Lecture 3
Overview of the LLVM Compiler

Substantial portions courtesy of Gennady Pekhimenko, Olatunji Ruwase, Chris Lattner, Vikram Adve, and David Koes

LLVM Compiler System

- **The LLVM Compiler Infrastructure**
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...
- **The LLVM Compiler Framework**
  - End-to-end compilers using the LLVM infrastructure
  - C and C++ are robust and aggressive:
    - Java, Scheme and others are in development
  - Emit C code or native code for X86, Sparc, PowerPC

Three primary LLVM components

- **The LLVM *Virtual Instruction Set***
  - The common language- and target-independent IR
  - Internal (IR) and external (persistent) representation
- **A collection of well-integrated libraries**
  - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, …
- **A collection of tools built from the libraries**
  - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer, …

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **The Pass Manager**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

Running Example: Argument Promotion

Consider use of by-reference parameters:

```
int callee(const int &X)
{
   return X+1;
}
int caller() {
   return callee(4);
}
```

compiles to

```
int callee(const int *X) {
   return *X+1;    // memory load
}
int caller() {
   int tmp;        // stack object
   tmp = 4;        // memory store
   return callee(&tmp);
}
```

We want:

```
int callee(int X) {
   return X+1;
}
int caller() {
   return
callee(4);
}
```

- call procedure with constant argument

Why is this hard?

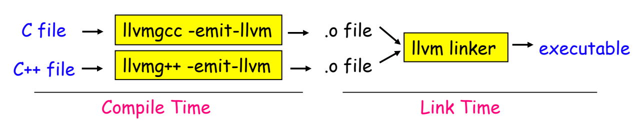
- **Requires interprocedural analysis:**
  - Must change the prototype of the callee
  - Must update all call sites → we must **know** all callers
  - What about callers outside the translation unit?
- **Requires alias analysis:**
  - Reference could alias other pointers in callee
  - Must know that loaded value doesn't change from function entry to the load
  - Must know the pointer is not being stored through
- **Reference might not be to a stack object!**

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **The Pass Manager**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

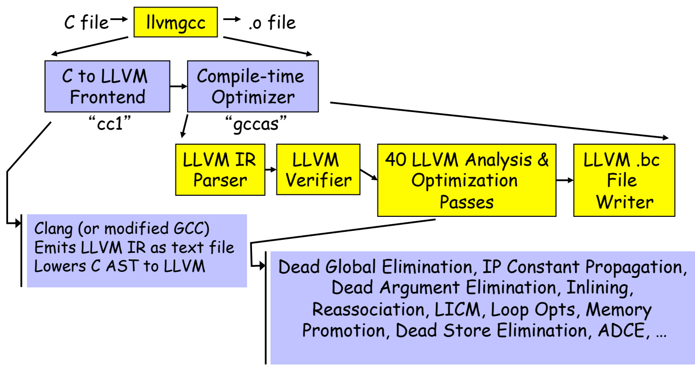
The LLVM C/C++ Compiler

- **From a high-level perspective, it is a standard compiler:**
  - Compatible with standard makefiles
  - Uses Clang (or possibly GCC 4.2) C and C++ parser

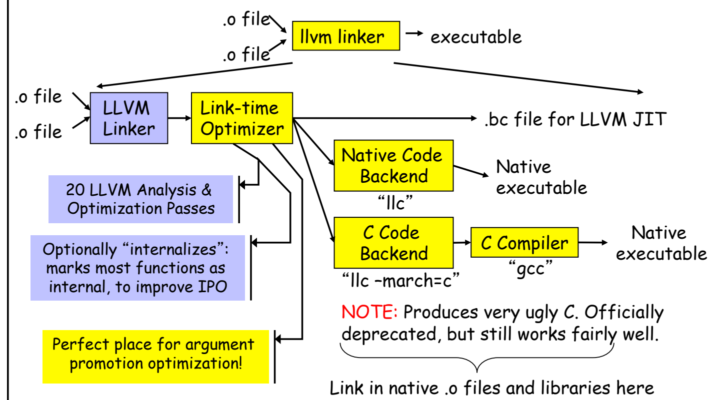


- **Distinguishing features:**
  - Uses LLVM optimizers (not GCC optimizers)
  - .o files contain LLVM IR/bytecode, not machine code
  - Executable can be bytecode (JIT'd) or machine code

Looking into events at compile-time



Looking into events at link-time



Goals of the compiler design

- **Analyze and optimize as early as possible:**
  - Compile-time opts reduce modify-rebuild-execute cycle
  - Compile-time optimizations reduce work at link-time (by shrinking the program)
- **All IPA/IPO make an open-world assumption**
  - Thus, they all work on libraries and at compile-time
  - "Internalize" pass enables "whole program" optzn
- **One IR (without lowering) for analysis & optzn**
  - Compile-time optzns can be run at link-time too!
  - The same IR is used as input to the JIT

IR design is the key to these goals!

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **The Pass Manager**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

Goals of LLVM Intermediate Representation (IR)

- **Easy to produce, understand, and define!**
- **Language- and Target-Independent**
- **One IR for analysis and optimization**
  - IR must be able to support aggressive IPO, loop opts, scalar opts, … high- *and* low-level optimization!
- **Optimize as much as early as possible**
  - Can't postpone everything until link or runtime
  - No lowering in the IR!

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
- **IR has text, binary, and in-memory forms**

```
for (i = 0; i < N; i++)
  Sum(&A[i], &P);
```

```
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
```


LLVM Instruction Set Overview (Continued)

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

```
for (i = 0; i < N; i++)
  Sum(&A[i], &P);
```

```
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
```


LLVM Type System Details

- **The entire type system consists of:**
  - Primitives: label, void, float, integer, …
    - Arbitrary bitwidth integers (i1, i32, i64)
  - Derived: pointer, array, structure, function
  - No high-level types: type-system is language neutral!
- **Type system allows arbitrary casts:**
  - Allows expressing weakly-typed languages, like C
  - Front-ends can implement safe languages
  - Also easy to define a type-safe subset of LLVM

See also: `docs/LangRef.html`

Lowering source-level types to LLVM

- **Source language types are lowered:**
  - Rich type systems expanded to simple type system
  - Implicit & abstract types are made explicit & concrete
- **Examples of lowering:**
  - References turn into pointers: `T&` → `T*`
  - Complex numbers: `complex float` → `{ float, float }`
  - Bitfields: `struct X { int Y:4; int Z:2; }` → `{ i32 }`
  - Inheritance: `class T : S { int X; }` → `{ S, i32 }`
  - Methods: `class T { void foo(); }` → `void foo(T*)`
- **Same idea as lowering to machine code**

LLVM Program Structure

- **Module contains Functions/GlobalVariables**
  - Module is unit of compilation/analysis/optimization
- **Function contains BasicBlocks/Arguments**
  - Functions roughly correspond to functions in C
- **BasicBlock contains list of instructions**
  - Each block ends in a control flow instruction
- **Instruction is opcode + vector of operands**
  - All operands have types
  - Instruction result is typed

Our Example, Compiled to LLVM

```
int callee(const int *X) {
   return *X+1;   // load
}
int caller() {
   int T;         // on stack
   T = 4;         // store
   return callee(&T);
}
```

```
internal int %callee(int* %X) {
   %tmp.1 = load int* %X
   %tmp.2 = add int %tmp.1, 1
   ret int %tmp.2
}
int %caller() {
   %T = alloca int
   store int 4, int* %T
   %tmp.3 = call int %callee(int* %T)
   ret int %tmp.3
}
```


Our Example, Compiled to LLVM

```
int callee(const int *X) {
   return *X+1;   // load
}
int caller() {
   int T;         // on stack
   T = 4;         // store
   return callee(&T);
}
```

```
internal int %callee(int* %X) {
   %tmp.1 = load int* %X
   %tmp.2 = add int %tmp.1, 1
   ret int %tmp.2
}
int %caller() {
   %T = alloca int
   store int 4, int* %T
   %tmp.3 = call int %callee(int* %T)
   ret int %tmp.3
}
```

- Stack allocation is explicit in LLVM

Our Example, Compiled to LLVM

```
int callee(const int *X) {
   return *X+1;   // load
}
int caller() {
   int T;         // on stack
   T = 4;         // store
   return callee(&T);
}
```

```
internal int %callee(int* %X) {
   %tmp.1 = load int* %X
   %tmp.2 = add int %tmp.1, 1
   ret int %tmp.2
}
int %caller() {
   %T = alloca int
   store int 4, int* %T
   %tmp.3 = call int %callee(int* %T)
   ret int %tmp.3
}
```

- All loads and stores are explicit in the LLVM representation

Our Example, Compiled to LLVM

```
int callee(const int *X) {
   return *X+1;   // load
}
int caller() {
   int T;         // on stack
   T = 4;         // store
   return callee(&T);
}
```

```
internal int %callee(int* %X) {
   %tmp.1 = load int* %X
   %tmp.2 = add int %tmp.1, 1
   ret int %tmp.2
}
int %caller() {
   %T = alloca int
   store int 4, int* %T
   %tmp.3 = call int %callee(int* %T)
   ret int %tmp.3
}
```

- Linker "internalizes" most functions in most cases

Our Example: Desired Transformation

```
internal int %callee(int* %X) {
   %tmp.1 = load int* %X
   %tmp.2 = add int %tmp.1, 1
   ret int %tmp.2
}
int %caller() {
   %T = alloca int
   store int 4, int* %T
   %tmp.3 = call int %callee(int* %T)
   ret int %tmp.3
}
```

```
internal int %callee(int %X.val) {
   %tmp.2 = add int %X.val, 1
   ret int %tmp.2
}
int %caller() {
   %tmp.3 = call int %callee(int 4)
   ret int %tmp.3
}
```

- Change the prototype for the function
- Update all call sites of "callee"
- Other transformation (-mem2reg) cleans up the rest

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **The Pass Manager**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

LLVM Coding Basics

- **Written in modern C++, uses the STL:**
  - Particularly the vector, set, and map classes
- **LLVM IR is almost all doubly-linked lists:**
  - Module contains lists of Functions & GlobalVariables
  - Function contains lists of BasicBlocks & Arguments
  - BasicBlock contains list of Instructions
- **Linked lists are traversed with iterators:**

```
Function *M = …
for (Function::iterator I = M->begin(); I != M->end(); ++I) {
  BasicBlock &BB = *I;
  ...
```

See also: docs/ProgrammersManual.html

LLVM Pass Manager

- **Compiler is organized as a series of "passes":**
  - Each pass is one analysis or transformation
- **Four types of passes:**
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time
- **Constraints imposed (e.g. FunctionPass):**
  - FunctionPass can only look at "current function"
  - Cannot maintain state across functions

See also: docs/WritingAnLLVMPass.html

Services provided by PassManager

- **Optimization of pass execution:**
  - Process a function at a time instead of a pass at a time
  - Example: three functions, *F*, *G*, *H* in input program, and two passes X & Y:
    "X(*F*)Y(*F*) X(*G*)Y(*G*) X(*H*)Y(*H*)" not "X(*F*)X(*G*)X(*H*) Y(*F*)Y(*G*)Y(*H*)"
  - Process functions in parallel on an SMP (future work)
- **Declarative dependency management:**
  - Automatically fulfill and manage analysis pass lifetimes
  - Share analyses between passes when safe:
    - e.g. "DominatorSet live unless pass modifies CFG"
- **Avoid boilerplate for traversal of program**

See also: docs/WritingAnLLVMPass.html

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **The Pass Manager**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

LLVM tools: two flavors

- **"Primitive" tools: do a single job**
  - llvm-as: Convert from .ll (text) to .bc (binary)
  - llvm-dis: Convert from .bc (binary) to .ll (text)
  - llvm-link: Link multiple .bc files together
  - llvm-prof: Print profile output to human readers
  - llvmc: Configurable compiler driver
- **Aggregate tools: pull in multiple features**
  - gccas/gccld: Compile/link-time optimizers for C/C++ FE
  - bugpoint: automatic compiler debugger
  - llvm-gcc/llvm-g++: C/C++ compilers

See also: docs/CommandGuide/

opt tool: 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:**

```
RegisterPass<SimpleArgPromotion> X("simpleargpromotion",
           "Promote 'by reference' arguments to 'by value'");
```

- **Standard mechanism for obtaining parameters**

```
opt<string> StringVar("sv", cl::desc("Long description of param"),
cl::value_desc("long_flag"));
```

- **From this, they are exposed through opt:**

```
> opt -load libsimpleargpromote.so -help
  ...
 -sccp               - Sparse Conditional Constant Propagation
 -simpleargpromotion - Promote 'by reference' arguments to 'by
 -simplifycfg        - Simplify the CFG
  ...
```


Assignment 1 - Practice

- **Introduction to LLVM**
  - Install and play with it
- **Learn interesting program properties**
  - Functions: name, arguments, return types, local or global
  - Compute live values using iterative dataflow analysis

Assignment 1 - Questions

- **Building Control Flow Graph**
- **Data Flow Analysis**
  - Available Expressions
    - Apply existing analysis
  - New Dataflow Analysis

Questions?

- **Thank you**