The LLVM Compiler Framework and Infrastructure

15745: Optimizing Compilers

Olatunji Ruwase

Substantial portions courtesy of 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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

Running example: arg 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);
}
```

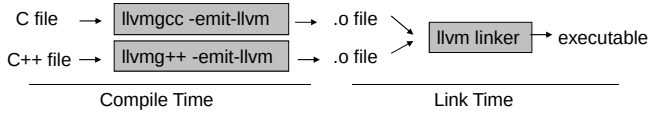
- ✓Eliminated load in callee
- ✓Eliminated store in caller
- ✓Eliminated stack slot for 'tmp'

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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

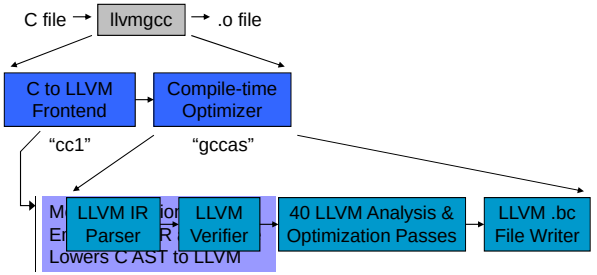
The LLVM C/C++ Compiler

- **From the high level, it is a standard compiler:**
  - Compatible with standard makefiles
  - Uses GCC 4.2 C and C++ parser

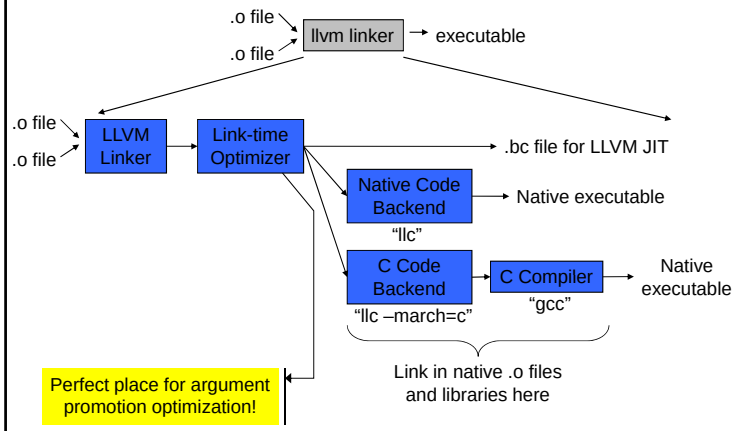


- **Distinguishing features:**
  - Uses LLVM optimizers, not GCC optimizers
  - .o files contain LLVM IR/bytecode, not machine code
  - Executable can be bytecode (JIT'd) or machine code

Looking into events at compile-time



Looking into events at link-time



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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

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**

```
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 #2

- **High-level information exposed in the code**
  - Explicit dataflow through SSA form
  - 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

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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

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 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

LLVM Dataflow Analysis

- **LLVM IR is in SSA form:**
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions
- **Control Flow Graph is always available:**
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG
- **Higher-level info implemented as passes:**
  - Dominators, CallGraph, induction vars, aliasing, GVN, …

See also: docs/ProgrammersManual.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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

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:**

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

- **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
  ...
```


Running Arg Promotion with opt

- **Basic execution with 'opt':**
  - **`opt -simpleargpromotion in.bc -o out.bc`**
  - Load .bc file, run pass, write out results
  - Use "-load filename.so" if compiled into a library
  - PassManager resolves all dependencies
- **Optionally choose an alias analysis to use:**
  - **`opt –basicaa –simpleargpromotion`** **(default)**
  - Alternatively, **`–steens-aa, –anders-aa, –ds-aa, …`**
- **Other useful options available:**
  - **`-stats`: Print statistics collected from the passes**
  - **`-time-passes`: Time each pass being run, print output**

LLC Tool: Static code generator

- **Compiles LLVM → native assembly language**
  - Currently for X86, Sparc, PowerPC (others in alpha)
  - `llc file.bc -o file.s -march=x86`
  - `as file.s –o file.o`
- **Compiles LLVM → portable C code**
  - `llc file.bc -o file.c -march=c`
  - `gcc –c file.c –o file.o`
- **Targets are modular & dynamically loadable:**
  - `llc –load libarm.so file.bc -march=arm`

25

LLI Tool: LLVM Execution Engine

- **LLI allows direct execution of .bc files**
  - **E.g.:** `lli grep.bc -i foo *.c`
- **LLI uses a Just-In-Time compiler if available:**
  - Uses same code generator as LLC
    - Optionally uses faster components than LLC
  - Emits machine code to memory instead of ".s" file
  - JIT is a library that can be embedded in other tools
- **Otherwise, it uses the LLVM interpreter:**
  - Interpreter is extremely simple and very slow
  - Interpreter is portable though!

26

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
- **LLVM C++ IR and important API's**
  - Basics, PassManager, dataflow
- **Important LLVM Tools**
  - opt, code generator, JIT
- **Assignment 1**

27

Assignment 1

- **Introduction to LLVM**
  - Install and play with it
- **Learn interesting program properties**
  - Functions: name, arguments, return types, local or global
- **Local optimizations (within basic blocks)**
  - Algebraic simplification
  - Strength reduction
  - Constant folding

28