeline…

Lowering complex operations

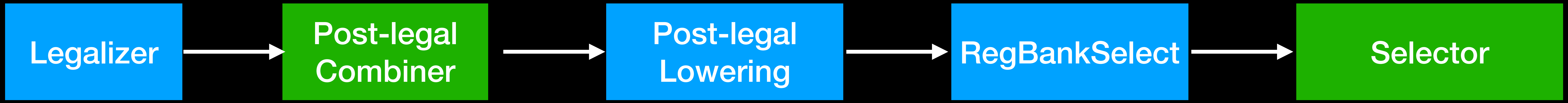
A closer look at lowering and selection



- If a shuffle remains intact through the post-legalizer combiner, then the selector is responsible for all remaining optimization/lowering.
- This is possible, but adds a lot of complexity to one pass.
  - The selector cannot easily create new Generic MachineInstrs.
- We decided to add a lowering pass instead.

Lowering complex operations

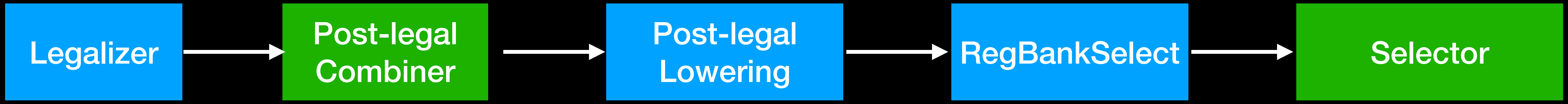
A closer look at lowering and selection



- Post-legalizer lowering runs at all optimization levels.
  - It can optimize, but also contains necessary transforms for selection.
- We essentially split the selection process over two passes.
- Let’s take a look at an example…

Lowering complex operations

Vector element reverse using G_SHUFFLE_VECTOR



- Our example is a G_SHUFFLE_VECTOR with a reverse element mask:

```
%shuf:_(<2 x s32>) = G_SHUFFLE_VECTOR %vec0(<2 x s32>), %vec1, shufflemask(1, 0)
```

```
%shuf:_(<2 x s32>) = G_SHUFFLE_VECTOR %vec0(<2 x s32>), %vec1, shufflemask(1, 0)
```

```
%shuf:_(<2 x s32>) = G_SHUFFLE_VECTOR %vec0(<2 x s32>), %vec1, shufflemask(1, 0)
```

```
def rev : GICombineRule<
  (defs root:$root, shuffle_matchdata:$matchinfo),
  (match (wip_match_opcode G_SHUFFLE_VECTOR):$root,
         [{ return matchREV(*${root}, MRI, ${matchinfo}); }]),
  (apply [{ applyShuffleVectorPseudo(*${root}, ${matchinfo}); }])
>;
```

```
%shuf:_(<2 x s32>) = G_SHUFFLE_VECTOR %vec0(<2 x s32>), %vec1, shufflemask(1, 0)
```

```
def rev : GICombineRule<
  (defs root:$root, shuffle_matchdata:$matchinfo),
  (match (wip_match_opcode G_SHUFFLE_VECTOR):$root,
         [{ return matchREV(*${root}, MRI, ${matchinfo}); }]),
  (apply [{ applyShuffleVectorPseudo(*${root}, ${matchinfo}); }])
>;
```

```
%shuf:_(<2 x s32>) = G_SHUFFLE_VECTOR %vec0(<2 x s32>), %vec1, shufflemask(1, 0)
```

```
static bool matchREV(MachineInstr &MI, MachineRegisterInfo &MRI,
                     ShuffleVectorPseudo &MatchInfo) {
  assert(MI.getOpcode() == TargetOpcode::G_SHUFFLE_VECTOR);
  ArrayRef<int> ShuffleMask = MI.getOperand(3).getShuffleMask();
  Register Dst = MI.getOperand(0).getReg();
  Register Src = MI.getOperand(1).getReg();
  LLT Ty = MRI.getType(Dst);
  unsigned EltSize = Ty.getScalarSizeInBits();
  // Element size for a rev cannot be 64.
  if (EltSize == 64)
    return false;
  unsigned NumElts = Ty.getNumElements();
  if (isREVMask(ShuffleMask, EltSize, NumElts, 64)) {
    MatchInfo = ShuffleVectorPseudo(AArch64::G_REV64, Dst, {Src});
    return true;
  }
  // ... G_REV32, G_REV64...
}
```

```
%shuf:_(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
static bool matchREV(MachineInstr &MI, MachineRegisterInfo &MRI,
                     ShuffleVectorPseudo &MatchInfo) {
  assert(MI.getOpcode() == TargetOpcode::G_SHUFFLE_VECTOR);
  ArrayRef<int> ShuffleMask = MI.getOperand(3).getShuffleMask();
  Register Dst = MI.getOperand(0).getReg();
  Register Src = MI.getOperand(1).getReg();
  LLT Ty = MRI.getType(Dst);
  unsigned EltSize = Ty.getScalarSizeInBits();
  // Element size for a rev cannot be 64.
  if (EltSize == 64)
    return false;
  unsigned NumElts = Ty.getNumElements();
  if (isREVMask(ShuffleMask, EltSize, NumElts, 64)) {
    MatchInfo = ShuffleVectorPseudo(AArch64::G_REV64, Dst, {Src});
    return true;
  }
  // ... G_REV32, G_REV64...
}
```

```
%shuf:_(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
def G_REV64 : AArch64GenericInstruction {
  let OutOperandList = (outs type0:$dst);
  let InOperandList = (ins type0:$src);
  let hasSideEffects = 0;
}
```

AArch64-only generic 64-bit rev

```
%shuf:_(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
def G_REV64 : AArch64GenericInstruction {
  let OutOperandList = (outs type0:$dst);
  let InOperandList = (ins type0:$src);
  let hasSideEffects = 0;
}
```

AArch64-only generic 64-bit rev

```
def : GINodeEquiv<G_REV64, AArch64rev64>;
```

Import Tablegen patterns for AArch64rev64

```
%shuf:fpr(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
%shuf:fpr(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
def : GINodeEquiv<G_REV64, AArch64rev64>;
```

```
%shuf:fpr(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
def : GINodeEquiv<G_REV64, AArch64rev64>;
```

Imported Tablegen rules for G_REV

```
(AArch64rev64:{*:[v2i32]} V64:{*:[v2i32]}:$Rn) => (REV64v2i32:{*:[v2i32]} V64:{*:[v2i32]}:$Rn)
(AArch64rev64:{*:[v2f32]} V64:{*:[v2f32]}:$Rn) => (REV64v2i32:{*:[v2f32]} V64:{*:[v2f32]}:$Rn)
(AArch64rev64:{*:[v4i16]} V64:{*:[v4i16]}:$Rn) => (REV64v4i16:{*:[v4i16]} V64:{*:[v4i16]}:$Rn)
...
```

```
%shuf:fpr(<2 x s32>) = G_REV64 %vec0(<2 x s32>)
```

```
def : GINodeEquiv<G_REV64, AArch64rev64>;
```

Imported Tablegen rules for G_REV

```
(AArch64rev64:{*:[v2i32]} V64:{*:[v2i32]}:$Rn) => (REV64v2i32:{*:[v2i32]} V64:{*:[v2i32]}:$Rn)
(AArch64rev64:{*:[v2f32]} V64:{*:[v2f32]}:$Rn) => (REV64v2i32:{*:[v2f32]} V64:{*:[v2f32]}:$Rn)
(AArch64rev64:{*:[v4i16]} V64:{*:[v4i16]}:$Rn) => (REV64v4i16:{*:[v4i16]} V64:{*:[v4i16]}:$Rn)
...
```

```
if (selectImpl(I, *CoverageInfo))
  return true;
```

`selectImpl` will try to select `%shuf` using an imported rule.

```
%shuf:fpr64 = REV64v2i32 %vec0
```

```
def : GINodeEquiv<G_REV64, AArch64rev64>;
```

Selected with 0 lines of code added to the selector!

Imported Tablegen rules for G_REV

```
(AArch64rev64:{*:[v2i32]} V64:{*:[v2i32]}:$Rn) => (REV64v2i32:{*:[v2i32]} V64:{*:[v2i32]}:$Rn)
(AArch64rev64:{*:[v2f32]} V64:{*:[v2f32]}:$Rn) => (REV64v2i32:{*:[v2f32]} V64:{*:[v2f32]}:$Rn)
(AArch64rev64:{*:[v4i16]} V64:{*:[v4i16]}:$Rn) => (REV64v4i16:{*:[v4i16]} V64:{*:[v4i16]}:$Rn)
...
```

```
if (selectImpl(I, *CoverageInfo))
  return true;
```

`selectImpl` will try to select `%shuf` using an imported rule.

The road ahead…

Remaining work

Addressing the long tail

- Fixing performance last major performance regressions is high priority.
- Improving general code size on -Os.
- Both of these have long tails of work. The last 1% is much harder than the first 10% of regressions to fix.
- …but we’re not far away. Expecting to be on par with SelectionDAG on the key metrics in the next few months.

Fallbacks

- When GlobalISel cannot handle some input IR, it “falls back” to SelectionDAG for that function.
- Across most test suites, average fallback rate is around 1% or less.
- Significant amounts of work went into reaching this level. Fallbacks now have a negligible effect on metrics.
- But eventual goal is to have no fallbacks at all.

Enabling?

- We think the benefits of GlobalISel in optimization opportunities and overall architecture are compelling.
- Plan to flip the switch for Darwin platforms when the metrics targets are reached.
- Eventually, we should show net improvements modulo some niche cases.

Acknowledgments

GlobalISel is a wide ranging effort

Special thanks to:

- Other targets, downstream and in tree, contributing heavily to development over the last few years.
- Occasional contributors for patches and reviews.
- Bug reports from external projects.

Q&A