Introduction to the LLVM Compiler Infrastructure

Chris Lattner
Apple Computer
clattner@apple.com

Gelato ICE 2006
April 25, 2006
LLVM Talk Overview

- Introducing LLVM
- Building a Static Compiler with LLVM Components
- LLVM Code Representation (IR)
- GCC + LLVM Integration
- Itanium Code Generator Status
- (more) Q&A
What is a Compiler?

A tool that inspects and manipulates a representation of programs

- Examples:
  - Traditional C compiler (gcc), Java JIT compiler (hotspot), system assembler (as), system linker (ld), IDEs (Xcode), refactoring tools, ...

- Intentionally a very broad definition

LLVM is not a compiler
What is a Compiler Infrastructure?

• Provides modular & reusable components for building compilers
  • Components are ideally language/target independent

• Reduces the time & cost to construct a particular compiler
  • A new compiler = glue code plus any components not available

• Allows components to be shared across different compilers
  • Improvements made to one compiler benefits the others

• Allows choice of the right component for the job
  • Does not force the use of “one true register allocator” or scheduler

LLVM is a compiler infrastructure
llvm-gcc is a compiler
What is the LLVM Compiler Infrastructure?

**Low Level Virtual Machine**

- A well-defined **Intermediate Representation (IR)** for programs
  - Language independent, target independent, easy to use

- Many high-quality libraries (components) with **clean interfaces**!
  - Optimizations, analyses, modular code generator, JIT compiler, accurate GC, profiling, debugging, X86/PPC/IA64/SPARC/Alpha code generators, link time optimization, IPA/IPO…

- Tools built from the libraries:
  - Aggressive optimizing C/C++/ObjC compiler, automated compiler debugger, compiler driver, modular optimizer, LLVM JIT…

This all exists and works today!

http://llvm.org/
Building a Static Compiler with LLVM Components
Example of a Simple Static Compiler

- Standard compiler organization, which uses LLVM as midlevel IR:
  - Language specific front-end lowers code to LLVM IR
  - Language/target independent optimizers improve code
  - Code generator converts LLVM code to target (e.g. IA64) code

Many compilers (e.g. GCC) follow this model.

http://llvm.org/
Front-end options for this compiler

- Front-ends are truly separate from optimizer & codegen
  - Can use front-end AST’s that are tailored to the source language
  - Optimizer & Codegen improvements benefit all front-ends
  - Front-ends generate debug info and include it in the IR

Language Front-end → LLVM IR → Mid-Level Optimizer → LLVM IR → Code Generator → .s file

Key LLVM Feature:
IR is small, simple, easy to understand, and is well defined

Retarget or write parsers for other languages

GCC Parsers

C/C++, FORTRAN, Ada, Java, ...

Python, JavaScript, ...

llvm-gcc currently uses the GCC 4.0.1 parsers

http://llvm.org/
Optimizer options for this compiler

- Optimizer is solely concerned with semantics of LLVM IR
  - Optimizer & Codegen **only know LLVM**, not all source languages
  - LLVM includes IP framework and aggressive IP optimizations
  - LLVM uses a modern and light-weight (fast) SSA-based optimizer

http://llvm.org/
Link-Time Optimization

• Link-time is a natural place for interprocedural optimizations
  − Cross-module optzn is natural and trivial (no makefile changes)
  − All optimizations respect limitations of incomplete programs
    − e.g. building an app with missing libraries, building a library, etc...
  − LTO has been available since LLVM 1.0!

http://llvm.org/
CodeGen options for this compiler

- The LLVM code generator is modern and modular:
  - Modern: maintains SSA form until register allocation
  - Modular: choose components based on compiler constraints
    - e.g. Itanium port uses a PQS, could use more aggressive scheduler
  - Fast: Representation is similar to the “compressed RTL” GCC proposal
CodeGen choices this compiler

- Portable IR provides flexibility for many different ways to codegen
- Note: IR can have symbols stripped, like machine code
  - ... LLVM IR does not suffer from Java/C#'s “easy to decompile” problem
More Aggressive Applications of LLVM

Run-time code generation
Efficient implementation of mini languages: Dynamically translate language to LLVM, then JIT compile.

Dynamic Code Specialization
Good for long-running computations with “dynamic constants”. Use LLVM to specialize run-time constants into the code, then optimize based on them.

Install-time Code Generation
Tune apps for the specific architecture at the end-user site

Vendor provides code generator?
“Old binaries scheduled for new chips”

http://llvm.org/
The LLVM Code Representation (IR)
Requirements on the LLVM IR

• IR must be usable through much of the compiler:
  – Produced by front-ends, consumed by code generator

• It must be language- and target-independent:
  – Including mixing of source languages within the same LLVM file
  – Allows cross-language analysis and optimization
  – Can still perform target-specific optimizations on it

• It must host a wide variety of optimizations and analyses:
  – Standard scalar optimizations (e.g. common subexpr elimination)
  – Loop optimizations (e.g. LICM, unrolling, unswitching, …)
  – Interprocedural (e.g. inlining, arg promotion, IP-SCCP, global var opt)
  – Must support both high- and low-level optimization
Design Approach of the LLVM IR

• Design IR as a typed Virtual Instruction Set
  – Operations are low-level instructions in CFG
  – Language- & target- independent semantics

• IR is designed with three isomorphic formats:
  – In memory IR - for the compiler to work on
  – On-disk compressed binary IR - Interchange format
  – On-disk text - Compiler debugging, inspection

• IR has a clean/simple design:
  – Small memory footprint, fast to manipulate
  – Easy to understand (and well specified/documentated)

http://llvm.org/docs/LangRef.html
LLVM IR Features

• Basic features:
  - Light-weight design, efficient and easy to understand
  - Scalars values are always in SSA form, memory never is
  - IR is fully typed and types are rigorously checked for consistency
  - Explicit array/struct accesses, supports alias/dependence analysis
  - Full support for vector/SIMD datatypes and operations
  - Full support for GCC-style inline assembly

• Minor features:
  - Exceptions are explicit in CFG, not an on-the-side datastructure
  - Includes support for Accurate Garbage Collection
  - IR is easily extensible with intrinsic functions
  - Supports custom calling conventions (required for guaranteed tail calls)
Example LLVM Tool: Bugpoint
Automatically reduce optimizer/codegen ICEs, miscompilations, and JIT failures

• Simple idea: binary search for bug
  - Figure out which pass (out of 60+) is causing the problem
  - Figure out what (code) input to the pass demonstrates the problem
• For a compiler crash:
  - Binary search pass list. Run previous passes to get its input.
  - Split up program, eliminate pieces not required for ICE
• For a miscompilation:
  - Run program with designated input to determine if it works
  - Split program, optimize/codegen half, link together, run.
• Can reduce 100K LOC program to a single basic block in 5 mins

• Simple tool reuses many LLVM libraries, relies on well defined IR

http://llvm.org/docs/Bugpoint.html
LLVM + GCC Integration
LLVM + Apple GCC Integration

• llvm-gcc 4.0 is the 3rd edition of llvm-gcc:
  – Based on Apple GCC 4.0.1 branch
  – GIMPLE to LLVM translation: ~6000 lines of code
  – Tight integration: llvm-gcc links in the LLVM libraries
  – GCC front-ends, LLVM optimizers & code generators

• Current status:
  – Mostly feature complete:
    – Supports C/C++/ObjC/ObjC++, vector support, debug info, has basic inline asm support, most GCC attributes, etc
  – Missing features (as of April 25, 2006):
    – No linker support for transparent IPO yet (exists in llvm-gcc3)
    – C++ Exception Handling (exists in llvm-gcc3)
    – long double and other minor features
LLVM + FSF GCC Integration: Design

• Most likely design point: replace tree-ssa with LLVM, keep RTL
  − Convert from GENERIC to LLVM in frontend
  − Convert from LLVM to RTL in the backend

• Design Advantages:
  − GCC gets LLVM LTO support, a light-weight IR and fast optimizer
  − LLVM is similar to tree-ssa: tree-ssa expertise should transfer well
  − By using the RTL backend, no GCC targets are lost

• Eventually could use native LLVM backends if desired:
  − Enables JIT compiler for Java, faster compiles, direct .o file emission, better codegen, easier porting to new targets
  − ... for the subset of GCC targets that are supported by LLVM

These thoughts are based on my impression of the GCC mailing list discussions, details subject to change!

http://llvm.org/
LLVM + FSF GCC Integration: Progress

• Remaining technical issues to resolve:
  − No LLVM to RTL backend implemented yet
  − Must forward port from Apple 4.0.1 branch to mainline
  − Must implement minor missing features

• Assigning control / Copyright assignment to FSF:
  − Ongoing project!
  − Progress since November:
    − FSF okay’s writing IR to disk, LTO proposal is made
    − No more web registration required to download LLVM
    − Official LLVM domain changes from llvm.cs.uiuc.edu to llvm.org
    − Many missing features implemented in LLVM (vector support, target intrinsics, inline asm, debugging, ...)
    − Continuing to work with the copyright clerk and related parties to complete paperwork
LLVM vs LTO for link-time optimization

- LTO advantages over LLVM:
  - LLVM is missing some functionality, has no LLVM-to-RTL backend yet
- LLVM advantages over LTO:
  - LLVM has had IPO support since before tree-ssa was started!
  - LLVM exists, works great, and can be evaluated today
  - LLVM has far more efficient data structures than GCC:
    - LLVM can represent 200K LOC in ~50M, GCC requires multi GB
    - Many projects to fix GCC’s mem usage have had limited success
  - Without major changes, LTO cannot link different languages or flags:
    - Langhooks and global flags like -ffast-math are a big problem
  - LTO suffers same class of bugs that IMA does:
    - Linking “GCC trees” is extremely hard to do 100% correctly
    - Cross language linking multiplies the problem many-fold
  - Does not try to solve front-end issues with IPA infrastructure!
LLVM Itanium Code Generator
LLVM Itanium Backend Status

• Itanium backend developed & maintained by Duraid Madina
  − Progress has been slow, due to lack of time and other commitments

• Current implementation:
  − Basically working, very few miscompilations
  − Missing many simple optimizations
  − Has trivial stop bit insertion, but no bundle aware hazard recognizer
  − No post-pass scheduling, predication, prefetching, modsched, etc
  − ~4000 lines of code (.cpp, .h, .td)

• Generated code is about 50-60% the performance of GCC
• When assembled with IAS, LLVM beats GCC on many programs
  − IAS is an ‘optimizing assembler’, which does scheduling/bundling

A small investment can go a long way!
LLVM Summary

• LLVM is a modular compiler infrastructure:
  − Primary focus is on providing good interfaces & robust components
  − LLVM can be used for many things other than simple static compilers!

• LLVM provides language- and target-independent components:
  − Does not force use of JIT, GC, or a particular object model
  − Code from different languages can be linked together and optimized

• LLVM is well designed and provides aggressive functionality:
  − Interprocedural optimization, link-time/install-time optimization today!

• LLVM 1.7 was released last week:
  − Huge number of new features, many codegen improvements
  − Give it a try: http://llvm.org/releases/

clattner@apple.com