Lecture 2
Overview of the LLVM Compiler

Dominic Chen

Thanks to:
Vikram Adve, Jonathan Burket, Deby Katz,
David Koes, Chris Lattner, Gennady Pekhimenko,
and Olatunji Ruwase, for their slides

LLVM Compiler System
The LLVM Compiler Infrastructure
- Provides reusable components for building compilers
- Reduce the time/cost to build a new compiler
- Build different kinds of compilers
- Our homework assignments focus on static compilers
- There are also JITs, trace-based optimizers, etc.

The LLVM Compiler Framework
- End-to-end compilers using the LLVM infrastructure
- Support for C and C++ is robust and aggressive
- Java, Scheme and others are in development
- Emit C code or native code for x86, SPARC, PowerPC

Visualizing the LLVM Compiler System

The LLVM Optimizer is a series of “passes”
- Analysis and optimization passes, run one after another
- Analysis passes do not change code, optimization passes do

LLVM Intermediate Form is a Virtual Instruction Set
- Language- and target-independent form
  - Used to perform the same passes for all source and target languages
  - Internal Representation (IR) and external (persistent) representation

LLVM: From Source to Binary
```c
int main() {
    int a = 5;
    int b = 3;
    return a - b;
}
```

Read "Life of an instruction in LLVM":
Machine Inst

BB0: derived from LLVM BB %entry
Live Ins: %EBP
  PUSH32r %EBP, %ESP<imp-def>, %ESP<imp-use>; flags: FrameSetup
  %EPO<def> = MOV32r %ESP; flags: FrameSetup
  %ESP<def,tied1> = SUB32ri8 %ESP<imp-def>, 12, %EFLAGS<imp-def,dead>; flags: FrameSetup
  MOV32mi %EBP, 1, %noreg, -4, %noreg; mem:ST4[%retval]
  MOV32mi %EBP, 1, %noreg, %noreg, 3; mem:ST4[%a]
  MOV32mi %EBP, 1, %noreg, -12, %noreg, 3; mem:ST4[%b]
  %EAX<def> = MOV32rm %EBP, -8, %noreg; mem:LD4[%a]
  %EAX<def,tied1> = ADD32ri8 %EAX<kill,tied0>, -3, %EFLAGS<imp-def,dead>
  %EAX<def,tied1> = ADD32ri8 %EAX<imp-def,tied1>, 12, %EFLAGS<imp-def,dead>
  %EBP<def> = POP32r %ESP<imp-def>, %ESP<imp-use>
  RETL %EAX

Assembly

main:                                   # @main
  # BB0:                                   # %entry
  pushl %ebp                             # <MCInst #2191 PUSH32r
  movl %esp, %ebp                        # <MCOperand Reg:20>
  subl $12, %esp                         # <MCInst #2685 SUB32ri8
  movl $0, -4(%ebp)                     # <MCInst #1554 MOV32mi
  movl $5, -8(%ebp)                     # <MCOperand Imm:1>
  movl $3, -12(%ebp)                    # <MCOperand Imm:12>
  movl %eax, %eax                       # <MCOperand Reg:0>
  movl %eax, -8(%ebp)                   # <MCOperand Imm:-4>
  movl %eax, %eax                       # <MCOperand Reg:0>
  movl %eax, -4(%ebp)                   # <MCOperand Imm:0>
Goals of LLVM Intermediate Representation (IR)

- Easy to produce, understand, and define
- Language- and Target-Independent
- One IR for analysis and optimization
- Supports high- and low-level optimization
- Optimize as much as early as possible

LLVM Instruction Set Overview

- Low-level and target-independent semantics
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers

```c
loop: ; preds = %bb0, %loop
  %i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum(float %AiAddr, %pair* %P)
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.1, %N
  br i1 %exitcond, label %outloop, label %loop

for (i = 0; i < N; i++)
  Sum(&A[i], &P);
```
LLVM Instruction Set Overview (continued)

- High-level information exposed in the code
  - Explicit dataflow through SSA form (more later in the class)
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
  - Preserves array subscript and structure indexing

Lowering Source-Level Types to LLVM

- Source language types are lowered:
  - Rich type systems expanded to simple types
  - Implicit & abstract types are made explicit & concrete

Examples of lowering:
- Reference turn into pointers: T& -> T*
- Complex numbers: complex float -> (float, float)
- Bitfields: struct X { int Y:4; int Z:2; } -> { i32 }

The entire type system consists of:
- Primitives: label, void, float, integer, ...
- Arbitrary bitwidth integers (i1, i32, i64, i1942652)
- Derived: pointer, array, structure, function (unions get turned into casts)
- No high-level types
- Type system allows arbitrary casts

Example Function in LLVM IR

```llvm
define i32 @main() #0 {
  entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

Example Function in LLVM IR

```llvm
define i32 @main() #0 {
  entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```
LLVM Instruction Hierarchy

LLVM Instructions <-> Values

int main() {
    int x;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}

Instruction I: %add1 = add nsw i32 %add, 3

You can’t “get” %add1 from Instruction I. Instruction serves as the Value %add1.

LLVM Instructions <-> Values

int main() {
    int x;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}

Definition of %add1:
%add1 = add nsw i32 %add, 3

You can’t “get” %add1 from Instruction I. Instruction serves as the Value %add1.

LLVM Instructions <-> Values

int main() {
    define i32 @main() #0 {
        entry:
        %retval = alloca i32, align 4
        %x = alloca i32, align 4
        %y = alloca i32, align 4
        %z = alloca i32, align 4
        %l = load i32* %x, align 4
        %r = load i32* %y, align 4
        store i32 %l, i32* %z, align 4
    }
}

Instruction I: %add1 = add nsw i32 %add, 3

You can’t “get” %add1 from Instruction I. Instruction serves as the Value %add1.

LLVM Instructions <-> Values

int main() {
    define i32 @main() #0 {
        entry:
        %retval = alloca i32, align 4
        %x = alloca i32, align 4
        %y = alloca i32, align 4
        %z = alloca i32, align 4
        %l = load i32* %x, align 4
        %r = load i32* %y, align 4
        store i32 %l, i32* %z, align 4
    }
}

Instruction I: %add1 = add nsw i32 %add, 3

You can’t “get” %add1 from Instruction I. Instruction serves as the Value %add1.

Only makes sense for an SSA Compiler
LLVM Program Structure

- **Module** contains **Functions and GlobalVariables**
  - Module is a unit of analysis, compilation, and optimization
- **Function** contains **BasicBlocks and Arguments**
  - Functions roughly correspond to functions in C
- **BasicBlock** contains a list of **Instructions**
  - Each block ends in a control flow instruction
- **Instruction** is an opcode + vector of operands

Traversal of the LLVM IR data structure usually occurs through doubly-linked lists

LLVM also supports the **Visitor Pattern** (more next time)

LLVM Pass Manager

- Compiler is organized as a series of “passes”:
  - Each pass is an analysis or transformation
  - Each pass can depend on results from previous passes
- Six useful types of passes:
  - BasicBlockPass: iterate over basic blocks, in no particular order
  - CallGraphSCCPass: iterate over SCC’s, in bottom-up call graph order
  - FunctionPass: iterate over functions, in no particular order
  - LoopPass: iterate over loops, in reverse nested order
  - ModulePass: general interprocedural pass over a program
  - RegionPass: iterate over single-entry/exit regions, in reverse nested order
- Passes have different constraints (e.g. FunctionPass):
  - FunctionPass can only look at the “current function”
  - Cannot maintain state across functions

LLVM Tools

- Basic LLVM Tools
  - llvm-dis: Convert from .bc (IR binary) to .ll (human-readable IR text)
  - llvm-as: Convert from .ll (human-readable IR text) to .bc (IR binary)
  - opt: LLVM optimizer
  - llc: LLVM static compiler
  - lli: LLVM bytecode interpreter
  - llvm-link: LLVM bytecode linker
  - llvm-ar: LLVM archiver
- Some Additional Tools
  - bugpoint - automatic test case reduction tool
  - llvm-extract - extract a function from an LLVM module
  - llvm-bcana - LLVM bytecode analyzer
  - FileCheck - Flexible pattern matching file verifier
  - tblgen - Target Description To C++ Code Generator
**opt: LLVM modular optimizer**

- **Invoke arbitrary sequence of passes:**
  - Completely control PassManager from command line
  - Supports loading passes as plugins from *.so files

  ```
  opt -load foo.so -pass1 -pass2 -pass3 x.bc -o y.bc
  ```

- **Passes "register" themselves:**
  - When you write a pass, you must write the registration

  ```
  RegisterPass<FunctionInfo> X("function-info",
  "15745: Function Information");
  ```