Faster Compilation with GlobalISel: Skipping LLVM-IR

Tobias Stadler
to.stadler@tum.de

(with contributions from Alexis Engelke)

Chair of Data Science and Engineering
Department of Computer Science
Technical University of Munich

EuroLLVM ’24, Vienna, AT, 2024-04-10
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
- Workload: JIT-compiling database queries, AArch64

![Graph showing compile-time comparison between FastISel-O0 and GlobalISel-O0]
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
- Workload: JIT-compiling database queries, AArch64

![Diagram showing the breakdown of compilation time for FastISel-O0 and GISel-O0]
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
- Workload: JIT-compiling database queries, AArch64

![Graph showing the timeline of various LLVM passes and their overhead]

- FastISel-O0
- GISel-O0

Legend:
- Gen. LLVM-IR
- LLVM-IR Passes
- ISel
- Other Passes
- Overhead

Passes:
- IRTranslator
- Legalizer
- RegBankSelect
- InstructionSelect
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
- Workload: JIT-compiling database queries, AArch64

![Diagram showing the workflow of IRTranslator, Legalizer, RegBankSelect, and InstructionSelect with different passes and overheads.]

- Skip LLVM-IR, build generic MachineIR directly?
- Are all passes necessary?
Motivation

- Long-term goal: SelectionDAG, FastISel, GlobalISel
- Workload: JIT-compiling database queries, AArch64

Are all passes necessary?

Skip LLVM-IR, build generic MachineIR directly?
Building MachineIR from the outside

- MachineIR is stored in the CodeGen pipeline!
  - owned by ImmutablePass MachineModuleInfoWrapperPass
  - MachineModuleInfo maps Function to MachineFunction

```cpp
auto* MMIWP = new MachineModuleInfoWrapperPass(static_cast<LLVMTargetMachine*>(TM));

Function *F = Function::Create(...);
// Add placeholder IR block (otherwise F is a declaration)
BasicBlock *BB = BasicBlock::Create(...);
IRBuilder IB(BB); IB.CreateUnreachable();

MachineModuleInfo &MMI = MMIWP->getMMI();
MachineFunction &MF = MMI.getOrCreateMachineFunction(*F);
// ... build MachineIR
// Debugging: MF.verify(); MF.dump();

legacy::PassManager PM;
TM->addPassesToEmitFile(PM, ..., MMIWP); // JIT: addPassesToEmitMC (needs patch)
```
Generic MachineIR Basics

%3:_(s32) = G_ADD %1:_, %2:_

▶ Generic target opcodes: TargetOpcode::G_*
▶ LowLevelType (LLT): scalar, pointer, vector

LLT llt = LLT::scalar(32);
Register reg = MRI.createGenericVirtualRegister(llt);
Generic MachineIR Basics

```plaintext
%3: _(s32) = G_ADD %1: _, %2: _
```

- **Generic target opcodes**: `TargetOpcode::G_*`
- **LowLevelType (LLT)**: scalar, pointer, vector

```plaintext
LLT llt = LLT::scalar(32);
Register reg = MRI.createGenericVirtualRegister(llt);
```

- **MachineIRBuilder**: helper to build generic `MachineInstrs` at insertion point
  - **DstOp**: Register, LLT, TargetRegisterClass
  - **SrcOp**: Register, MachineInstrBuilder, immediate, ...

```plaintext
MachineBasicBlock *MBB = MF.CreateMachineBasicBlock();
MF.push_back(MBB);

MachineIRBuilder MIRBuilder(*MBB, MBB->end());
MachineInstrBuilder dst = MIRBuilder.buildInstr(TargetOpcode::G_ADD, {llt}, {arg1, arg2});
MIRBuilder.buildAdd(llt, reg, dst);
```
Constants and Globals

- Generic instructions are defined on virtual Registers → no immediate operands
- Constants must be explicitly materialized into virtual Registers
- \texttt{g\_constant}: ConstantInt, \texttt{g\_fconstant}: ConstantFP
- \texttt{g\_global\_value}: pointer to LLVM-IR global
Control Flow

- **G_BR**: unconditional branch
- **G_BRCOND**: conditional branch, fall-through on false
- Legal only at end of basic block (either one G_BRCOND, one G_BR or both)
- Block successor/predecessor lists need to be manually updated
- addSuccessor calls addPredecessor

```cpp
MBB.addSuccessor(otherMBB, BranchProbability::getOne());
// or:
MBB.addSuccessorWithoutProb(OtherMBB);
```
No alloca abstraction!

MachineFrameInfo: tracks abstract stack frame until prolog/epilog insertion

MachineFrameInfo &MFI = MF.getFrameInfo();
Stack Frame

- No alloca abstraction!
- MachineFrameInfo: tracks abstract stack frame until prolog/epilog insertion

```cpp
MachineFrameInfo &MFI = MF.getFrameInfo();

- Create static stack objects directly in MachineFrameInfo
- Materialize stack address using G_FRAME_INDEX

```cpp
int frameIndex = MFI.CreateStackObject(size, align, false);
MIRBuilder.buildFrameIndex(dst, frameIndex);
```
Stack Frame

▶ No alloca abstraction!
▶ MachineFrameInfo: tracks abstract stack frame until prolog/epilog insertion

MachineFrameInfo &MFI = MF.getFrameInfo();

▶ Create static stack objects directly in MachineFrameInfo
▶ Materialize stack address using G_FRAME_INDEX

int frameIndex = MFI.CreateStackObject(size, align, false);
MIRBuilder.buildFrameIndex(dst, frameIndex);

▶ Dynamic allocation using G_DYN_STACKALLOC

MIRBuilder.buildDynStackAlloc(dst, size, align);
MFI.CreateVariableSizedObject(align, nullptr /* alloca instr */);
G_LOAD, G_STORE, G_ATOMICRMW_ADD, ...

MachineMemOperand: describes memory access

```cpp
auto* MMO = MF.getMachineMemOperand(
  MachinePointerInfo(addressSpace),
  // or: e.g MachinePointerInfo::getFixedStack(MF, frameIndex)
MachineMemOperand::MOLoad | MachineMemOperand::MOStore,
LLT::scalar(64),
Align(8),
// optional:
AAMDNodes(), // Aliasing metadata
nullptr, // Range metadata
// Atomic
SyncScope::System,
AtomicOrdering::SequentiallyConsistent,
// Atomic failure ordering (e.g G_ATOMIC_CMPXCHG_WITH_SUCCESS)
AtomicOrdering::SequentiallyConsistent
);
```
Calls

- CallLowering
  - implemented by Target
  - lowers calling convention using physical register copies and target instructions

```cpp
int64_t some_lib_call(int64_t a, int64_t b)

Type* ty = Type::getInt64Ty(...);
CallLowering::CallLoweringInfo CLI;
CLI.Callee = MachineOperand::CreateES("some_lib_call"); // -> external symbol
  // CreateReg(...) -> indirect call
  // CreateGA(...) -> llvm::Function
CLI.OrigArgs.emplace_back(regForA, ty, 0);
CLI.OrigArgs.emplace_back(regForB, ty, 1);
CLI.OrigRet = {regForRetVal, ty, 0};
CallLowering& CL = *MF.getSubtarget().getCallLowering();
bool Success = CL.lowerCall(MIRBuilder, CLI);
```
Function Arguments and Returns

FunctionLoweringInfo FuncInfo;
FuncInfo.MF = &MF;
// return using registers or need sret demotion?
FuncInfo.CanLowerReturn = CL.checkReturnTypeForCallConv(MF);
Function Arguments and Returns

FunctionLoweringInfo FuncInfo;
FuncInfo.MF = &MF;
// return using registers or need sret demotion?
FuncInfo.CanLowerReturn = CL.checkReturnTypeForCallConv(MF);

▶ Formal arguments

bool Success = CL.lowerFormalArguments(MIRBuilder, *F, {reg0, ...}, FuncInfo);
Function LoweringInfo FuncInfo;
FuncInfo.MF = &MF;
// return using registers or need sret demotion?
FuncInfo.CanLowerReturn = CL.checkReturnTypeForCallConv(MF);

▶ Formal arguments

bool Success = CL.lowerFormalArguments(MIRBuilder, *F, {reg0, ...}, FuncInfo);

▶ Return

// void
bool Success = CL.lowerReturn(MIRBuilder, nullptr, {}, FuncInfo, 0);
// Value
Value* pseudoVal = llvm::UndefValue::get(F->getReturnType()); // care only about Type
bool Success = CL.lowerReturn(MIRBuilder, pseudoVal, reg, FuncInfo, 0);
Misc. Lowerings

- No switches $\rightarrow$ lower to branches manually
- No getelementptr $\rightarrow$ G_PTR_ADD
- Intrinsic
  - G_INTRINSIC, G_INTRINSIC_W_SIDE_EFFECTS, ...
  - MachineIRBuilder::buildIntrinsic() picks correct intrinsic opcode
  - some intrinsics translate to own generic opcode
    - memcpy $\rightarrow$ G_MEMCPY
    - uadd_with_overflow $\rightarrow$ G_UADDO
    - ...

Skipping GlobalISel passes

- Combiner
  - MIR-to-MIR rewriting
  - iterate MIR and greedily match instructions until convergence

- AArch64(O0)PreLegalizerCombiner

- Localizer
  - moving/duplicating constants close to their uses
  - shorter live ranges → work around register allocation limitations

- AArch64PostLegalizerLowering
  - Combiner for lowering certain instructions (mostly G_SHUFFLE_VECTOR)

- Pipeline customization via:
  - command-line-flags: -start-before=legalizer
  - patching AArch64TargetMachine
Minimal run-time regression from skipping non-mandatory passes!
Generating clean IR matters

1 Umbra DBMS, all TPC-DS queries, sf-1, Apple M1, performance cores only
Compile-time O0

- gMIR-O0 vs. GISel-O0: −27%
- gMIR-O0 vs. FastISel-O0: +19%
- IRTranslator vs. manual gMIR construction: −15%
- MachineIR construction is expensive
- IR tuned for low FastISel fallback-rate, but 49% of ISel spent in SelectionDAG
Compile-time O2

- GISel-O2 vs. DagISel-O2: −8%
- GlobalISel Combiners are expensive
- gMIR-O2 vs. GISel-O2: −23%
Tradeoffs

+
  ▶ Compile-time
  ▶ Complete control over MachineIR
Tradeoffs

+ ▶ Compile-time
  ▶ Complete control over MachineIR

− ▶ No IR passes
  ▶ no middle-end optimizations
  ▶ no Mem2Reg → construct SSA manually
  ▶ No SelectionDAG fallback
  ▶ (Needs patched LLVM)
(Possible) Improvements

- Early elision of `G_AND` artifacts in Legalizer
  - CTMark: −5.6% O0 size..text, −0.9% compile-time

- Visibility: llvm-compile-time-tracker GlobalISel configuration?
- Combiners
  - eliminate/reduce fixed-point iteration (like InstCombine)
- RegBankSelect
  - disable for O0
  - extend InstructionSelect to handle LLTs
Summary

- Emitting generic MachineIR directly is possible and can improve compile-time
- CodeGen pipeline has tuning potential, especially with GlobalISel
- Difficult to match compile-time performance of low fallback-rate FastISel
- Future work
  - experiment with integrating FastISel (or FastISel-like capabilities) into IRTranslator
  - “FastISelMachineIRBuilder”? Auto-generation from TableGen patterns?