Agile approach to lifting LLVM MIR code into LLVM IR

(https://github.com/ajlekcahdp4/llvm-bleach)

Romanov Alexander

Binary translators

Generic

- Specialized low-level representation
- Difficult to support new architectures
- Register mapping
- Instruction mapping

LLVM-based

- Lifting to LLVM IR
- Practically unlimited target architecture set
- Well-developed compiler and tooling infrastructure

LLVM-based lifters

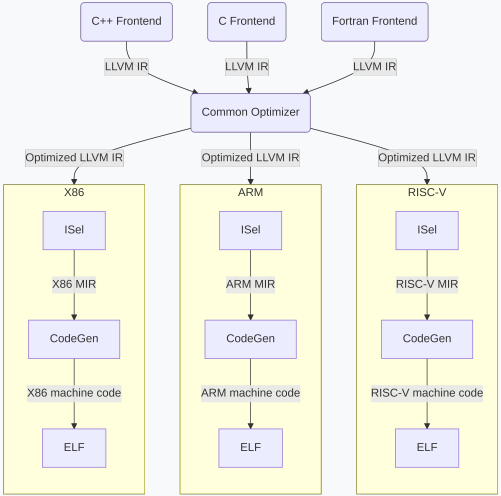
Existing solutions

- mcsema (2014)
- mctoll (2019)
- instrew (2020)
- biotite (2025)

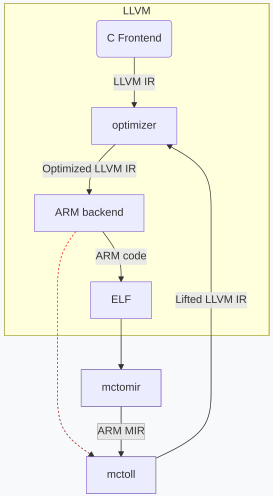
Flaws

- Target-specific lifting algorithm
- Difficult to support new source architectures

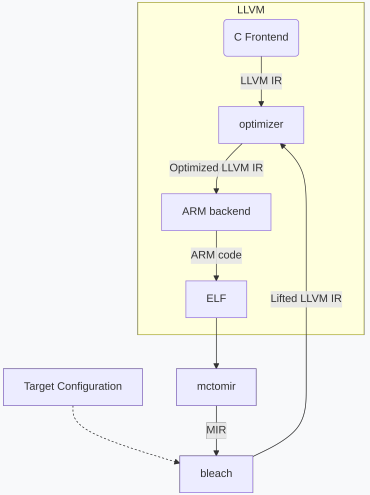
LLVM compiler



LLVM-based lifters



solution - llvm-bleach



llvm-bleach

Innovations

- Source target description - configuration file
- Do NOT attempt to lift target signature
- Generic (target-independent) translation algorithm
- Does not use LLVM backend during lifting

llvm-bleach

Advantages

- Easy to support new source architectures
- Configurable lifting process
- Simplified build system

Generic Approach

Each MIR instruction is mapped to LLVM function

- Source operand -> argument
- Destination operand -> return value

MIR Instruction

```
$x1 = ADD $x2, $x3
```

LLVM Function

```
define i64 @ADD(i64 %0, i64 %1) {
    %3 = add i64 %1, %0
    ret i64 %3
}
```

Code translation

MIR Block (RISC-V)

```
bb.1:
  $x19 = MUL $x7, $x2
  $x28 = ADD $x21, $x19
  $x13 = ADD $x13, $x1
  BNE $x13, $x3, %bb.1
```

Output LLVM IR

```
bb1:
%19 = call i64 @MUL(i64 %x7, i64 %x2)
%x28 = call i64 @ADD(i64 %x21, i64 %19)
%13 = call i64 @ADD(i64 %x13, i64 %x1)
%cmp = icmp ne i64 %13, %x3
br i1 %cmp, label %bb1, label %bb2
```

Output LLVM IR after inlining

```
bb1:
  %19 = mul i64 %x7, %x2
  %x28 = add i64 %x21, %19
  %13 = add i64 %x13, %x1
  %cmp = icmp ne i64 %13, %x3
  br i1 %cmp, label %bb1, label %bb2
```

Function Calls

- Custom calling convention

State struct - the only argument

```
define void @foo(ptr %0) {
  %GPR = getelementptr %register_state, ptr %0, i32 0, i32 0
  ... # loading registers from state
  %x2_upd = add i64 %x2, -1
  ... # save updated registers to state
  call void @bar(ptr %0)
  ... # reloading registers from state
  %x23_upd = mul i64 %x16, %x14
  ... # save updated registers to state
  ret void
}
```

Algorithm

```
def lift(F: function):
  loadRegsFromState(F.StateArgument)
  for MBB in F:
    for I in MBB:
      if I.isCall():
        saveRegsToState(F.StateArgument)
        BB.insertCall(I.Callee, F.StateArgument)
        reloadRegs(F.StateArgument)
      else if I.isCondBranch():
        Cond = BB.insertCall(I.function, I.Src1, I.Src2)
        BB.insertBranch(Cond, I.ifTrue, I.ifFalse);
      else:
        Regs[I.Dst] = BB.insertCall(I.func, I.Src1, I.Src2)
  saveRegsToState(F.StateArgument)
```

Peering forward

- Target configuration generation from formal specification (sail-riscv for RISC-V)
- Syscall translation
- Support both Dynamic and Static binary translation
- Testing of clang compiler
- Using llvm-bleach in binary instrumentation

Thank you for your attention!