A taste of GlobalISel

Alex Bradbury asb@igalia.com

EuroLLVM 2023, 2023-05-11
Approach

● Goal: Provide an intro to GlobalISel and how to use it with examples where possible.
● High level intro -> examples -> building up from the basics -> specific highlights -> End
● Won’t (as much as possible) assume SelectionDAG or even backend knowledge
● Out of scope: e.g. vector compilation
What is GlobalISel?

- GlobalISel
  - Global: Operates on whole function rather than a single BB
    - IMHO use of MachineIR is more of a defining feature than “global” scope.
  - ISel: Instruction Selection
- Compilation flow (simplified)
  - .c \rightarrow LLVM IR \rightarrow SelectionDAG \rightarrow MachineInstr \rightarrow MCInst \rightarrow .o
  - .c \rightarrow LLVM IR \rightarrow MachineInstr (generic) \rightarrow MachineInstr \rightarrow MCInst \rightarrow .o
What is MIR?

- A serialisation of MachineFunction(s) into a YAML container.
  - (Optional) first document is the embedded LLVM IR module.
  - Ensuing documents are serialised machine functions.
- Container used throughout the whole GlobalISel process.
- Not exclusive to GlobalISel - often helpful outside of GISel contexts.

```yaml
---
name: outline_0
tracksRegLiveness: true
isOutlined: false
body:
  bb.0:
    liveins: $x10, $x11
    $x11 = ORI $x11, 1023
    $x12 = ADDI $x10, 17
    $x11 = AND $x12, $x11
    $x10 = SUB $x10, $x11
    PseudoRET implicit $x10
...```
Why GlobalISel?

- **Performance**
  - No need to introduce a new dedicated intermediate representation as in SelectionDAG.

- **Granularity**
  - Operate on the whole function rather than individual basic blocks.

- **Modularity**
  - Enable more code reuse than e.g. FastISel and SelectionDAG (which share little code)

Ref: https://llvm.org/docs/GlobalISel/index.html
How does GlobalISel work?

- **IR Translator**
  - Translate LLVM IR to gMIR. Largely analogous to SelectionDAGBuilder.

- **(Combiner)**
  - Optional optimisations. Replace instructions with “better” (faster or smaller code size) alternatives.

- **Legalizer**
  - Replace unsupported operations with supported ones. No illegal instructions can remain after this pass.

- **(Combiner)**
  - Optional optimisations. Replace instructions with “better” (faster or smaller code size) alternatives.

- **Register Bank Selector**
  - Assign register banks to virtual registers (more on this later).

- **Instruction Selector**
  - Select target instructions from the gMIR generic instructions. No gMIR can remain after this pass.
Vs SelectionDAG pipeline

- Build initial DAG
- Optimize SelectionDAG (DAG combiner)
- Legalize SelectionDAG types (eliminate any types unsupported by the target)
- Optimize SelectionDAG (DAG combiner)
- Legalize SelectionDAG operations (eliminate operations unsupported by the target)
- Optimize SelectionDAG (DAG combiner)
- Select instructions (translate to DAG of target instructions)
- SelectionDAG scheduling and MachineFunction emission
GlobalISel status

- **Sources**: [Bringing up GlobalISel for optimized AArch64 codegen](2021 LLVM Dev Meeting), [RFC: Enabling GlobalISel for Apple AArch64 platforms](Jul’22, Discourse). Enabled in D137296, committed Nov’22 (though later reverted).
- “For O3 on Apple platforms we generate we see GlobalISel within 1% of SelectionDAG geomean [for performance and code size metrics]”
- “For compile time, on average we see improvements of about 5%”
  - ~2.5x faster comparing just equivalents part of CodeGen.
- ‘Fallback rate’ of ~1%.
- No scalable vector support
- Not (yet?) demonstrating codegen improvements over SelectionDAG due to global scope.
RISC-V overview

- Modern RISC ISA with open specification.
- 32 and 64-bit base ISAs (RV32I, RV64I)
- Broken up into a large number of ISA extensions referred to with single letters or short strings.
  - e.g. RV64IMAFDC supports the integer multiplication, atomics, single+double precision floating point, and compressed instruction set extensions.
- ILP32, ILP32E, ILP32F, ILP32D, LP64, LP64F, LP64D ABIs
- Vector instruction set extension with scalable vectors (as opposed to fixed length). Not covered in this talk.
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add \t$rd, $rs1, $rs2";
}

MC layer reminder

‘Flattened’ TableGen description of a simple instruction:
A simple example: selecting basic arithmetic

- The IRTranslator will produce **generic** Machine IR (gMIR)
  - e.g. G_ADD, G_XOR etc (see GenericOpcodes.td)
- We want to e.g. select a RISCV::ADD instruction for G_ADD with register operands
  - RISCVInstructionSelector::select(MachineInstr &MI) will be responsible for this.

```cpp
bool RISCVInstructionSelector::select(
    MachineInstr &I) const {
  // Non-generic instructions needing special handling
  // ..
  // Hand-written selects
  // ..
  // Table-genned function for imported SDag patterns.
  if (selectImpl(I, *CoverageInfo))
    return true;

  return false;
}
```
A simple example: selecting basic arithmetic

- Just as for SDAGISel, the basics are very simple - most of the work is in support code and corner cases.
- Can rely on imported SDAG patterns for the most part with some provisos
  - See llvm/Target/GlobalISel/SelectionDAGCompat.td
    - `def : GINodeEquiv< G_ADD, add>;`
    - (You’ll need to define your own for custom SDNodes mapping to target pseudos)
  - ComplexPatterns can’t be imported: use GIComplexOperandMatcher and GIComplexPatternEquiv
  - Replace PatLeaf with ImmLeaf, IntImmLeaf, FPImmLeaf etc as appropriate
Basic arithmetic/logical operations with legalisation

- Goal: transform generic machine instructions so they are legal
  - Type and operation legalisation aren’t separate
- Key references
  - RISCVLegalizerInfo.cpp
  - getActionDefinitionsBuilder API
  - LegalizeAction enum

Example:

```c
getActionDefinitionsBuilder({G_ADD, G_SUB})
  .legalFor({XLenLLT})
  .customFor({s32})
  .clampScalar(0, XLenLLT, XLenLLT);
```
Basic arithmetic/logical operations with legalisation

```cpp
getActionDefinitionsBuilder({G_ADD, G_SUB})
  .legalFor({XLenLLT})
  .customFor({s32})
  .clampScalar(0, XLenLLT, XLenLLT);
```

- LLT, “Low Level Type” is intended to replace usage of EVT in SelectionDAG.
  - e.g. `LLT s32 = LLT::scalar(32)`
    - Note: Can’t differentiate different types of same width
- If the customFor predicate is satisfied, lower via backend-supplied `legalizeCustom`
- `clampScalar`: Indicates the range of legal type widths
- Lots of shared logic in `llvm/lib/CodeGen/GlobalISel/LegalizerHelper.cpp`
Legalising types wider than native width

- G_MERGE_VALUES and G_UNMERGE_VALUES are introduced.

e.g. i64 add on RV32

%2:_(s32) = COPY $x10
%3:_(s32) = COPY $x11
%4:_(s32) = COPY $x12
%5:_(s32) = COPY $x13
%1:_(s64) = G_MERGE_VALUES %4(s32), %5(s32)
%0:_(s64) = G_MERGE_VALUES %2(s32), %3(s32)
%6:_(s64) = G_ADD %0, %1
%7:_(s32), %8:_(s32) = G_UNMERGE_VALUES %6(s64)
$x10 = COPY %7(s32)
$x11 = COPY %8(s32)
PseudoRET implicit $x10, implicit $x11
Register banks and register selection

- Targets commonly have a register bank for GPRs and FPRs.
- Generic virtual registers initially just have an LLT constraint, then (through RegBankSelect) become constrained to a register bank.
- `def GPRRegBank : RegisterBank<"GPRB", [GPR]>;`
- `RISCVRegisterBankInfo::getInstrMapping(const MachineInstr &MI)` is responsible for returning the default bank assignments for a given instruction.
- Floating point operations are different opcodes to integer (e.g. G_ADD vs G_FADD), so the type can be inferred.
getInstrMapping
excerpt from PPC

Observe G_F* must be performed on FPRs, while e.g. G_LOAD/G_STORE could be GPRs or FPRs.

```c
getInstrMapping
{
    Register SrcReg = MI.getOperand(1).getReg();
    unsigned Size = getSizeInBits(SrcReg, MRI, TRI);

    assert((Size == 32 || Size == 64 || Size == 128) &&
           "Unsupported floating point types!
"
);
    switch (Size) {
        case 32:
            OperandsMapping = getValueMapping(PMI_FPR32);
            break;
        case 64:
            OperandsMapping = getValueMapping(PMI_FPR64);
            break;
        case 128:
            OperandsMapping = getValueMapping(PMI_VEC128);
            break;
    }
    break;
```
Building up GlobalISel support in a backend
Basic infrastructure

- Introduce the most basic infrastructure
  - RISCVCallLowering
    - lowerReturn (can initially only support void returns), lowerFormalArguments, lowerCall (can return false initially)
    - Called by IRTranslator
  - RISCVInstructionSelector
    - select() just calling selectImpl
  - RISCVLegalizerInfo
    - Can initially be empty
  - RISCVRegisterBanks
    - Define the GPR register bank
  - Add GISel passes to RISCVTargetMachine, getters+initialisers to RISCVSubtarget
Return, argument, and call lowering

- Implemented in RISCVCallLowering
- ‘Incoming’ and ‘outgoing’ value handlers provide implementations of getStackAddress, assignValueToAddress, assignValueToReg
- e.g. implementation of RISCVCallReturnHandler::assignValueToReg

```cpp
void assignValueToReg(Register ValVReg, Register PhysReg,
                       CCValAssign VA) override {
    // Copy argument received in physical register to desired VReg.
    MIB.addDef(PhysReg, RegState::Implicit);
    MIRBuilder.buildCopy(ValVReg, PhysReg);
}
```
Handling constants

- The IRTranslator lowers constants to G_CONSTANT or G_FCONSTANT instructions.
- The selector can then fold constants into immediate operands.
- See lib/CodeGen/GlobalISel/Localizer.cpp for a pass to minimise the live range of G_CONSTANTs to reduce register pressure.
- RISCVInstructionSelector uses the shared RISCVMatInt class to generate an efficient sequence to materialise arbitrary constants (shared between SDag, MC layer, GISel).
Edit/compile/test loop - tips

- Study existing tests. e.g. test/CodeGen/RISCV/*
- Make heavy use of update_{llc,mir}_test_checks.py to generate and maintain CHECK lines.
- High quality tests and high test coverage is \_essential\_ and has a high return on investment
- Misc
  - Ensure you have a debug+asserts build
  - -debug flag to llc
  - -print-after-all to llc
  - llvm_unreachable, assert
  - Fire up your favourite debugger
  - sys::PrintStackTrace(llvm::errs())
Edit/compile/test loop - tips

- **-global-isel-abort**
  - 0 Disables the abort (i.e. fallback to SDag)
  - 1 Enables it
  - 2 Disables the abort but emits a diagnostic on failure

- Use **-run-pass** or **-stop-after llc** options to control pass execution
  - e.g. **-stop-after=irtranslator** (useful for producing an MIR used in a **-run-pass** based test case)

- Use **-simplify-mir llc** option to produce simpler MIR
  - More tips in the MIR Lang Ref doc

- Use **llvm-tblgen --stats** for statistics on patterns emitted and **--warn-on-skipped patterns** for more details on patterns that can’t be imported.
Optimisations

- Not yet pursued in RISC-V, so see AArch64 for in-tree examples
  - AArch64PreLegalizerCombiner (also an O0 variant) and AArch64PostLegalizerCombiner.cpp
  - Matching rules defined in AArch64Combine.td using
    
    ```
    def fold_global_offset_matchdata : GIDefMatchData<'std::pair<uint64_t, uint64_t>'>;
    def fold_global_offset : GICombineRule<
        (defs root:$root, fold_global_offset_matchdata:$matchinfo),
        (match (wip_match_opcode GLOBAL_VALUE):$root,
            [{ return matchFoldGlobalOffset(*${root}, MRI, ${matchinfo}); }]),
        (apply [{  return applyFoldGlobalOffset(*${root}, MRI, B, Observer, ${matchinfo});}])
    >;
    ```
Observations and open questions

- Reference GISel backend
  - I’ve used RISC-V as it’s what I know - PPC may be a better reference right now, but hoping this will change during this year.
- Clear examples where GISel is easier than SelectionDAG?
  - Fewer in-tree examples can also make it harder (vs SelectionDAG where you can often review multiple approaches across different targets)
- GloballSel-only targets?
- Fine-grained operation legality for ‘W’ RV64 instructions?
- Following AArch64 in adopting a lowering pass?
- RISC-V GISel next steps
Credits and other resources

- Thanks
  - All GlobalISel contributors and doc authors.
  - RISC-V GlobalISel contributors, especially Lewis Revill.

- Other resources
  - PPC GlobalISel patches
  - GlobalISel cookbook [D137111](#)
  - [GlobalISel docs](#)
  - [2017 GlobalISel tutorial](#)
  - LLVM Weekly!

- Questions?
- Contact: asb@igalia.com