Generating Optimized Code with GlobalISel

Or: GlobalISel going beyond "it works"
Agenda

• What is GlobalISel?
• GlobalISel Combiner and Helpers
• Testing and Debugging
• Declarative Combiner
But first...
History

- In 2017, we got GlobalISel fully working for our target
  - Fast compile time, but codegen quality was significantly lower
History

• In 2017, we got GlobalISel fully working for our target
  ▶ Fast compile time, but codegen quality was significantly lower

• Added several new features to improve codegen quality
In 2017, we got GlobalISel fully working for our target

- Fast compile time, but codegen quality was significantly lower

- Added several new features to improve codegen quality

- By 2019, the codegen quality has improved
Apple GPU Compiler Uses GlobalISel
What is GlobalISel?

• GlobalISel is a new instruction selection framework
What is GlobalISel?

- GlobalISel is a new instruction selection framework
- Supports more global optimization (e.g. match across BasicBlocks)
What is GlobalISel?

• GlobalISel is a new instruction selection framework

• Supports more global optimization (e.g. match across BasicBlocks)

• More flexible

  ▶ From the speed of FastISel to the quality of SelectionDAGISel
What is GlobalISel?

• GlobalISel is a new instruction selection framework
• Supports more global optimization (e.g. match across BasicBlocks)
• More flexible
  ▪ From the speed of FastISel to the quality of SelectionDAGISel
• Easier to understand, maintain, and test
What is GlobalISel?

- GlobalISel is a new instruction selection framework
- Supports more global optimization (e.g. match across BasicBlocks)
- More flexible
  - From the speed of FastISel to the quality of SelectionDAGISel
- Easier to understand, maintain, and test
- Keeps all state in the Machine IR (MIR)
What is GlobalISel?

- GlobalISel is a new instruction selection framework
- Supports more global optimization (e.g. match across BasicBlocks)
- More flexible
  - From the speed of FastISel to the quality of SelectionDAGISel
- Easier to understand, maintain, and test
- Keeps all state in the Machine IR (MIR)
Anatomy of GlobalISel

---

LLVM-IR → Generic Machine Instructions (gMIR) → Generic Machine Instructions and Machine Instructions → Machine Instructions (MIR)

- IR Translator
- Legalizer
- Register Bank Selector
- Instruction Selector
IR Translator

Convert LLVM-IR into gMIR

LLVM-IR → IR Translator → Generic Machine Instructions (gMIR) → Legalizer → Generic Machine Instructions and Machine Instructions → Register Bank Selector → Instruction Selector → Machine Instructions (MIR)
Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019

LLVM-IR → Generic Machine Instructions (gMIR) → Generic Machine Instructions and Machine Instructions → Machine Instructions (MIR)

- IR Translator
- Legalizer
- Register Bank Selector
- Instruction Selector
Legalizer

Replace unsupported operations with supported ones

LLVM-IR → IR Translator → Legalizer → Register Bank Selector → Instruction Selector → Machine Instructions (MIR)

- LLVM-IR
- Generic Machine Instructions (gMIR)
- Generic Machine Instructions and Machine
- Machine Instructions (MIR)
Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019

- LLVM-IR
- Generic Machine Instructions (gMIR)
- Generic Machine Instructions and Machine Instructions
- Machine Instructions (MIR)

- IR Translator
- Legalizer
- Register Bank Selector
- Instruction Selector
Register Bank Selector

Binds registers to a Register Bank

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019

LLVM-IR

Translator

IR

Translator

Legalizer

Register

Bank

Selector

Instruction

Selector

Generic Machine Instructions (gMIR)

Generic Machine Instructions and Machine Instructions

Machine Instructions (MIR)
Generating Optimized Code with GlobalSel • LLVM Dev Meeting 2019

Instruction Selector

Select target instructions

LLVM-IR → Generic Machine Instructions (gMIR) → Generic Machine Instructions and Machine Instructions → Machine Instructions (MIR)

IR Translator → Legalizer → Register Bank Selector → Instruction Selector
Anatomy of GlobalISel

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Anatomy of GlobalISel

LLVM-IR → IR Translator → Generic Machine Instructions (gMIR) → Legalizer → Generic Machine Instructions and Machine Instructions → Register Bank Selector → Instruction Selector → Machine Instructions (MIR)

Tutorial: Head First into GlobalISel
Aditya Nandakumar, Daniel Sanders, and Justin Bogner
2017 LLVM Developers’ Meeting
Anatomy of GlobalISel

LLVM-IR → Generic Machine Instructions (gMIR) → Generic Machine Instructions and Machine Instructions → Machine Instructions (MIR)

IR Translator → Legalizer → Register Bank Selector → Instruction Selector
Combining Optimized Code with GlobalISel

17

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Combiner

Simplify/Optimize gMIR/MIR

LLVM-IR → Generic Machine Instructions (gMIR) → Machine Instructions (MIR)

IR Translator → Combiner 1 → Legalizer → Combiner 2 → Register Bank Selector → Instruction Selector

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019

Combiner

Simplify/Optimize gMIR/MIR

LLVM-IR ➔ Generic Machine Instructions (gMIR) ➔ Generic Machine Instructions and Machine Instructions ➔ Machine Instructions (MIR)

IR Translator ➔ Combiner 1 ➔ Legalizer ➔ Combiner 2 ➔ Register Bank Selector ➔ Instruction Selector
Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Why do we need combiners?
CodeGen Quality

- SelectionDAGISel
- GlobalISel w/o Opt
- GlobalISel w/Opt

Instruction Count (%)

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
CodeGen Quality

<2%
CodeGen Quality

- SelectionDAGISel
- GlobalISel w/o Opt
- GlobalISel w/Opt

Instruction Count (%):
- 0%
- 50%
- 100%
- 150%
Compile Time Performance

- SelectionDAGISel
- GlobalISel

Compile Time (%)
Compile Time Performance

- SelectionDAGISel
- GlobalISel

Generate Optimized Code with GlobalISel • LLVM Dev Meeting 2019
Compile Time Performance - ISel Only

- SelectionDAGISel
- GlobalISel

Graph showing compile time performance with SelectionDAGISel and GlobalISel.
Compile Time Performance - ISel Only

45%
Compile Time Performance - ISel Only

- SelectionDAGISel
- GlobalISel

Compile Time (%)
Features Needed

• Common Subexpression Elimination (CSE)
• Combiners
• KnownBits
• SimplifyDemandedBits
• Considered using MachineCSE, but it was expensive
CSE

- Considered using MachineCSE, but it was expensive
- We chose a continuous CSE approach
• Considered using MachineCSE, but it was expensive

• We chose a continuous CSE approach

• Instructions are CSE'd at creation time using CSEMIRBuilder
  ▶ Information is provided by an analysis pass
  ▶ BasicBlock-local
  ▶ Supports a subset of generic operations
Things to be aware of

- CSE needs to be informed of:
  - Changes to MachineInstrs (creation, modification, and erasure)
- Installs a delegate to handle creation/erasure automatically
- Installs a change observer to inform changes
Compile Time Cost

- We were expecting this to come at a big compile-time cost

- Improved compile time for some cases
  - Later passes had less work to do
Combiner

• Applies a set of combine rules
• Important for producing good code
• Expensive in terms of compile-time
What is a combine?

- An optimization that transforms a pattern into something more desirable
What is a combine?

• An optimization that transforms a pattern into something more desirable

```assembly
define i32 @foo(i8 %in) {
  %ext1 = zext i8 %in to i16
  %ext2 = zext i16 %ext1 to i32
  ret i32 %ext2
}
```
What is a combine?

• An optimization that transforms a pattern into something more desirable

```llvm
define i32 @foo(i8 %in) {
  %ext1 = zext i8 %in to i16
  %ext2 = zext i16 %ext1 to i32
  ret i32 %ext2
}
```
What is a combine?

• An optimization that transforms a pattern into something more desirable

```llvm
define i32 @foo(i8 %in) {
  %ext2 = zext i8 %in to i32
  ret i32 %ext2
}
```
GlobalISel Combiner

- GlobalISel Combiner consists of 3 main pieces
  - **Combiner** iterates over the MachineFunction
  - **CombinerInfo** specifies which operations to be combined and how
  - **CombinerHelper** is a library of generic combines
GlobalISelCombiner

MyTargetCombinerPass

Combiner

MyTargetCombinerInfo : CombinerInfo

combine(...)

CombinerHelper
A Basic Combiner

```cpp
bool MyTargetCombinerInfo::combine(GISelChangeObserver &Observer, 
    MachineInstr &MI, 
    MachineIRBuilder &B) const {

    MyTargetCombinerHelper TCH(Observer, B, KB);
    // ...
    // Try all combines.
    if (OptimizeAggresively)
        return TCH.tryCombine(MI);

    // Combine COPY only.
    if (MI.getOpcode() == TargetOpcode::COPY)
        return TCH.tryCombineCopy(MI);

    return false;
}
```
A Simple Combine

```cpp
bool MyTargetCombinerHelper::combineExt(GISelChangeObserver &Observer,
    MachineInstr &MI, MachineIRBuilder &B) const {
  // ..
  // Combine zext(zext x) -> zext x
  if (MI.getOpcode() == TargetOpcode::G_ZEXT) {
    Register SrcReg = MI.getOperand(1).getReg();
    MachineInstr *SrcMI = MRI.getVRegDef(SrcReg);
    // Check if SrcMI is a G_ZEXT.
    if (SrcMI->getOpcode() == TargetOpcode::G_ZEXT) {
      SrcReg = SrcMI->getOperand(1).getReg();
      B.buildZExt(Reg, SrcReg);
      MI.eraseFromParent();
    }
  }
  return true;
  // ...
}
```
bool MyTargetCombinerHelper::combineExt(GISelChangeObserver &Observer, 
    MachineInstr &MI, MachineIRBuilder &B) const {
    // ...
    // Combine zext(zext x) -> zext x
    if (MI.getOpcode() == TargetOpcode::G_ZEXT) {
        Register SrcReg = MI.getOperand(1).getReg();
        MachineInstr *SrcMI = MRI.getVRegDef(SrcReg);
        // Check if SrcMI is a G_ZEXT.
        if (SrcMI->getOpcode() == TargetOpcode::G_ZEXT) {
            SrcReg = SrcMI->getOperand(1).getReg();
            B.buildZExt(Reg, SrcReg);
            MI.eraseFromParent();
            return true;
        }
    }
    // ...
}
MIPatternMatch

• Simple and easy mechanism to match generic patterns

• Similar to what we have for LLVM IR

• Combines can be implemented easily using matchers
MIPatternMatch

- Simple and easy mechanism to match generic patterns
- Similar to what we have for LLVM IR
- Combines can be implemented easily using matchers

```c
// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg))))) {
    B.buildZExt(Reg, SrcReg);
    MI.eraseFromParent();
    return true;
}
```
A Simpler Combine

// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg))))) {
    B.buildZExt(Reg, SrcReg);
    MI.eraseFromParent();
    return true;
}
A Simpler Combine

// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg))))) {
    B.buildZExt(Reg, SrcReg);
    MI.eraseFromParent();
    return true;
}

// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg))))) {
    Observer.changingInstr(MI);
    MI.getOperand(1).setReg(SrcReg);
    Observer.changedInstr(MI);
    return true;
}
A Simpler Combine

// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg)))))) {
  B.buildZExt(Reg, SrcReg);
  MI.eraseFromParent();
  return true;
}

// Combine zext(zext x) -> zext x
Register SrcReg;
if (mi_match(Reg, MRI, m_GZext(m_GZext(m_Reg(SrcReg)))))) {
  Observer.changingInstr(MI);
  MI.getOperand(1).setReg(SrcReg);
  Observer.changedInstr(MI);
  return true;
}
Informing the Observer

• Observer needs to be informed when something changed
  • createdInstr() and erasedInstr() are handled automatically
  • changingInstr() and changedInstr() are handled manually and mandatory for MRI.setRegClass(), MO.setReg(), etc.
KnownBits Analysis

• Many combines are only valid for certain cases
KnownBits Analysis

• Many combines are only valid for certain cases

  ▪ \((a + 1) → (a | 1)\) is only valid if \((a & 1) == 0\)
KnownBits Analysis

• Many combines are only valid for certain cases
  - \((a + 1) \rightarrow (a \mid 1)\) is only valid if \((a \& 1) == 0\)

• We added an analysis pass to provide this information
KnownBits Analysis

• Many combines are only valid for certain cases
  
  ▶ (a + 1) → (a | 1) is only valid if (a & 1) == 0

• We added an analysis pass to provide this information

• Currently provides known-ones, known-zeros, and unknowns
Example

%1:(s32) = G_CONSTANT i32 0xFF0
%2:(s32) = G_AND %0, %1
%3:(s32) = G_CONSTANT i32 0x0FF
%4:(s32) = G_AND %2, %3

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%0</td>
<td>0x????????</td>
</tr>
<tr>
<td>%1</td>
<td>0x00000FF0</td>
</tr>
<tr>
<td>%2</td>
<td>0x00000FF</td>
</tr>
<tr>
<td>%3</td>
<td>0x00000FF</td>
</tr>
<tr>
<td>%4</td>
<td>0x00000FF</td>
</tr>
</tbody>
</table>

? = Unknown
Example

\[
\begin{align*}
%1: \text{(s32)} &= \text{G\_CONSTANT i32 } 0xFF0 \\
%2: \text{(s32)} &= \text{G\_AND } %0, %1 \\
%3: \text{(s32)} &= \text{G\_CONSTANT i32 } 0x0FF \\
%4: \text{(s32)} &= \text{G\_AND } %2, %3
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%0</td>
<td>0x????????????</td>
</tr>
<tr>
<td>%1</td>
<td>0x00000FF00</td>
</tr>
<tr>
<td>%2</td>
<td>0x00000??00</td>
</tr>
<tr>
<td>%3</td>
<td>0x000000FF</td>
</tr>
<tr>
<td>%4</td>
<td>0x000000?0</td>
</tr>
</tbody>
</table>

? = Unknown
### Example

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%0</td>
<td>0x??????????</td>
</tr>
<tr>
<td>%1</td>
<td>0x000000FF0</td>
</tr>
<tr>
<td>%2</td>
<td>0x000000??0</td>
</tr>
<tr>
<td>%3</td>
<td>0x0000000FF</td>
</tr>
<tr>
<td>%4</td>
<td>0x00000000?0</td>
</tr>
</tbody>
</table>

%1: (s32) = G_CONSTANT i32 0xFF0
%2: (s32) = G_AND %0, %1
%3: (s32) = G_CONSTANT i32 0x0FF
%4: (s32) = G_AND %2, %3

? = Unknown
Example

\%5: (s32) = G_CONSTANT i32 0xF0
\%4: (s32) = G_AND \%2, \%3

<table>
<thead>
<tr>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%0</td>
<td>0x?????????</td>
</tr>
<tr>
<td>%1</td>
<td>0x00000FF0</td>
</tr>
<tr>
<td>%2</td>
<td>0x00000??0</td>
</tr>
<tr>
<td>%3</td>
<td>0x000000FF</td>
</tr>
<tr>
<td>%4</td>
<td>0x000000?0</td>
</tr>
<tr>
<td>%5</td>
<td>0x000000F0</td>
</tr>
</tbody>
</table>

? = Unknown
Why an Analysis Pass?

• In SelectionDAGISel, computeKnownBits() is just a function

• In GlobalISel, it’s an Analysis Pass
Why an Analysis Pass?

• In SelectionDAGISel, computeKnownBits() is just a function

• In GlobalISel, it’s an Analysis Pass

• It allows us to add support for:
  ▶ Caching within a pass
  ▶ Caching between passes
  ▶ Early exit when enough is known
Why an Analysis Pass?

• In SelectionDAGISel, computeKnownBits() is just a function

• In GlobalISel, it’s an Analysis Pass

• It allows us to add support for:
  ▶ Caching within a pass
  ▶ Caching between passes
  ▶ Early exit when enough is known

• Allows us to have alternative implementations
Extending KnownBits

```cpp
void MyTargetLowering::computeKnownBitsForTargetInstr(
    GISelKnownBits &Analysis, Register R, KnownBits &Known,
    const APInt &DemandedElts, const MachineRegisterInfo &MRI,
    unsigned Depth = 0) const override {
    // ...
    switch (Opcode) {
    // ...
    case TargetOpcode::ANDWr: {
        Analysis.computeKnownBitsImpl(MI.getOperand(2).getReg(), Known, DemandedElts, Depth + 1);
        Analysis.computeKnownBitsImpl(MI.getOperand(1).getReg(), Known2, DemandedElts, Depth + 1);
        Known.One &= Known2.One;
        Known.Zero |= Known2.Zero;
        break;
    }
    // ...
    }
    // ...
}
```
KnownBits Analysis

- Allows optimizations that otherwise wouldn't be possible
- Available to any MachineFunction pass
- Caching will make it cheaper than SelectionDAGISelect's equivalent
Simplify Demanded Bits

• Essentially a special case of Combine

• Tries to eliminate calculations that contribute to the bits that are never read
SimplifyDemandedBits

• Essentially a special case of Combine

• Tries to eliminate calculations that contribute to the bits that are never read

• If demand mask is $0xF0$:

  $\text{\langle a \ll 16 \rangle | (b \& 0xFFFF) \rightarrow (b \& 0xFFFF)}$
SimplifyDemandedBits

• Essentially a special case of Combine

• Tries to eliminate calculations that contribute to the bits that are never read

• If demand mask is 0xF0:

  ▶ (a << 16) | (b & 0xFFFF) → (b & 0xFFFF)

• Not upstreamed yet, but we plan to fix that soon
Testing

LLVM-IR → SelectionDAG → Machine Instructions (MIR)

LLVM-IR → SelectionDAGISel → MIR
Generating Optimized Code with GlobalISel
LLVM Dev Meeting 2019

- LLVM-IR
- SelectionDAG
- SelectionDAGISel
- Machine Instructions (MIR)

- LLVM-IR
- Generic Machine Instructions (gMIR), Machine Instructions (MIR), and gMIR+MIR mixed

- LLVM-IR
- IR Translator
- Legalizer
- Register Bank Selector
- Instruction Selector
- gMIR
- gMIR + MIR
- gMIR + MIR
- MIR
Unit Testing

• Unit Testable too

  ▶ We use FileCheck as a library to check results
  ▶ It allows us to test exactly what optimizations do
Debugging

• It is error prone to implement optimizations from scratch
  ▶ Special cases
    ▶ Floating Point Precision issues (e.g. \( x \times y + z \rightarrow \text{fma}(x, y, z) \))
    ▶ Porting can be difficult too due to differences vs SelectionDAGISel

• It is especially hard to debug on GPUs
  ▶ Xcode has tool to debug shaders, but it relies on the compiler being correct
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function
BlockExtractor

- LLVM Pass used by llvm-extract
- Promotes specified BasicBlocks to functions
- Exploitable to find critical block(s) for a bug
- GlobalISel can be disabled per function

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
BlockExtractor

• Search space still too large?
  ▶ Split the BasicBlocks and repeat
BlockExtractor

• Search space still too large?
  ▶ Split the BasicBlocks and repeat

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
• Search space still too large?
  ▸ Split the BasicBlocks and repeat
BlockExtractor

- Search space still too large?
  - Split the BasicBlocks and repeat
BlockExtractor

- Search space still too large?
  - Split the BasicBlocks and repeat
• All the components are upstream

• You will need a driver script to put them together

$ ./bin/llvm-extract -o - -S \
  -b 'foo:bb9;bb20' <input> > extracted.ll
Advice
Advice: Minimize Fallbacks

• Falling back:
  ▶ Wastes compile time
  ▶ Skews quality metrics
Advice: Track Metrics Closely

- Catch regressions early
- Celebrate wins
Advice: Identify Key Optimizations

- Identify important optimizations
- Code Coverage Insights
- Minimize with BlockExtractor

- SelectionDAGISel Function
  - BB1 → BB2 → BB3
  - 40 instructions

- SDAGISel
  - BB4

- GlobalISel
  - BB1 → BB2 → BB3 → BB4
  - 45 instructions, 5 due to BB4
Advice: Starting a Combiner

• Simple combines go a long way
• PreLegalizerCombiner and PostLegalizerCombiner are easy starting points
Advice: Freedom

- Remember: Not a fixed pipeline
- Can replace passes
- Insert a pass where appropriate
Work In Progress
Declarative Combiner

- Modify RuleSets
  - Targets may wish to disable rules or make them only apply in certain circumstances
- Analyze RuleSets
  - Enables various kinds of tooling
- Optimize RuleSets
Goals

• Test Combine rules in isolation

• More debuggable: infinite loops and large rule-sets

• More target control

• Enable tools: profilers, coverage, static-analysis, proof engines

• Be independent of algorithm used
def : GICombineRule<
    (defs reg:$D, reg:$S),
    (match
        (G_ZEXT s16:$t1, s8:$S),
        (G_ZEXT s32:$D, s16:$t1)),
    (apply
        (G_ZEXT s32:$D, s8:$S))>;
Why not SelectionDAG's patterns?

- They can't describe several classes of DAG.
  - Only the bottom-up tree-like DAG's with limited node sharing
- Can't describe multiple results from one instruction
Example - SelectionDAG Style

- Analyzes def of the G_SEXT, G_ANYEXT, G_ZEXT in isolation
- Folds the G_LOAD down into the extend, duplicating the load
- Volatile/Atomics rejected unless hasOneUse()
Example - GlobalISel Style

Generating Optimized Code with GlobalISel • LLVM Dev Meeting 2019
def : GICombineRule<
  (defs reg:$D, reg:$S, instr:$MI0, instr:$MI1),
  (match (G_ZEXT $t0, $S):$MI0,
    (G_ZEXT $D, $t0):$MI1,
    (isScalarType type:$D),
    (isLargerType type:$D, type:$S)),
  (apply (G_ZEXT $D, $S, (debug_locations $MI0, $MI1))))>;
Rule Selection

• CombinerHelpers are declared in TableGen

• Specifies a class name and a list of combines in priority order

• Generated combiner ensures this order is honoured but still optimizes

```python
def MyPreLegalizerCombinerHelper : GICombinerHelper<
    "MyGenPreLegalizerCombinerHelper",
    [copy_prop, fold_add_0, fold_mul_1, postpone_sext_for_add, postpone_zext_for_add, postpone_sext_for_sub, postpone_zext_for_sub, extending_loads]>;
```
Rule Selection

• CombineGroups may be specified to factor out:
  ▶ Common combines (e.g. identities)
  ▶ Common target features (e.g. unfused_muladd, load_multiple, bswap)

```python
def identities : GICombineGroup<[fold_add_0, fold_mul_1]>;
def trivial_combines : GICombineGroup<[copy_prop, identities]>;
def postpone_extends : GICombineGroup<[postpone_sext_for_add, postpone_zext_for_add, postpone_sext_for_sub, postpone_zext_for_sub]>;
def MyPreLegalizerCombinerHelper: GICombinerHelper<"MyGenPreLegalizerCombinerHelper", [trivial_combines, postpone_extends, extending_loads]>;
```
Rule Selection

• CombineGroups may be specified to factor out:
  ▶ Common combines (e.g. identities)
  ▶ Common target features (e.g. unfused_muladd, load_multiple, bswap)

```python
def MyPreLegalizerCombinerHelper: GICombinerHelper<
    "MyGenPreLegalizerCombinerHelper",
    [trivial_combines, postpone_extends, extending_loads]>
```
Rule Selection

• Generated combiner includes a command line option when asked
  ▶ \(-\text{myprelegalizercombiner-disable-rule}=1\)
  ▶ \(-\text{myprelegalizercombiner-disable-rule}=0\,-50,75\,-100\)
  ▶ \(-\text{myprelegalizercombiner-disable-rule}=\text{fold}_2\,\text{plus}_2\,\text{to}_5\)

```python
def MyPreLegalizerCombinerHelper: GICombinerHelper<
    "MyGenPreLegalizerCombinerHelper",
    [trivial_combines, postpone_extends, extending_loads]> {  
    let DisableRuleOption = "myprelegalizercombiner-disable-rule";
}
```
Rule Selection

- Sometimes we can generally use a group but there's a small flaw
- Combiners (and maybe groups in future) can modify their contents
- Exact modifiers TBD

```python
def MyPreLegalizerCombinerHelper: GICombinerHelper<
    "MyGenPreLegalizerCombinerHelper",
    [trivial_combines, postpone_extends, extending_loads]> { 
    let DisableRuleOption = "myprelegalizercombiner-disable-rule";
    let Modifiers = [(disable_rule copy_prop),
                     (add_predicate lower_add_to_or, (when_profitable $d))];
}
```
Integration

• Generates a Combiner

• Integrate into CombinerInfo via constructor and combine() tweak

```cpp
AArch64GenPreLegalizerCombinerHelper Generated;
if (!Generated.parseCommandLineOption())
    report_fatal_error("Invalid rule identifier");

if (Generated.tryCombineAll(Observer, MI, B))
    return true;
```
def : GICombineRule<
    (defs reg:$D, reg:$S),
    (match (G_ZEXT s32:$t1, s8:$S),
        (G_ZEXT s16:$D, s32:$t1),
        (require (aloff 03, armv8, neon)),
        (a_b_testing "Experiment54")),
    (apply (G_ZEXT s16:$D, s8:$S),
        (debug_print "Investigate this test"),
        (tweet "@llvmorg" "Optimization win! 👻")))>;
Development Tools

• Coverage - Are rules tested? Do they trigger in practice?

• Profiler - Are they worth their cost?

• Controlled application of rules?
  ▶ If I applied them in this order would the outcome be better?
Debugging Tools

• Rule Bisection - Which one caused a miscompilation?

• N-stable Loop Detection - Why doesn't my combiner terminate?

• Rule Proving, i.e. ALIVE for backend? - Are my rules correct?

• State machine debugger? - What is the combiner doing?

• MIR Patches - Re-construct intermediate MIR
Recap
Recap
Recap

• In 2017, we got GlobalISel working but code quality wasn't there yet
Recap

• In 2017, we got GlobalISel working but code quality wasn't there yet

• Added continuous CSE, Combine, KnownBits, and SimplifyDemandedBits
Recap

• In 2017, we got GlobalISel working but code quality wasn't there yet

• Added continuous CSE, Combine, KnownBits, and SimplifyDemandedBits

• By 2019:
  ▶ Compile time 45% faster than SelectionDAGISel
  ▶ Generated code quality on par with SelectionDAGISel
Recap

• In 2017, we got GlobalISel working but code quality wasn't there yet
  • Added continuous CSE, Combine, KnownBits, and SimplifyDemandedBits

• By 2019:
  ▶ Compile time 45% faster than SelectionDAGISel
  ▶ Generated code quality on par with SelectionDAGISel

• Other targets are actively working on GlobalISel
Recap

• In 2017, we got GlobalISel working but code quality wasn't there yet

• Added continuous CSE, Combine, KnownBits, and SimplifyDemandedBits

• By 2019:
  ▷ Compile time 45% faster than SelectionDAGISel
  ▷ Generated code quality on par with SelectionDAGISel

• Other targets are actively working on GlobalISel

• We shipped it! You might even be using it!