The LLVM Compiler Framework and Infrastructure (Part 1)

Presented by Gennady Pekhimenko

Substantial portions courtesy of Olatunji Ruwase, Chris Lattner, Vikram Adve, and David Koes
The 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
- Assignment Overview
Running example: arg promotion

Consider use of by-reference parameters:

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

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

Compiles to

- Eliminated load in callee
- Eliminated store in caller
- Eliminated stack slot for ‘tmp’

We want:

```c
int callee(int X) {
    return X+1;
}
int caller() {
    return callee(4);
}
```
Why is this hard?

- Requires interprocedural analysis:
  - Must change the prototype of the callee
  - Must update all call sites $\rightarrow$ 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
- Assignment Overview
The LLVM C/C++ Compiler

- From the high level, it is a standard compiler:
  - Compatible with standard makefiles
  - Uses 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

```
C file → llvmgcc -emit-llvm → .o file   LLVM linker → executable
C++ file → llvmp++ -emit-llvm → .o file
```

Compile Time                  Link Time
Looking into events at compile-time

C file → llvmgcc → .o file

C to LLVM Frontend

Compile-time Optimizer

“cc1”

“gccas”

LLVM IR Parser

LLVM Verifier

Lowers C AST to LLVM

40 LLVM Analysis & Optimization Passes

LLVM .bc File Writer

Dead Global Elimination, IP Constant Propagation, Dead Argument Elimination, Inlining, Reassociation, LICM, Loop Optxs, Memory Promotion, Dead Store Elimination, ADCE, …
Looking into events at link-time

LLVM Linker ➔ Link-time Optimizer ➔ LLVM linker ➔ executable

.o file ➔ .o file

Native Code Backend “llc” ➔ Native executable

C Code Backend “llc -march=c” ➔ C Compiler “gcc” ➔ Native executable

20 LLVM Analysis & Optimization Passes

Optionally “internalizes”: marks most functions as internal, to improve IPO

Perfect place for argument promotion optimization!

NOTE: Produces very ugly C. Officially deprecated, but still works fairly well.

Link in native .o files and libraries here
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
- Assignment Overview
Goals of LLVM IR

- **Easy to produce, understand, and define!**
- **Language- and Target-Independent**
  - AST-level IR (e.g. ANDF, UNCOL) is not very feasible
    - Every analysis/xform must know about ‘all’ languages
- **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 #1

- **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**

```assembly
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
```
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

```assembly
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
```
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: \( T& \rightarrow T* \)
  - Complex numbers: `complex float \rightarrow \{ float, float \}`
  - Bitfields: `struct X { int Y:4; int Z:2; } \rightarrow \{ i32 \}`
  - Inheritance: `class T : S { int X; } \rightarrow \{ S, i32 \}`
  - Methods: `class T { void foo(); } \rightarrow 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
}

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

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
- Assignment Overview
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:**

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

See also: docs/ProgrammersManual.html
Compiler is organized as a series of ‘passes’: 
- Each pass is one analysis or transformation

Four types of Pass:
- 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 $X$ & $Y$:
    
    \[
    X(F)Y(F) X(G)Y(G) X(H)Y(H) \quad \text{not} \quad X(F)X(G)X(H) Y(F)Y(G)Y(H)
    \]
  - 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](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
- Assignment Overview
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
...
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
- Assignment Overview
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